COMPONENTXCHANGE: AN E-EXCHANGE FOR SOFTWARE COMPONENTS

Sriram Varadarajan
Bell-Labs, Lucent Technologies
Murray Hill, NJ 07974, USA
vsriram@research.bell-labs.com

Atul Kumar, Deepak Gupta, Pankaj Jalote
Department of Computer Science and Engineering
Indian Institute of Technology
Kanpur - 208016, INDIA
[atul, deepak, jalote]@iitk.ac.in

ABSTRACT

A key challenge in Component-Based Software Engineering (CBSE) is finding appropriate software components for reuse. A component broker that allows component vendors to make available their modules, and also allows component integrators to search and select components matching their requirements will help solve this problem. In this paper, we describe an E-Exchange for buying and selling of software components over the World Wide Web. The ComponentXchange uses an XML based specification language for components, which is a semi-formal specification language that can describe a rich and an extensible set of component properties including functional properties, non-functional attributes and licensing aspects.

ComponentXchange supports two models of component trading. A component can be used either by downloading it and integrating it into the client application or by accessing it remotely over the network. It has a licensing server that enforces the licensing terms and conditions. ComponentXchange can be easily extended to support different payment models like pay-per-use, pay-per-user, etc.

KEYWORDS

Software Exchange, Component Specification, CDML, XML, Component Licensing.

1. INTRODUCTION

Component-based Software Engineering (CBSE) approaches are fast gaining importance in the mainstream software community. This interest is driven by the hope that these approaches will result in substantial cost savings and in software systems of higher quality. This is sought to be achieved by assembling systems from existing software components, instead of building them as monolithic applications from the ground-up.

Popular approaches in component-based software development emphasize the use of pre-existing commercial software components (or, Commercial-Off-The-Shelf (COTS) software components) in software development [18]. With COTS based software development, component development and component integration is done by different organizations. Component vendors engage in component development and component integrators or application developers perform component integration. The success of this approach clearly depends on the successful selection of COTS software components that meet the given requirements [5].

To aid component integrators in searching the right components produced by component vendors, there is a role for a mediator between them [1]. This mediator would perform the role of a component broker, and aid in the component acquisition phase of software development.

Given the increased use and reach of the Internet, it is clear that the mediator between vendors and integrators should use the Internet for the communication. Currently, a considerable amount of trading, auctioning, etc., is taking place through the Internet and many E-exchanges for different commodities already exist. Building an E-Exchange for components over the Internet will allow vendors from all over the world to build and sell components to integrators all over the world. Such an exchange where an integrator can choose components from multiple vendors will benefit the integrators. They, in turn, can encourage more vendors to publish their components making selection of components ever easier.

In this paper, we describe the design and implementation of ComponentXchange - an E-Exchange for software components. It creates a software component marketplace where component integrators and component vendors can buy and sell software components. Component Integrators are provided with an infrastructure for searching through the commercial software components produced by component vendors across the world and locating the software components that best match their requirements.

Components are specified using an XML based specification language: the Component Description Markup Language (CDML). CDML allows specification of functional as well as non-functional properties of components. It can also be used to specify the licensing terms and conditions of a component - which are very important for a marketplace.

Component integrators access components purchased through ComponentXchange in two ways. Either the component is downloaded to the client and later integrated into the client application, or its services are accessed remotely over the network.

The rest of the paper is organized as follows. In section 2, we discuss how components are specified and provide a description of the Component Description Markup Language (CDML) that can be used by component providers for publishing their components. In section 3 we describe how a component is searched and retrieved by a integrator. The design and implementation of ComponentXchange are described in section 4. We present a brief survey of the related work from the literature in section 5. Section 6 concludes the paper with a summary.

2. SPECIFICATION OF SOFTWARE COMPONENTS

An exchange will require all the components being sold to be specified properly so a buyer can search for it and take an informed decision about buying it. For specification, the properties of components can be partitioned in four categories - syntactic, behavioral, synchronization and quality-of-service [2].

Syntactic aspects of a component, also refered to as its interface signature, characterize its functionality and form the basis of all other aspects of component interface. CORBA IDL is a language that can be used for this purpose [13].

Behavioral specifications define the *outcome* of operations. Approaches to specifying behavior range from informal methods to highly formal ones. Informal specifications generally rely on textual descriptions of component behavior in natural language. Formal approaches use formal languages to specify the behavior.

Non-functional properties, or Quality attributes, include Quality-Of-Service (QoS) properties such as performance, reliability, availability and global attributes of a component such as portability and adaptability among others. NoFun [7] is an example that provides a notation for describing non-functional attributes like time and space efficiency, reusability, maintainability, reliability and usability. The QoS Modeling Language (QML) [8] uses the abstraction of QoS contracts to represent various QoS properties like reliability, performance, availability, etc. of distributed objects.

We now describe the Component Description Markup Language (CDML), the XML based specification language, that is used for specifying components in ComponentXchange. CDML is expressive enough to capture most of the properties of a component that a component integrator may consider while searching for a component satisfying his/her requirements, including non-functional and licensing aspects. It also permits efficient and automated searching for components. It is extensible and allows specification of new user-defined perspectives or aspects of software components.

2.1 Aspect Based Component Description

CDML follows an aspect based component description approach for providing a framework for specifying all systemic and non-systemic aspects of a software component in a uniform, consistent and a sufficiently formal manner so as to enable efficient component selection and retrieval.

Aspects are horizontal slices of a system's functional and non-functional properties. The concept of component aspects provides a better categorization of a component's capabilities, allowing the specification of different properties to be organized in a better manner. Component aspects provide a framework for multi-perspective specification of components, allowing a component developer or integrator to describe or view a component's capabilities from different perspectives.

In CDML, a component is specified by a collection of aspects. Aspects themselves are grouped in different aspect categories. CDML supports the following aspect categories: Syntactic aspects, Functional aspects, Non-Functional aspects, and Licensing and Commerce aspects.

The model for representing syntactic aspects in CDML is similar to that followed in the CORBA Component Model (CCM) [12]. The syntactic aspects of a component specify the following:

- *Provided Interfaces*: This is the set of interfaces that a component exposes to a client. An interface consists of an interface name, a set of *methods* and *attributes*. A method is specified by a method name, the type of result returned, the parameters passed to it and the exception thrown by the method on its invocation. An attribute consists of an attribute name and its data type.
- *Required Interfaces*: A component may depend on the services of other components to provide its services. Required interfaces specify those interfaces that other components must have as provided interfaces.
- *Events*: The set of events generated by a component or the set of events, which a component responds to, are specified. An event is identified by an event name and its direction - *out* if the event is generated, and *in* if the component receives the event from another component.

Functional aspects of a component are specified as a set of properties using a semi-formal approach. Each property is represented by a *name-value* pair.

For specification of *non-functional aspects*, CDML borrows from QML [8]. A non-functional characteristic of the component is specified as a *contract*. A contract is specified by a set of constraints along multiple dimensions. A constraint consists of a name, an operator, and a value. The name refers to the name of the dimension, or a *property* of dimension. *Dimension properties* allow for more complex characterizations of constraints. They can be used for characterizing measured values over some time period.

The licensing and commerce aspects are specified using license types. License types define the scope and use of a specific software component. For example, in concurrent license type, the charges may be based on the number of simultaneous connections to the component. On the other hand, in a pay-per-use license type, charges are calculated on a per-use basis.

2.2 XML-based Aspect Representation

We use XML and its related technologies for the actual specification of a component using the approach of aspect based component description. We have chosen XML because of its expressive power which allows representing complex information structures easily. We use the emerging XML Schema standard [19] as a meta-language for specifying the syntax and structure of CDML. XML Schema supports rich data types and allows validation of information based on information models represented in schemas.

The CDML XML schema consists of a set of core schema type definitions. A CDML document representing a component specification, generally, contains the following key elements: *Component, SyntacticAspects, FunctionalAspects, NonFunctionalAspects* and *LicensingCommerceAspects*. The core types defined by CDML schema specify the structure and constraints on these elements.

The example shown below is the CDML specification of a relational database component. This component can be used as a backend in software applications. It provides standard interfaces for querying, updating and administering the database.

```
<Component URI="corbaloc://db.com/oracle/"
  AccessType="Service"
  ComponentType="CORBA" >
  <SyntacticAspects>
```

```
      <Interface Name="Query">
        <Method Name="Select" ReturnType="RecordSet">
          <Parameters>
            <Parameter Name="QryString" Type="string"/>
          </Parameters>
          <Exceptions>
            <Exception Name="SQLException"/>
          </Exceptions>
        </Method>
      </Interface>
      <!-- Other interfaces like Update, Admin,..-->
    </SyntacticAspects>
    <FunctionalAspects>
      <Properties>
        <Name="MAXTABLES" Value="1024"/>
        <Name="MAXTRIGGERS" Value="16"/>
      </Properties>
    </FunctionalAspects>
    <NonFunctionalAspects>
      <Contract Type="http://www.iso.org/performance.xml">
        <Dimension Name="Delay" units="msec">
          <Properties>
            <Property Name="Mean" Value="50"
                 Relation="LT">
            <Property Name="Variance" Value="0.001"
                 Relation="LE">
          </Properties>
        </Dimension>
      </Contract>
    </NonFunctionalAspects>
    <Licensing_Commerce_Aspects>
      <LicenseType>Pay-Per-Use</LicenseType>
      <PricingStructure>
        <Pricing InterfaceName="Query" Cost="0.1"
             Currency="dollar"/>
        <Pricing InterfaceName="Update" Cost="2"
             Currency="dollar"/>
        <Pricing InterfaceName="Admin" Cost="10"
             Currency="dollar"/>
      </PricingStructure>
    </Licensing_Commerce_Aspects>
  </Component>
```

As we can see, the database component is uniquely identified by a URI. The component is characterized by syntactic, functional, non-functional and licensing-commerce aspects. The AccessType (in this case "service", i.e., a user can use it but not download it) and the component type are also specified. The *Component* element has four sub-elements, *SyntacticAspects*, *FunctionalAspects*, *NonFunctionalAspects*, and *Licensing_Commerce_Aspects*, each of which represents an aspect category.

We can see that the component has an interface named *Query*, which has a method named *Select*. The method returns an object of type *RecordSet* and accepts a parameter named *QryString* of data type *string*. It throws an exception named *SQLException*.

In the example, the functional properties specify that the maximum number of tables that can be created at a time is 1024 and the maximum number of triggers is 16.

The example contains a single contract specifying the component's non-functional aspects. The contract specifies the *performance* non-functional aspect of component, as identified by the URL supplied in the Type attribute of *Contract* tag. It contains a single dimension, *Delay*. We see that the performance contract is specified by the following two constraints.

Mean(Delay) < 50 msec

Variance(Delay) <= 0.001 msec

Lastly, the licensing and commerce aspects of the component are specified in the example. Customers can purchase the component's services on a *Pay-per-use* basis. Further, commerce information is also provided regarding the cost of using the service. It states that a user needs to pay 0.1 dollars for using the *Query* interface, 2 dollars for *Update* interface and 10 dollars for using the *Admin* interface.

3. COMPONENT LOCATION AND RETRIEVAL

In ComponentXchange, the searching and retrieval is done through Matchmaking. For a given specification of client requirements, the matchmaking module of ComponentXchange retrieves a set of potentially useful components for clients.

In general, Matchmaking involves comparing an encoded description of client requirements to the encoded descriptions of the components in the repository to select those components that closely match client requirements. There are two basic approaches to matchmaking [10]:

- *partial-order based retrieval*: In this approach, retrieval is based on the existence of a partial order relation (logical implication) between component description and client query. The algorithm retrieves all those components whose encoded descriptions logically imply the client query.
- *distance-based retrieval*: In this approach, a distance metric is computed for each component with respect to the given client query. This metric is used to retrieve components. Components having a smaller distance value are considered a better match compared to a component having a larger distance value. A distance value of 0 implies that the component fully satisfies all the requirements specified by the client.

Our approach is based on partial order matching. For a given client query, only those components that satisfy *all* the constraints specified in the query are retrieved.

In ComponentXchange, a client query is organized into a set of aspect categories and client requirements are specified along these aspect categories. The matchmaking is performed by multiple matchmaker components each matchmaker specializing in a particular aspect category. A matchmaker component compares the client queries and component specifications with respect to its aspect category. For example, a syntactic aspects matchmaker accepts a client query and does matchmaking only along syntactic aspects. It returns those components that satisfy the constraints specified in the *SyntacticAspects* section of the client query.

The interface of the matchmaking module is exposed through the dispatcher component. The dispatcher component accepts a client query and returns a list of components that satisfy the client query. The dispatcher invokes multiple matchmakers to perform matchmaking. It splits the client query along multiple aspect categories and sends each part to the corresponding matchmakers specializing in the particular aspect category. The dispatcher splits a query into multiple sub-queries which are sent to their respective matchmakers. The dispatcher determines the final result by computing the *intersection* of the results returned by individual matchmakers.

We describe the functioning of individual matchmakers through an example. Consider the client query shown below.

```
<Component>
  <SyntacticAspects>
    <ProvidedInterfaces>
      <Interface>
        <Method Name="Select" ReturnType="RecordSet">
        </Method>
      </Interface>
    </ProvidedInterfaces>
  </SyntacticAspects>
  <FunctionalAspects>
    <Properties>
      <Property Name="MAXTABLES" Value="500"
           Relation="GE"/>
    </Properties>
  </FunctionalAspects>
  <NonFunctionalAspects>
    <Contract
    Type="http://www.iso.org/performance.xml">
      <Dimension Name="Delay" units="msec">
        <Properties>
          <Property Name="Mean" Value="60"
           Relation="LT">
        </Properties>
      </Dimension>
    </Contract>
  </NonFunctionalAspects>
```

```
    <LicensingCommerceAspects>
      <LicenseType>Pay-Per-Use</LicenseType>
      <PricingStructure>
        <Pricing Cost="0.2" Currency="dollar"/>
      <PricingStructure>
    </LicensingCommerceAspects>
  </Component>
```

In this example shown, there are two constraints: the component should have a method named “Select”, and it should return a “RecordSet”. The syntactic matchmaker returns only those components that satisfy *both* these constraints.

Property predicates are used to represent the constraints on functional properties that the client is interested in. In the above example, the client imposes a constraint “MAXTABLES > 500”, i.e, the database should allow the creation of at least 500 tables. The matchmaker selects all components that satisfy the specified constraint.

The relation of *contract conformance* between contracts is used for matchmaking of non-functional aspects. A contract conforms to another contract, if all its constraints conform to those of the other contract. Constraint conformance defines when one constraint in a contract can be considered stronger, or as strong as, another constraint for the same dimension in another contract.

To define constraint conformance, we need the notion of stronger and weaker elements in the domain of values of a dimension. In some cases, a lesser value may be conceptually stronger than a larger value. For example, in dimensions such as latency, smaller numbers represent stronger commitments than larger numbers. Therefore, for every dimension we need to specify whether smaller domain elements are stronger than or weaker than larger domain elements. The **decreasing** and **increasing** qualifiers in the dimension declaration provide that information. If a dimension is declared decreasing, we map “stronger than” to “less than” (<). Thus a value is stronger than another value, if it is smaller. An increasing dimension maps “stronger than” to “greater than” (>). The semantics will be that larger values are considered stronger. For example, “latency < 10” conforms to “latency < 20” as the dimension latency is decreasing.

In the example client query given earlier, the matchmaker for non-functional aspects searches the component description repository to locate those components that *conform* to the client requirement contract. This matchmaker will select the database component as its contract conforms to the client contract (delay < 50 msec conforms to delay < 60 msec).

For the licensing requirements, the matchmaker retrieves all components that support the *Pay-Per-Use* license type *and* which costs less than or equal to 0.2 dollars per use. In general, we can have any of the common license types like basic, capacity, concurrent, etc.

4. DESIGN AND IMPLEMENTATION

ComponentXchange is a web-based online intermediary that connects a large group of component integrators and component vendors. It is modeled on a particular class of online e-marketplaces called *vertical hubs* or *fat butterflies*. The term *fat butterfly*, in describing e-markets, comes from the depiction of numerous suppliers representing one wing of a butterfly and numerous customers as the other.

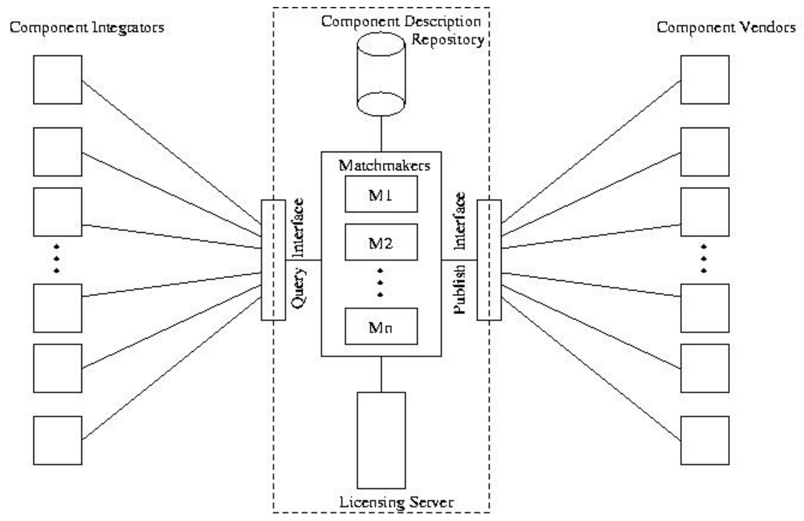

Figure 1. Fat Butterfly model of ComponentXchange

Figure 1. depicts the “Butterfly” model of ComponentXchange highlighting the system's major components and their interaction with component integrators and vendors. Group of component integrators represent one wing and a group of component vendors the other. ComponentXchange acts as the *hub* connecting the integrators and vendors. In this design, a larger participation by vendors and integrators ensures greater benefits to all the participants, i.e., *fatter* the system, the better it is.

Component integrators and component vendors interact with ComponentXchange through well-defined interfaces. ComponentXchange exposes two interfaces to interact with the outside world: the *publish* interface and the *query* interface. Component vendors use the *publish* interface to publish component descriptions using a specification language. The *query* interface is used by component integrators to search for software components. At the heart of ComponentXchange is matchmaking, which is done using a dynamic set of matchmakers. ComponentXchange uses a licensing server to enforce licensing terms and conditions of components sold through the system.

We followed a component-based approach for building ComponentXchange. We used many pre-existing components like the KWEELT XML Query engine [15] for implementing most of the matchmaking functionality.

The desire to use pre-existing components in ComponentXchange affected some of the implementation decisions. We used plain XML file as an XML database primarily due to lack of freely available XML databases that supported a rich query interface. Though this choice will not permit ComponentXchange to scale up beyond a point, it is suitable for small exchanges.

4.1 Publishing Components

Component Vendors are required to publish their software components to ComponentXchange if they wish to sell their components through it. Vendors need to provide CDML specifications of components to the exchange for publishing their components. The design provides for tools to partially automate the specification of components by component developers. For example, syntactic interface of a component can be discovered at run-time using *introspection* techniques.

ComponentXchange provides a *publish* interface, which is a web-based interface for component vendors to register their components with ComponentXchange. To register a component, component vendor submits a document containing the CDML specification document using the publish interface.

The publish interface basically accepts component specification documents submitted by vendors and stores them in the component description repository. The component description repository is a database for storing CDML specification of components registered with ComponentXchange. Matchmakers use this repository to match component descriptions stored in the repository with client requirements.

The interfaces for component vendors and component integrators respectively have been implemented using the Java Servlets API. Vendors are provided with a HTML form for specifying the location of the XML document containing the component specification. The browser sends the XML document specified by the user to the Publish Servlet, using the POST method. The Publish Servlet extracts the XML file sent by the user from the POST payload, and then invokes an XML parser to validate whether the sent document conforms to the CDML schema. If the document conforms to CDML, it is stored in the component description repository, else, an error is notified to the user.

Component description repository is a database containing the CDML descriptions of all components registered with ComponentXchange. The repository is implemented as a plain XML file. We used plain XML file as an XML database primarily due to lack of freely available XML databases that supported a rich query interface. The publish interface uses a standard XML parser for adding new component descriptions to the repository. The Kweelt query engine also employs a standard parser for reading from the repository.

4.2 Querying for Components

The web-based *query* interface is provided for component integrators to search for components.

The query interface accepts a client requirement document as input and returns a list of components that satisfy the requirements. It uses the matchmaker module to retrieve components satisfying client requirements. The document submitted by the client is an XML based encoding of client requirements. The output is an XML document that gives a list of components satisfying the input query. However, before the output is sent to the client, it is converted to HTML.

For representing client requirements, an XML schema is used that largely conforms to the CDML Schema. While in principle, CDML can be used without modification for specifying client queries, this approach is rigid and does not allow clients to specify those properties about which the client has only partial knowledge. In general, all differences of component requirement specification schema with CDML pertain to relaxing the rules for specification of components so as to accommodate partial specifications.

The Query interface provides tools for helping component integrators to formulate complex queries, apart from providing the ability to query the system for software components. A component integrator is presented with a page containing an applet, which helps the integrator to formulate complex queries. Apart from providing an easy-to-use, intuitive interface, the applet also ensures that the queries submitted to the system conform to the requirement specification schema.

The functioning of the *Query* Servlet is simple. It accepts an XML document representing a query splits the query along multiple aspect categories and forwards the set of sub-queries to the *Dispatcher* component of the matchmaking module. It sends a hashtable containing aspect category name as key and the corresponding sub-query as value. The Dispatcher component returns a list of URLs identifying the set of components that match the client query. The Query Servlet generates an HTML page for displaying the query results and sends it to the browser.

4.3 Matchmaking

The matchmaking module consists of a *Dispatcher* component and multiple *Matchmaker* components. The Dispatcher receives a set of *aspect category* specific queries, which it forwards to appropriate matchmakers. This component is implemented as a plain Java class and it provides a static method '*dispatch'* for dispatching client queries to various matchmakers. This component runs in the context of the Query Servlet.

All matchmakers are implemented as Servlets and the Dispatcher interacts with them using the HTTP POST method. The Dispatcher iterates over the set of aspect category specific queries and sends each query to the corresponding matchmaker as part of the HTTP payload. All Matchmakers return an XML string that contains a set of URLs that represent the matching components.

The interfaces exposed by the matchmakers and their interaction with the Dispatcher is standardized. The Dispatcher identifies the appropriate matchmaker component for a given input query by following a standard convention for naming the matchmakers. The Dispatcher determines the final result of matchmaking by computing the intersection of the results returned by individual matchmakers. A component is returned as part of the result only if all the matchmakers match it.

All matchmakers, except the matchmaker for non-functional aspects, were built using the KWEELT XML Query engine [15]. The matchmaker for non-functional aspects was implemented using XML parsers and standard data structures available as part of the Java 2 Standard API.

The decision to use Kweelt was based on our requirement for an XML engine that processes a rich set of queries. In our survey of different XML query languages currently implemented, Quilt XML query language seemed to be the most powerful and flexible one. Kweelt is currently the only available XML engine that provides support for Quilt.

The primary function of matchmakers built using Kweelt engine is to accept input queries represented as an XML string and convert them into queries of the Quilt query language. The matchmaker invokes the Kweelt query engine through the Java interfaces exposed by the engine. The result returned by Kweelt contains the set of components matching the given query. The matchmaker sends the results to the Dispatcher component.

4.4 Licensing Support

In ComponentXchange, licensing support is provided through a licensing server. When integrators purchase a component through ComponentXchange, they are provided with a unique license key, which allows them to access component services. This license key is used in interactions between the *sold* component and the licensing server to ensure that the licensing terms are being met and also to prevent unauthorized access to component services. The interaction between licensing server and components is through a standard API that is similar to CORBA licensing service.

ComponentXchange supports three types of licensing: *pay-per-use*, *pay-per-user* and *pay-per-time*. In pay-per-use, a component integrator purchases some units of usage of a component. To ensure that the component is used only for the specified number of times, a license key is generated for every purchase and sent to the buyer and the licensing server. When the purchased component is integrated into an application and its services are invoked, the component sends this information to the licensing server along with the name of the interface and the method being currently accessed by the client. Depending on the number of units of usage available for the particular license key, the license server allows or disallows the component to service the request.

Similar is the implementation for pay-per-time. In this, instead of number of accesses, the license server keeps track of the amount of time a component is used by a user. In pay-per-user, a set of keys is generated and given to the buyer depending on how many licenses the buyer is purchasing. Any user must have one of the valid keys, thereby allowing the number of concurrent users to be limited by the number of keys issued.

An important issue in enforcing licensing is the overhead of sending messages periodically over the Internet for licensing purposes. This issue could become very severe in cases of fine-grained software components. One way to solve this issue could be to integrate a micro-payment protocol in our licensing framework. There are some very efficient off-line micro payment protocols that avoid sending messages across the network frequently.

For trading components through ComponentXchange, it is necessary that the components be *license-aware*. There are two general approaches in which components can be license-enabled. One is to provide code-generators that take the source code of the component as input and embed licensing calls at appropriate places. The other approach is useful in cases where the source code of the component is not available. In this approach, tools are provided for generating wrappers around components so that they are made license-aware.

Licensing support is provided in the system using a Licensing Server. Licensing server exposes a standard API for interaction with licensed components and ComponentXchange. The API has been exposed through both CORBA and Servlet interfaces. The reason for providing multiple interfaces is to provide licensing service API to component belonging to different component models. For example, Java beans components would use Servlet interfaces while CORBA components would use CORBA interfaces.

The licensing server uses a lightweight relational database for storing license keys and other license and payment information persistently.

5. RELATED WORK

This section presents a review of some of the important literature in the related fields of software component specification and retrieval. We also discuss the important features provided by a few existing commercial software component marketplaces on the Web.

A lot of work relates to extending conventional interface specification techniques to support a richer and a more descriptive specification of software components. Formal specification languages like VDM, Z and Larch have been developed to specify the behavior of software components. They provide complete and consistent description of components and retrieval systems based on these could be very precise. However, these languages are very complex and are not widely used in the mainstream software industry. One of the main reasons for choosing a semi-formal approach in ComponentXchange was to allow users to specify and search for components easily.

At another extreme are text-based component retrieval systems. In a text based method, the functionality of the component is described as plain text in natural language and string pattern matching is used for searching and retrieving the relevant software components. A number of software libraries [6, 9] use plain text encoding and search for component retrieval. Though simple to use and inexpensive, text-based approach does not produce precise results and requires human interpretation [3].

AI and Knowledge Representation techniques are also used to specify the semantics of components in a better manner. In [14], conventional traders are enhanced with semantic networks to support the cognitive domain of application users, through learning of new ways to describe services. Adaptive READ is another interesting project that uses AI techniques for information retrieval. It acquires knowledge by observing previous search processes from users and adapting to user needs based on it. AI based approaches are important to the field of component retrieval. However, these systems are complex to build and they are not used widely.

Agora is a search engine for software components on the web [16]. It provides agents that search the Web for components. It uses the technique of introspection for dynamically discovering the syntactic interfaces of the component, which are indexed and stored by the search engine. Its main drawback is that it relies only on the syntactic aspects of a component to perform its search operation.

A few commercial Web-based software component marketplaces have emerged like Flashline.com and Componex (http://www.componex.biz). These marketplaces provide a relatively simple search function but allow browsing of their component repository by organizing and classifying software components according to various parameters like technology/platform, domain, etc. The approach followed by the Componex system is quite similar to ComponentXchange. In Componex, components are described using a well-defined XML schema and its search function allows users specify certain key elements like component type, technology type, etc. While the schema for component specification in Componex provides a good framework for classifying components and allows for specification of many attributes of a software component, it is not as flexible as ComponentXchange with regards to extending component specifications. In ComponentXchange, the component specification language, CDML, could be easily extended to support new user-defined perspectives (or, aspect-categories) on components. Matchmakers supporting new aspect-categories could then be dynamically plugged-in to the system. In general, commercial approaches sacrifice precision of search for ease of use.

6. CONCLUSION

With the increasing popularity of Component Based Software Engineering approaches, a large number of commercial off-the-shelf (COTS) components are being made available to component integrators and developers by component vendors the world over. However, a key challenge in using COTS components to build software systems is to search and locate components that best match given requirements. In this work, we have attempted to provide a solution to this problem by building ComponentXchange, a Web-based software component exchange that acts as an online intermediary between component integrators and component vendors, allowing buying and selling of software components.

An important issue present in building component exchanges and component retrieval systems in general, is the specification of software components. A richer specification of software components provides a foundation on which better and effective search and retrieval facilities can be built. For the purpose of specifying components in ComponentXchange, we have developed an XML based specification language for software components called the Component Description Markup Language (CDML). The primary objective of designing CDML was to provide developers with an easy-to-use notation for specifying components, while allowing components to be specified in a sufficiently formal manner. ComponentXchange defines an extensible framework for specifying components' capabilities along multiple aspects. Currently, we have defined a vocabulary for capturing syntactic, functional, non-functional, licensing and commerce aspects of components.

To enable e-commerce of software components, ComponentXchange provides facilities for licensing and usage tracking. ComponentXchange uses a licensing server to provide the licensing support. Components interact with licensing server to ensure that licensing terms are being met. Further details about the different aspects of the ComponentXchange are discussed in [17].

REFERENCES

1. Aoyama, M., 1998. New Age of Software Development: How Component-Based Software Engineering Changes the Way of Software Development. *Proceedings of International Workshop on Component-Based Software Engineering*. Kyoto, Japan.
2. Beugnard, A. et al, 1999. Making components contract aware. *In IEEE Computer*, 32(7), pp 38-45.
3. Blair, D. and Maron, M. E., 1985. An Evaluation of Retrieval Effectiveness for a Full-Text Document-Retrieval System. *In Communications of the ACM*, 28(3), pp 289-299.
4. Burton, B. et al, 1987. The Resuable Software Library. *In IEEE Software*, pp. 25-33.
5. Ncube, C. and Neil A.M. Maiden, 1999. PORE: Procurement-Oriented Requirements Engineering Method for the Component-Based Systems Engineering Development Paradigm. *Proceedings of International Workshop on Component-Based Software Engineering*, Los Angeles, CA.
6. Frakes W. B. and Nejmeh, B. A., 1990. An Information System for Software Reuse. *In Software Reuse: Emerging Technology*, IEEE Computer Society Press, pp. 142-151.
7. Franch, X., 1998. Systematic Formulation of Non-Functional Characteristics of Software. *Proceedings of 3rd IEEE International Conference on Requirements Engineering,* Colorado Springs, CO.
8. Frolund, Svend and Koistinen, Jari, 1998. QML: A language for quality of service specification. *Technical Report HPL-98-10*, Hewlett-Packard Laboratories.
9. Maarek, Y. S. et al. An Information Retrieval Approach for Automatically Constructing Software Libraries. *In IEEE Transactions on Software Engineering,* 17(8). pp. 800-813.
10. Mili, H. et al, 1995. Reusing Software: Issues and Research Directions. *In IEEE Transactions On Software Engineering,* 21(6), pp. 528-562.
11. Mili, Hafedh et al, 1994. Practitioner and Softclass: A Comparative Study of Two Software Reuse Research Projects. *In Journal of Systems and Software*, vol 27.
12. Object Management Group (OMG). CORBA Component Model. *OMG TC Document orbos/99-02-05*. http://www.omg.org/cgi-bin/doc?orbos/99-02-05.
13. Object Management Group (OMG). CORBA Services: Common Object Services Specifications. http://www.omg.org/cgi-bin/doc?ptc/99-10-04
14. Puder, A. et al, 1995. Service Trading Using Conceptual Structures. *Proceedings of the Third International Conference on Conceptual Structures*, Santa Cruz, California.
15. Sahuguet, Arnaud, 2000. KWEELT, the Making-of: Mistakes Made and Lessons Learned. *Technical Report*, Department of Computer and Information Science, University of Pennsylvania.
16. Seacord, Robert et al, 1998. Agora: A Search Engine for Software Components. *Technical Report CMU/SEI-98-TR-011*, Software Engineering Institute, Carnegie Mellon University.
17. V. Sriram, 2001. ComponentXchange: An E-Exchange for Software Components. *M.Tech. Thesis,* Department of Computer Science & Engineering, Indian Institute of Technology, Kanpur.
18. Wallnau, K. C., 1999. On Software Components and Commercial ("COTS") Software. *Proceedings of International Workshop on Component-Based Software Engineering*. Los Angeles, CA.
19. XML Schema Part 0: Primer. W3C Candidate Recommendation, World-Wide Web Consortium, October 2000. http://www.w3.org/TR/xmlschema-0/