A Trading Service for COTS Components

LUIS IRIBARNE[1], JOSÉ M. TROYA[2] AND ANTONIO VALLECILLO[2]

[1]*Dpto. Lenguajes y Computación, Escuela Politécnica Superior, Universidad de Almería, Carretera de Sacramento s/n, 04120 Almería, Spain*
[2]*Dpto. Lenguajes y Ciencias de la Computación, Facultad de Informática, Universidad de Málaga, Campus de Teatinos, 29071 Málaga, Spain*
Email: luis.iribarne@ual.es, troya@lcc.uma.es, av@lcc.uma.es

Component-based software development (CBSD) has gained recognition as one of the key technologies for the construction of high-quality, evolvable, large complex systems in a timely and affordable manner. In CBSD, the development effort becomes one of gradual discovery about the components, their capabilities and the incompatibilities that arise when they are used in concert. Thus, trading becomes one of the cornerstones of CBSD. However, most of the existing methods for CBSD do not make effective use of traders. In this paper, we analyze the required features for commercial off-the-shelf (COTS) components traders, and introduce **COTStrader**, an Internet-based trader for COTS components. In addition, we discuss how the **COTStrader** can be integrated into a spiral methodology for CBSD, providing partially automated support for building COTS-based systems.

1. INTRODUCTION

Component-based software development (CBSD) is generating an increased interest due to the development of plug-and-play reusable software, which has led to the concept of 'commercial off-the-shelf' (COTS) software components. CBSD moves organizations from application development to application assembly. Constructing an application now involves the use of prefabricated pieces, perhaps developed at different times, by different people and possibly with different uses in mind. The ultimate goal is to be able to reduce development times, costs and efforts, while improving the flexibility, reliability and reusability of the final application due to the (re)use of software components already tested and validated.

This approach challenges some of the current Software Engineering methods and tools. For instance, the traditional top-down development method, based on successive refinements of the system requirements until a suitable concrete implementation of the final application's components is reached, is not transferable to the component-based development. In CBSD the system designer has also to take into account the specification of pre-developed COTS components that may exist in software repositories when building the system's initial requirements, in order to incorporate them into all phases of the development process [1, 2, 3]. There is a significant shift from a development-centric approach toward a procurement-centric one [4], aimed at the search and acquisition of COTS components in order to reuse them for building software applications. Here, system architects,

designers and builders must accept the trade-offs among three main concerns: users' requirements, system architecture and COTS products [5, 6].

Under this new setting, the proper search and selection processes of COTS components have become cornerstones of every effective COTS development. However, these processes currently face serious limitations, generally due to three main reasons. First, the information available about the components is not expressive enough for their effective selection. Second, the search and evaluation criteria allowed by current traders are usually too simple to provide practical utility. And finally, current CBSD methodologies do not make effective use of trading for searching and locating components offering the required services.

First, one of the key issues in CBSD is the use of more complete, concise and unambiguous component documentation (i.e. specifications). In the case of COTS components, their black-box nature hinders the understanding of their internal behavior. Furthermore, only functional properties are usually taken into account, while some other information crucial to component selection is missing, such as protocol or semantic information [7] or extra-functional requirements [8, 9]. Software component vendors are also of no help, providing scarce and unstructured information about the components they sell/license [10].

Second, component searching and matching processes are (in theory) delegated to traders, but the problem is that existing traders do not provide all the functionality required for an effective COTS component trading in open

and independently extensible systems such as the Internet, as discussed in [11].

Finally, traders are not fully integrated into current methodologies to achieve effective component-based development, hence missing many of the potential advantages provided by traders such as information discovery or partially automated selection of candidate components.

In this paper, we focus on the requirements for an effective trading service for COTS components in open systems; we analyze some of the limitations of current traders; and we present a trading service that tries to address most of their current shortcomings. An extension of the information used for describing COTS component services is also presented, where not only are the functional aspects of the components taken into account, but also extra-functional requirements, architectural constraints and some other non-technical factors. Based on this information, a template for describing services queries is defined. With all of this, it is possible to improve both the service 'export' and 'import' processes, and to design and develop enhanced traders that make use of the Internet facilities for locating and retrieving COTS components. Finally, we will see how it is possible to integrate the trader into spiral methodologies for building component-based systems, providing partially automated support for the search and selection processes of COTS components that match some of the architectural constraints. This kind of spiral methodology is the one commonly used in CBSD (see, e.g. [12]). It progressively produces more detailed requirements, architectural specifications and system designs by repeated iterations, until a stable solution is found.

Throughout the document, we will consider components to be binary units of possibly independent production, acquisition and deployment that interact to form a functioning system [13]. Although our proposal is valid for any kind of software components, we have focused mainly on COTS components. The adjective COTS will refer to a special kind of component: a commercial entity—i.e. one that can be sold or licensed—that allows for packaging, distribution, storage, retrieval and customization by users and which is usually coarse-grained and lives in software repositories [14].

The rest of the paper is organized as follows. Section 2 describes the features that traders should have, and discusses the shortcomings of current traders. Section 3 defines the concepts of 'service' and 'service type' that will be used in our context. Then, Section 4 describes the COTStrader, an implementation of an Internet-based trader for COTS components, together with a proposal for documenting components. Section 5 discusses how the trader can be effectively used within spiral methodologies for building COTS-based systems, providing automated support for CBSD. We have also looked at the situation of existing COTS component vendors, trying to analyze how large the gap is between the information they currently provide and the information needed for effective trading. The results are presented in Section 6. Finally, Sections 7 and 8 relate this work to other similar approaches and draw some conclusions, respectively.

2. REQUIREMENTS FOR COTS TRADERS

Trading is the natural mechanism defined in object- and component-based systems for searching and locating services. A client role that requires a particular component service can query a matchmaking agent (the trader) for references to available components that provide the kind of service required. Service advertisements are called 'exports', while queries are called 'imports'. The trader provides just the references to possible service providers, but without intervening in the service provision itself.

Trading is not only relevant to CBSD. Context-aware software, such as mobile computing or adaptable applications, can greatly benefit from trading since it provides service discovery in local environments and enables automatic application re-contexting. Moreover, enhanced traders with quality of service (QoS) facilities can provide the means of self-configuring multimedia applications. In this paper, we will concentrate only on COTS component trading. However, most of our discourse is also applicable to all disciplines in which trading is required.

2.1. Trading requirements

The following list presents the features and characteristics that we think traders should have in order to provide an effective COTS component trading service in open systems.

(i) *Heterogeneous component model*. A trader should not restrict itself to a particular component or object model, but it should be able to (simultaneously) deal with different component models and platforms, such as CORBA, the CORBA Component Model (CCM), Enterprise Java Beans (EJB), Microsoft's Component Object Model (COM) and .NET etc. Heterogeneous traders should trade with multiple service access protocols, and accommodate the evolution of the current models.

(ii) *A trader is more than a search engine*. Traders may superficially resemble search engines, but perform more structured searches. In a trader, the matchmaking heuristics need to model the vocabulary, distance functions and equivalence classes in a domain-specific property space, in contrast to the keyword-based, domain-neutral matchmaking supported by search engines.

(iii) *Federation*. Cooperating traders can federate using different strategies. The direct federation approach requires traders to communicate directly to (and know) the traders they federate with. Although it is a very secure and controlled scheme, the communication overhead grows with the number of federated traders. In the repository-based federation, multiple traders read and write to the same service repository; traders are not directly aware of each other, so this approach is scalable. The problem is the implementation of a globally accessible repository. However, this may not be an issue in the Internet,

since search engines may naturally provide the basis for such a repository.

(iv) *Service composition and adaptation.* Current traders focus on one-to-one matches between client requests and available service instances. A compositional trader could also consider one-to-many matches, by which a client request can also be fulfilled by appropriately composing several available services, which together provide the complete services. Furthermore, one-to-one exact matchmaking is also inadequate in a number of situations, e.g. when format, QoS or performance mismatches happen between the client and the closest matching service instance. This problem can be addressed by composing such a service instance with format translators and performance monitors to reduce mismatches.

(v) *Multiple interfaces.* In object-oriented systems, services are described by interfaces, and each object provides just one interface (although it may be obtained from many by multiple inheritance). However, components may simultaneously offer several interfaces, and thus, services should be defined in terms of sets of interfaces. This fact has to be especially considered when integrating components, since conflicts between components may appear to be offering common interfaces [15].

(vi) *Softmatchmaking.* Traditional 'exact' matches between imports and exports are very restrictive in real situations in which more 'relaxed' matching criteria should be used. This is even more important when trading for services in open and independently extensible systems, such as the Internet, where method names and the operations that comprise the offered services are chosen in a non-standardized arbitrary way and without agreed procedures. Therefore, partial matches that also select those candidates that may provide (part of) a required service should be allowed when building the list of candidate components.

(vii) *Use of heuristics and metrics (preferences).* Users should be able to specify heuristic functions and metrics when looking for components, especially in the case of soft matches.

(viii) *Extended subtyping procedures.* Current traders organize services in a service type hierarchy in order to carry out the service matching process. Central to type matching is the notion of subtyping. TypeA is a subtype of TypeB (TypeA $\leqslant$ TypeB) if instances of TypeA can substitute instances of TypeB and clients are not able to detect the change [16]. Subtyping is now checked merely at the signature level, but extensions need to be defined in order to cope with 'behavioral' information [17], 'protocols' [18], QoS etc.

(ix) *Independent extensibility and scalability.* Component behavior, extra-functional requirements, QoS, marketing information and semantic data should also be considered. The information managed by the trader should be possible to be extended by users in an independent way, and still the trader should be able to use all its functionalities and capabilities. Furthermore, no matter how traders federate or organize, scalability should be guaranteed in large open distributed environments such as the Internet.

(x) *Support of both 'automatic' and 'store and forward' imports.* If a trader cannot fully satisfy a request, it can either automatically reply back to the client with a denial (automatic behavior), or it may also store the request and postpone the reply until a suitable service provider is found (store and forward import).

(xi) *Support of both 'push' and 'pull' models.* In a push model, exporters directly contact the trader to register their services. A probably more convenient way of working in a large open and distributed environment is by using a pull model, by which exporters make the publicity campaigns of their provided services available at their Web sites, for instance, and the traders are continuously looking for new services. Bots and search engines can be used to enhance current traders, which use a push model that crawls the Web looking for services and 'pushing' them into the traders.

(xii) *Delegation.* Traders should be able to delegate requests to other (smarter) traders, if they cannot resolve them. Delegation of the complete request or only parts of it should be desirable.

2.2. Shortcomings of current traders

Existing traders mostly follow the ODP trading model [19]. Although this is a complete and very well-designed trading model, it follows the general ODP (object-oriented) model, and therefore it presents some limitations when used in open systems. Actually, based on the experience obtained from the existing commercial implementations of the trading service (e.g. [20, 21, 22]) and on the basis of some closely related works and academic traders (e.g. [11, 23, 24, 25, 26]), we can see how current traders: (1) deal with homogeneous object models only; (2) use direct federation; (3) do not allow service composition or adaptation; (4) work with 'exact' matches, and only at the signature level; (5) do not deal with multiple interfaces; and (6) are based on a push model only.

Our main purpose is to design a trader that can help overcome these limitations; a trader specifically designed to deal with COTS components in Internet-based environments.

3. SERVICES AND SERVICE TYPES

Components offer services, clients request services and traders deal with services. Therefore, we should start by defining what a service is. We will adopt here the ISO definition of service given in the ODP trading service specification [19]: 'A service is a set of capabilities provided by an object at a computational level. A service is an instance of service type'.

In ODP, a service type is an interface signature type, a set of property definitions and a set of rules about the modes of the service properties. Interface signatures describe the service functionality in terms of attributes and methods (ODP is object-oriented). Service properties are triplets (name, value type, mode), where the name identifies the property (e.g. *AllowEncryption*), the value type establishes the type of allowed values for the property (e.g. Boolean) and the mode specifies whether the property is read-only or read-and-write, or whether it is optional or mandatory. In addition, properties can also be declared as dynamic, which means that instead of having a fixed value assigned, they have an external evaluator associated, which is in charge of dynamically providing the current value of the property (e.g. the current length of a queue).

This kind of ODP service type is the one commonly used in most current traders. However, we need to extend this definition in order to allow more complete descriptions of the services, and to accommodate the specific requirements of COTS component-based systems.

In our context, a service type will consist of four main parts. The first one describes the *functional* (i.e. computational) aspects of the service, including both syntactic (i.e. signature) and semantic information. Unlike the ODP service type which contains just one interface, our functional description of a service defines the sets of provided and required interfaces. Semantic information can be described at two different levels, depending on whether it specifies the behavior of the operations (which can be described using pre/post conditions, for instance), or the relative order in which a component expects their methods to be called, and the way it calls other components' methods—i.e. its service access protocols (also called choreography).

The second part describes the *extra-functional* aspects of the service (e.g. QoS, security etc.) in a similar way to ODP i.e. by means of services properties.

The third part contains the *packaging* information: about how to download, deploy and install the COTS component that provides the required service, including implementation details, context and architectural constraints etc.

Finally, the *marketing* information deals with the rest of the non-technical issues of the service, such as licensing and pricing information, vendor details, special offers etc.

4. A COTStrader FOR OPEN SYSTEMS

Once we have identified the requirements of a trader in open systems, this section describes the implementation of an Internet-based trader for COTS components, called COTStrader.

We have divided this section into three main parts. The first one deals with component and service documentation, i.e. how to describe the services provided by the COTS components in terms of their service types. The second part defines how clients may query the trader and express service imports. Finally, Section 4.3 describes the trading process of COTStrader.

We have used XML as the language for documenting components (i.e. describing services) and expressing queries. XML is simple, extensible and widely accepted within the Internet community.

In order to illustrate our proposal we will use a simple example, that of a one-place buffer component with the usual operations `read()` and `write()`. It also makes use of another component to print out the values written in its cell, using method `print()` every time a new value is written. Despite its simplicity, this example will allow us to illustrate our approach for documenting components, the export and import processes and the way the trader works. This example, together with several other examples and applications, is available at the COTStrader Web site http://www.cotstrader.com/.

4.1. Exporting services

For the description of a COTS component we have designed some Document Type Definition (DTD) templates based on the W3C's XMLSchema schema language (`http://www.w3c.org/2000/10/XMLSchema`). The following shows a simple skeletal instance of the schema template designed for the case of the `OnePlaceBuffer` component.

```
<?xml version="1.0"?>
<COTScomponent name="OnePlaceBuffer"
  xmlns="http://www.cotstrader.com/
    COTS-XMLSchema.xsd"
  xmlns:xsd="http://www.w3.org/2000/10/
    XMLSchema">
  <functional> ...   </functional>
  <properties> ...   </properties>
  <packaging> ...   </packaging>
  <marketing> ...   </marketing>
</COTScomponent>
```

As we can see in the XML document, the description of a component begins with a `COTScomponent` tag where the DTD name spaces used and the name of the component are established. Then, the body of the XML document describes the four main parts that a service type consists of, as defined in Section 3.

4.1.1. *Functional description*

The functional description of a component follows an approach similar to that of most common component models such as CCM, EJB or EDOC [27]. Actually, it can be used to describe components of any of these models. The information contained in this part includes the set of interfaces that the component implements, the set of interfaces that the component requires from other components, and the sets of emitted/consumed events.

```
<functional>
 <providedInterfaces>
   <interface name="interfaceA"> ...
      </interface>
```

```
    <interface name="interfaceB"> ...
      </interface>
  </providedInterfaces>
  <requiredInterfaces>
    <interface name="interfaceC"> ...
      </interface>
    <interface name="interfaceD"> ...
      </interface>
  </requiredInterfaces>
  <consumedEvents> ... </consumedEvents>
  <producedEvents> ... </producedEvents>
  <choreography> ... </choreography>
</functional>
```

The last part (`<choreography>`) allows the description of the 'service access protocols', which define the relative order in which the component expects its methods and events to be called, and the order in which it emits events and calls other components' methods. Semantic information can also be added to the syntactic description of the operations using a `<behavior>` tag.

Let us illustrate all this using the one-place buffer example, for which the first two parts can be described as follows.

```
<providedInterfaces>
 <interface name="IOnePlaceBuffer">
   <description notation="IDL-CORBA">
     interface IOnePlaceBuffer {
       void write(in long x); long read(); };
   </description>
   <exactMatching href="..../servlet/
     IDL-CORBA.exact"/>
   <softMatching href="..../servlet/
     IDL-CORBA.soft"/>
   <behavior>
     <description notation="Larch-CORBA">
     interface IOnePlaceBuffer {
        initially self' = empty;
        void write(in long x){
         requires isEmpty(self);
           modifies self;
         ensures self' = append(self^,x);
        }
        long  read() {
         requires ~isEmpty(self);
           modifies self;
         ensures isEmpty(self') /\
           result = head(self^);
        }
      }
     </description>
     <exactMatching href="..../servlet/
       Larch-CORBA.exact"/>
     <softMatching href="..../servlet/
       Larch-CORBA.soft"/>
   </behavior>
 </interface>
</providedInterfaces>
```

```
<requiredInterfaces>
  <interface name="Printer">
    <description notation="IDL-CORBA">
      interface Printer {void print
        (in long x); };
    </description>
  </interface>
</requiredInterfaces>
```

As we can see, not only can the syntactical description of the interfaces (i.e. their signatures) be expressed, but also their behavioral semantics using any notation, such as Larch-CORBA (http://www.cs.iastate.edu/~leavens/main.html#LarchCORBA). This notation allows the specification of the pre- and post-conditions of the methods implemented by the component.

The main information described in this XML template usually comes inside the `<description>` tag. This information can be either implicitly included in this tag, or a reference to an external location can be made using the `href` attribute.

We did not want to commit to any particular notation to express the information contained in the XML templates. Therefore, the 'notation' attribute is present in most fields. It currently has several pre-defined values, but it is a matter of clients and servers agreeing to the values they want to use. More than providing a syntax for importing and exporting services, our goal is to provide a template that can be used for clients and service providers to express the information they want to share; a template that is able to evolve as the market does.

Associated to each notation for describing behavior, there may also be a reference to a couple of procedures that will allow the trader to do the matchmaking process. In this example, 'Larch-CORBA.exact' is the name of a program that is able to decide whether two behavioral descriptions A and B, written in Larch-CORBA, satisfy that A can replace B [28]. Analogously, the second program 'Larch-CORBA.soft' is the one in charge of implementing the softmatchmaking process [29].

These programs have the same interface. They are both servlets that accept as arguments two pieces of text (i.e. two strings) with the two specifications to compare, and return a Boolean value with the result of the comparison: TRUE if the specification given in the first argument is a subtype of the second, FALSE otherwise.

We have distinguished between two kinds of matchings: exact ($\leqslant_E$) and soft ($\leqslant_S$). Exact matching is the usual kind: given two interfaces A and B, we shall say that $A \leqslant_E B$ if A can replace B, i.e. if A is a subtype of B using the common subtyping relations for interface signatures [19, 30]. This operator can be defined not only at the signature level, but also be naturally extended to deal with the semantic descriptions of components, following the usual subtyping relations for pre- and post-conditions [17, 31] or protocols [16, 18].

On the other hand, softmatchmaking between interfaces is defined for achieving partial matches. At the signature level we shall say that $A \leqslant_S B$ if interface B 'contains'

some of the services defined in A, and we shall also write $A \cap B \neq \emptyset$ (please note that $\leqslant_S$ is not a pre-order). This relation intuitively means that B offers at least one of the methods required by A. The extension of this operator to deal with semantic information is due to Mili *et al.* [29], who also define semantic distances between component specifications.

The last part, called 'choreography', deals with the semantic aspects of the component that globally describe its behavior, and that cannot be usually captured by the semantics of the individual interfaces, namely the relative order in which the component expects its methods to be called, the way it calls other components' methods, or how operations in separate interfaces interleave. In the example, the Larch description uses pre- and post-conditions to specify the behavior of the operations provided by the buffer. However, this information does not capture details as to when the required operation `print` is invoked, or the partial order in which the buffer operations should be called. This is commonly known as protocol information [7, 32], and can be expressed in many different notations such as Petri Nets, π-calculus, Message Sequence Charts etc. In this example, the actual description is given in-line and there are two external references to the programs that implement the exact and softmatchmaking checks [18]. The parameters of these two servlet programs are again two strings with the specifications to compare, and the result is a Boolean value indicating if the specification given in the first argument is a subtype of the second.

```
<choreography>
 <description notation="PI-CORBA">
   Empty(ref,printer) =
     ref?write(x,rep).printer!print(x).
       printer?().rep!().
                        Full(ref,printer,x) ;
   Full(ref,printer,x) =
     ref?read(rep).rep!(x).
       Empty(ref,printer) ;
 </description>
 <exactMatching href=".../servlet/
   PI-CORBA.exact" />
 <softMatching href=".../servlet/
   PI-CORBA.soft" />
</choreography>
```

The choreography for the buffer is expressed in a π-calculus-based notation [32]. As we can see, initially, the buffer only accepts a `write` operation, and then calls the `print` method of the `printer` component, waits for its response and replies to the `write` operation. After that, its behavior is defined by process `Full`, which specifies that the buffer can only accept a `read` operation in that state. A response is sent once this operation has been received, and the buffer changes its state to `Empty`. In [7] we can find more information on protocols and their importance for specifying components, beyond pre- and post-condition information. Further information on the notation used can be found in [32], and the subtyping mechanisms used in the matching programs are described in [18].

4.1.2. Extra-functional description

The second part of the `COTScomponent` template describes *extra-functional* aspects (e.g. QoS, 'ilities', 'nesses' etc.) in a similar way to ODP, i.e. by means of service properties [33].

We have studied the importance of extra-functional information and how to include it into our COTS documents [34]. We have adopted the ODP way of describing extra-functional properties, using properties, which is the usual way in which the extra-functional aspects of objects, services and components are expressed in the literature. We suggest using W3C types for describing properties, although any notation is valid for describing them (e.g. the OMG's CCM style [35, pp. 10–365], that also uses an XML vocabulary).

Thus, the `<properties>` tag will describe a collection of properties, each one indicated by a `<property>` tag, and with a type and a value associated (see below). Dynamic properties can also be implemented in this approach, indicating the reference to the external program that will evaluate their current values.

```
<properties notation="W3C">
 <property name="capacity">
  <type>xsd:int</type> <value>1</value>
 </property>
 <property name="isRunningNow">
  <type>xsd:boolean</type>
  <value href=".../running.cgi"/>
    <!--dynamic property-->
 </property>
 <property name="keywords" composition="AND">
  <property name="keyword">
   <type>xsd:string</type>
     <value>storage</value>
  </property>
  <property name="keyword">
   <type>xsd:string</type>
     <value>bounded</value>
  </property>
 </property>
</properties>
```

Please notice how we allow keyword-based searches too, including the special property 'keywords'.

In order to deal effectively with extra-functional requirements, we have used some principles from a qualitative approach called the NFR Framework [9]. This approach is based on the explicit representation and analysis of extra-functional requirements. Considering their complex nature, we cannot always say that extra-functional requirements can be entirely accomplished or satisfied. Rather, the NFR Framework represents extra-functional requirements as softgoals, which do not necessarily have *a priori*, clear-cut criteria of satisfaction.

In addition, extra-functional requirements can contribute positively or negatively, and fully or partially, toward achieving other extra-functional requirements. First, they are decomposed into more specific extra-functional requirements. For instance, the security requirement can

be considered to be quite broad and abstract. To explicitly deal with such a broad requirement, we may need to break it down into smaller parts, so that unambiguous solutions can be found. By treating this high-level requirement as a softgoal to be achieved, we can decompose it into more specific subgoals, which together satisfy the higher level softgoal (this is an AND type of contribution). For instance, the security softgoal can be decomposed into three sub-softgoals: integrity, confidentiality and availability. Another kind of composition is the OR type: the softgoal is satisfied if any of its subgoals is.

Thus, at the COTS component description level we can enrich the description of properties, which may be either single properties, or composition of properties. Composition can be either AND-composition or OR-composition.

```
<property name="authorization"
   composition="OR">
   <property name="userAuthorization">
     <type>xsd:string</type>
       <value>LOGIN</value>
   </property>
   <property name="managerAuthorization">
     <type>xsd:string</type>
       <value>ADMIN</value>
   </property>
</property>
```

We may need to express that a given property (no matter whether it is simple or composed) is 'implemented by' a given functional element (e.g. an interface). In NFR terms, this expresses that the functional element 'operationalizes' the property, i.e. this element is the one in charge of 'implementing' it. This permits traceability of requirements (in order to determine the functional elements which provide a given extra-functional requirement). Analogously, we can express that a given property is 'present' in a given functional element, i.e. this element 'exhibits' the property (by default, the whole component).

For instance, think of a property named 'userAuthorization' that reflects the requirement of having to authorize any component user before accessing any of its services. Among the many alternative ways of implementing it, imagine that the component provides an interface (ILogin) with operations for user 'login'. In this sense, the userAuthorization property is implemented by the ILogin interface, which can be expressed as a child XML element of the property:

```
  <property name="userAuthorization">
    <type>xsd:string</type>
    <value>LOGIN</value>
    <implementedBy>ILogin</implementedBy>
  </property>
```

Expressing that a given property is implemented in (i.e. supported by) a component or in a functional element may also be useful. For instance, the fact that the response time of

any of the operations provided by the IOnePlaceBuffer interface is less than 10 ms can be expressed as follows:

```
<property name="responseTime">
   <type>xsd:float</type><value>10</value>
   <implementedIn>IOnePlaceBuffer
   </implementedIn>
</property>
```

For a list of the quality properties for COTS components, the interested reader can consult [36].

4.1.3. Packaging/architectural constraints
This part contains the *packaging* information about how to download, deploy and install the COTS component that provides the required service, including implementation details, context and architectural constraints etc. Again, there is a `<description>` tag with the appropriate information written according to a particular notation, either implicitly encoded in the XML document, or pointed at by an `href` reference.

```
<packaging>
   <description notation="CCM-softpkg"
     href=".../MyImplementOfOnePlaceBuffer.
       csd" />
</packaging>
```

In this example, the CCM 'softpackage' [35] description style is used. A CORBA component package maintains one or more implementations of a component, and consists of one or more descriptors and a set of files. The descriptors describe the characteristics of the package—such as its contents or its dependencies—and point to its several files. This information allows description of the resources, configuration files, the location of different implementations of the component for different operating systems, the way those implementations are packaged, the resources and the external programs they need etc.

4.1.4. Marketing information
Finally, other non-technical details of the service are also described in this section. Typical information described here includes vendor information, licensing and commercial aspects, certificates, vendor support, level of customization allowed etc.

```
<marketing>
 <license href=".../license.html" />
 <expirydate>05-10-2003</expirydate>
 <certificate href=".../lcard.png" />
 <vendor>
   <companyname>E-Brokering corp.
   </companyname>
   <webpage>http://www.e-B.com/</webpage>
   <mailto>sales@e-Brokering.com</mailto>
   <address>
     <zip>04120</zip>
     <street>Ctra Sacramento s/n</street>
```

```
    <city>Almeria</city>
    <country>Spain</country>
  </address>
</vendor>
<description>A one-place buffer
  </description>
<ManMonthsRD>1</ManMonthsRD>
<ManMonthsSkillFactor>3
  </ManMonthsSkillFactor>
<LinesOfCode>53</LinesOfCode>
<Developer-CMM-level>4</Developer-CMM-level>
</marketing>
```

We have also included three tags that capture the information currently provided by ComponentSource (www.componentsource.com), one of the major software component vendors for the industry. These tags are the `ManMonthsRD`, `ManMonthsSkillFactor` and `LinesOfCode`. The Man Months Research and Development is the time taken to research, develop and test the current version of the product. The Man Months Skill Factor is the business skill-level needed to design and develop the product in the number of man months indicated in the `ManMonthsRD` field. Finally, Lines of Code is the number of lines of code that is in the current version of the product. Please note how these values try to measure the effort involved in developing the component, so the potential acquirer can decide whether it is worth buying the component, or if it is better to develop it from scratch—which is probably the most crucial decision. (An old rule of thumb in CBSD says that if you have to adapt more than 20% of the component functionality in order to integrate into your system, you had better develop it yourself.)

Another interesting information is the capacity and maturity of the development environment of the component developer, expressed in terms of the Capability Maturity Model (CMM). Finally, it is also worth noting the `<expirydate>` tag, which allows old service offers to be readily purged.

4.2. Importing services

Once we have described how to document component services, this section discusses how to import them, i.e. how a client may locate them using the COTStrader service.

In order to import a service, the client needs to provide two XML documents. The first one, called `COTSquery`, contains the selection criteria to be used by the trader to look for the service. The second document describes the main features of the required service.

```
<?xml version="1.0"?>
<COTSquery name="ClientQuery"
 xmlns="http://www.cotstrader.com/
   COTS-XMLSchema.xsd">

<COTSdescription href=".../Query1.xml" />
```

```
<functionalMatching>
  <interfaceMatching>
    <exactMatching href=".../
      exact2match.cgi" />
  </interfaceMatching>
  <choreographyMatching>
    <softMatching />
  </choreographyMatching>
</functionalMatching>

<propertyMatching>
  <constraints notation="XQuery">
    (responseTime <= 10) and
     (isRunningNow = TRUE)
  </constraints>
  <preferences notation="ODP">first
  </preferences>
</propertyMatching>

<packagingMatching notation="XQuery">
  description/notation = "CCM-softpkg" and
      (description/implementation/os/
       name="WinNT" or
      description/implementation/os/
       name="Solaris")
</packagingMatching>

<marketingMatching notation="XQuery">
  vendor/address/country = "Spain"
</marketingMatching>
</COTSquery>
```

The first section (`COTSdescription`) explains the target service by using the previously described `COTScomponent` template. The client fills in the required values (interfaces and properties) that will be used to compare against the information available in the service exports that the trader knows about. An example of such a description template is shown below (the one referenced as `Query1` in the `ClientQuery` template above).

```
<?xml version="1.0"?>
<COTScomponent name="Query1"
 xmlns="http://www.cotstrader.com/
   COTS-XMLSchema.xsd">
 <functional>
  <providedInterfaces>
   <interface name="onePlaceBuffer">
    <description notation="IDL-CORBA">
     interface IOnePlaceBuffer {
       void write(in long x); long read();
     };
    </description>
   </interface>
  </providedInterfaces>
 </functional>
 <properties notation="W3C">
  <property name="responseTime"
    priority="7">
```

```
    <type>xsd:float</type>
  </property>
  <property name="isRunningNow"
    priority="2">
    <type>xsd:boolean</type>
  </property>
 </properties>
</COTScomponent>
```

The last four parts of the `ClientQuery` XML document describe the selection criteria to be used. In the functional part, apart from the services description, the required kind of matching in each case can also be specified. The client may specify whether the matchmaking process should be exact or soft, and optionally the matchmaking program to be used (in that case, the program originally stated in the target COTS component description is ignored).

Property-based matching is done in the usual way for ODP traders, using constraints and preferences. Constraints are Boolean expressions consisting of service property values, constants, relational operators (`<`, `>=`, `=`, `!=`), logical operators (`not`, `and`, `or`) and parenthesis, that specify the matching criteria in order to include a component in the trader's list of candidates for the current search. Constraints are evaluated by the trader by substituting the property names with their actual values, and then evaluating the logical expression. Components whose constraints are evaluated to be false are discarded. We have used the notation defined in the W3C's XML QueryAlgebra proposal to write the matching expression (http://www.w3.org/TR/query-algebra/).

Another issue during the selection/matching process is resolving conflicts between contradictory properties. Assigning priorities is one of the possible solutions to deal with conflicts. Priorities can be assigned to each first-level property in the `<properties>` tag, using the scale 0 (very low) to 9 (very high), which is the scale commonly used in many decision-making processes. For instance, some security properties may be in conflict with some performance requirements (security checks may consume some time, which will have a negative impact on the response time of the component). Assigning priorities can help to decide the order between components in case none of them implement all required properties (e.g. a very fast but not secure component versus another component that implements security mechanisms but with a slower response time than required).

Preferences allow the sorting the list of candidates according to a given criterion using the terms `first`, `random`, `min(expr)` and `max(expr)`, where `expr` is a simple mathematical expression involving property names [19].

Finally, packaging and marketing information is matched using expressions that relate the values in the appropriate tags (`<packaging>` or `<marketing>`) of the `COTSdescription` query. In this example, the notation defined in the W3C's QueryAlgebra proposal is used again for building the 'select' expressions.

4.3. The trading process

COTStrader is an implementation of a trading service that provides a mapping between a client query (i.e. an import operation) and a set of COTS components that may act as valid service providers for that query (exporter candidates). We have already discussed the notation for exporting and importing services, by means of XML documents. In this section, we will discuss how the trading process works, and how potential clients can make use of it.

Our implementation of the **COTStrader** consists of a CORBA object, which has two main interfaces, `Register` and `Lookup`, similar to the ones supported by the ODP trader.

```
module COTStrader {
 interface Register {
   boolean export  (in  string
                       XMLCOTSComponent,
                    in  string userID,
                    out string  results );
   boolean withdraw(in  string
                       XMLCOTSComponent,
                    in  string userID,
                    out string results );
   boolean replace (in  string
                      oldXMLCOTSComponent,
                    in  string olduserID,
                    in  string
                      newXMLCOTSComponent,
                    in  string newuserID,
                    out string results );
 };
 interface Lookup {
   boolean query  (in  string
                      XMLCOTSqueryTemplate,
                   in  long
                     MaxCandidates,
                   in  boolean
                     StoreAndForward,
                   out long    nHits,
                   out string  templates,
                   out string  results);
 };
};
```

The three methods of interface `Register` allow registration of a `COTScomponent` document in the trader, repository, its removal and updating of this information respectively. All methods return `TRUE` if the operation succeeds, `FALSE` if it fails. The `results` parameter contains a description of the failure in the latter case.

Interface `Lookup` has only the `query` operation, used to look for components. In addition to the `COTSquery` template with the selection criteria, the user may specify the maximum number of candidates to be returned, and whether a 'store-and-forward' policy is followed. This operation returns the number of services found and a string with the

sequence of `COTScomponent` templates, one for each component found.

COTS vendors may export their services to the **COTStrader** using either a push or a pull model. In the first case, the exporter directly accesses the trader and registers the component information using the operations listed above. Apart from this CORBA interface, we have built a portal to access the **COTStrader** services using Web forms.

In the pull model, the exporter only has to leave the XML document with the component template in a place accessible to Web search engines. Together with the basic trader we have also developed another component called the **ServiceFetcher**, which uses automated searchers (bots) and well-known search engines to locate Web pages that contain XML descriptions of services, i.e. `COTScomponent` templates. Once found, they are automatically registered at the **COTStrader**.

Our current implementation of **COTStrader** just keeps a database with the XML registered templates, which serves as its repository. In this sense, it is just a prototype implementation to validate the feasibility of our proposal.

Once a query is received, the trader looks in the repository trying to match existing component descriptions with the required one. The algorithm currently followed by the **COTStrader** to match whether a COTSquery template Q matches a template T from the repository (hence including T in the list of candidates for that query) performs the following steps:

(i) If a `<marketingMatching>` tag is present in Q, the QueryAlgebra expression is evaluated for the fields of the corresponding tag of T referenced in the expression. T is discarded if the expression is evaluated to be false. It may happen that any field in the expression is present in T; in this case the expression evaluates to `TRUE`.

(ii) The same happens for the `<packaging-Matching>` tag.

(iii) The same happens for the `<propertyMatching>` tag (which includes not only quality properties, but also keywords etc). In this case, the dynamic properties should be evaluated prior to evaluating the QueryAlgebra expression. The criterion described in the `<preferences>` tag is used to insert T into the list of candidates, in case it is finally selected.

(iv) Functional information is then matched using the 'matching' programs. Signatures (provided and required interfaces, in this order) are matched first, then events if they are indicated, then choreographies and finally behaviors. This follows the usual least-cost pattern [37], which first filters those functional elements that are easier to test.

In the case of functional elements, the templates might specify the matching programs for every particular element: interface, behavior or protocol (events are only syntactically matched).

(i) In case a matching program m is specified in Q for a particular element, and the notations in which the elements are described in Q and T are compatible, the program m is used for the element.

(ii) If there is no such matching program specified in Q, but there is one in T, it is used.

(iii) Otherwise, if there is a default matching program in the **COTStrader** that can handle the notation in which the elements to be matched are described, it is used.

(iv) Otherwise, the component is marked as 'potential' candidate, since no checks are possible for that particular element.

After repeating this process for all elements in Q, template T is: (a) discarded if any of the tests has failed, (b) included in the list of candidates if all tests have succeeded or (c) considered as 'potential' candidate if all tests have either succeeded or there was no matching program for a particular functional element. The manner of dealing with 'potential' candidates will depend on the sort of matching selected by the client (soft or exact).

(i) In the case of soft matching, T is included in the list of candidates, after those which passed all the tests.

(ii) In the case of exact matching, our algorithm compares only the signature information, checking that:

    (a) all provided interfaces and consumed events defined in Q should be present in T, and they need to be syntactically equal;

    (b) the sets of required interfaces and emitted events in Q should be a superset of those in T, using a syntactic match.

5. BUILDING COTS-BASED SYSTEMS USING THE TRADER

As mentioned in the Introduction section, CBSD challenges some of the current Software Engineering methods and tools. For instance, the traditional top-down or bottom-up development methods are not directly transferable to CBSD. Here, the system designer has also to take into account the specification of pre-developed COTS components available in software repositories, incorporating them into all phases of the development process [1, 2, 3].

Current solutions addressing these issues are usually based on spiral methodologies (see e.g. [12]), which progressively produce more detailed requirements, architectural specifications and system designs by repeated iterations.

A good methodology for CBSD that follows this approach is due to Cheesman and Daniels [38]. After the requirement analysis phase, an abstract and preliminary software architecture of the system is defined from the user's requirements, which defines its high-level structure, exposing its organization as a collection of interacting components. These are called abstract components that are then matched against the list of concrete COTS components available in software repositories. This is the so-called gap analysis problem for which our trader tries to provide a solution. This trading process produces a list of the candidate components that could form part of the application: both because they provide some of the required services, and because they may
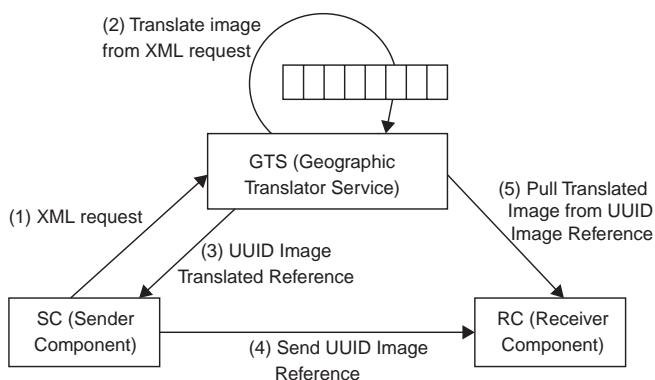
FIGURE 1. A schematic view of the GTS example.

fulfil some of the user's (extra-functional) requirements such as price, security limitations etc.

With this list, the system architecture is re-examined in order to accommodate as many candidates from the list as possible. There are usually different ways to combine the candidate components to build the system. Such different combinations (that we will call configurations) need to be generated and then shown to the system designer for a decision to be made: which configuration is the one that best matches the user requirements, which components are still missing and hence need to be developed and how much the initial software architecture should be changed (and whether it is worth changing) in order to accommodate the COTS components found.

Then, the system requirements are matched by the software architect against those provided by the obtained architecture, and revised if needed. The process starts again until a software architecture that meets the user requirements and is 'implementable' from COTS components is obtained [38].

As we can see, the software architecture is refined (re-adjusted) at each step of the iteration. At the initial stages, instead of specifying abstract component interfaces *a priori*, they are obtained after analyzing the software components offered in the component marketplace. The first trading for components is made basically looking for the main required 'features' of the components, using just selected keywords, and 'soft' matchmaking. As the architecture gets more and more refined, the searches are based on more 'exact' matches. In this way, the architecture of the system is always built by taking into account the third-party components available in software repositories, using a bottom-up approach.

In this section we will see an example of how it is possible to integrate the COTStrader into this kind of spiral process, automating most of the search and selection activities.

5.1. An example application

In order to illustrate our proposal we will use another example, extracted from a large application we developed within an industrial project, in which COTS components had to be used as much as possible. The example application comes from the distributed geographic information systems (GIS) arena, and consists of a common service to convert spatial images, usually known as a Geographic translator service (GTS). Briefly, a Sender component needs to send a formatted image to a Receiver component, but instead of sending it directly, it uses a translator service for dealing with all the issues related to image format conversion and compression. This simplifies both the Sender and the Receiver, taking away from them all those format-compatibility issues.

The way the service works is shown in Figure 1. First, the Sender forwards a request to the GTS with the required service and its related information:

```
<image url="http://.../download/">
 <name input="RiverImage"
      output="RiverImage"/>
 <format input="DWG" output="DXF"/>
 <compression input=".zip" output=".tar"/>
</image>
```

Then, the GTS downloads a zip-compressed DWG image from the `http` site, generates a DXF file with the same name, stores it in a buffer, associates a unique (Universal Unique IDentifier UUID) to it and returns the UUID to the Sender to extract the converted file from the GTS buffer.

The following sections briefly describe how the example system was built. The interested reader can consult [39] for a more detailed description of this example.

5.2. Describing software architectures

Complex software systems and applications require expressive notations to represent their architectures. Traditionally, specialized Architecture Description Languages (ADLs) have been used, allowing the formal description of the structure and behavior of the architecture of the application being represented [40]. However, the formality and lack of visual support of most ADLs have encouraged the quest for more user-friendly notations.

One of the proposals that makes use of UML to represent software architectures is UML-RT [41], which originally defined UML extensions for modeling real-time systems, but that has also been successfully used in wider environments, in particular to describe the software architecture of financial and banking systems [42]. It is also supported by commercial tools (e.g. Rational Rose RealTime).

Figure 2 shows the software architecture of the example application drawn using UML-RT. Components are represented by means of UML-RT capsules. Users' requirements and other component properties are documented by means of UML notes and tagged values (not shown for simplicity in the drawing).

This subsystem contains a main component, the `Translator`, which uses the services of four other components: a file compressor, a geographical image converter, a buffer for storing the image files, and a component for intermediate representation and manipulation of the XML data. Components `XDR` and `XMLBuffer` use another component (`DOM`) to deal with XML documents using the
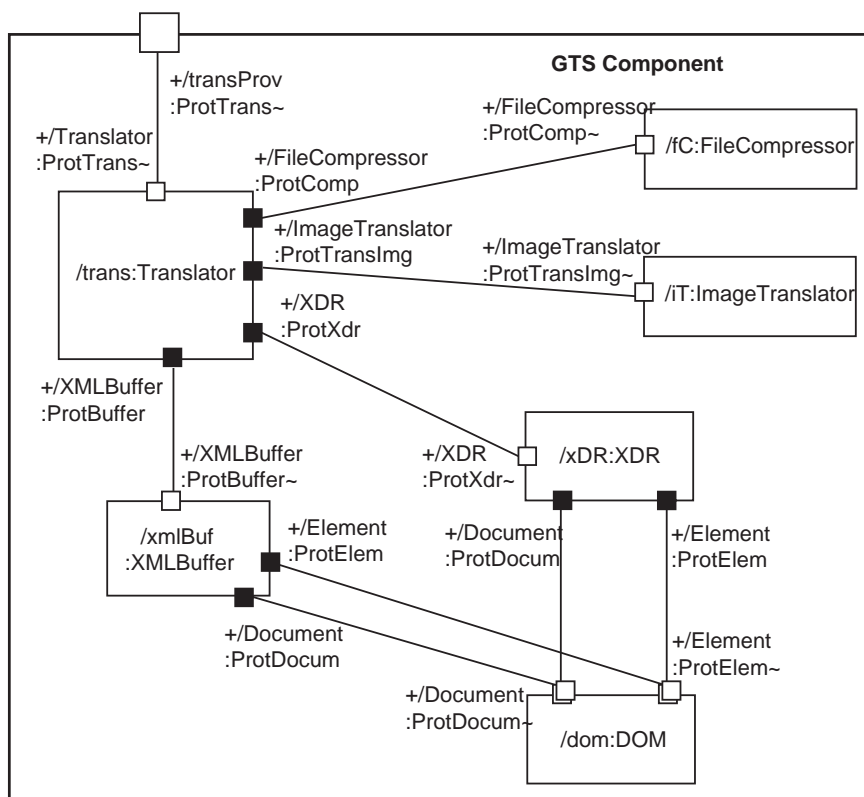
FIGURE 2. The GTS software architecture in UML-RT.

DOM model. The interface of the DOM component is available in commercial software packages such as IBM XML4J or Sun JAXP.

5.3. Extracting component information

Once the software architecture of the application is drawn, the information about the components, the services they offer and require, and their properties must be extracted from the UML-RT diagram. For that purpose we use a process that parses the (RTMDL) files produced by Rational Rose RealTime, and produces a list of XML COTScomponent templates with the description of the components found in that architecture. Moreover, we have a generic tool that processes XML Metadata Interchange (XMI) files (www.omg.org) and produces the COTScomponent templates, since we did not want to commit to any particular tool or graphical notation.

5.4. Invoking the trader services

Once we have a list of COTScomponent templates describing the services that the components in our system should have, the next step is to invoke the trader. For each of those templates describing the abstract components, the trader produces a list of candidate components, which are available to implement the system. This process was described in detail in Section 4.

As we mentioned earlier, the matching operations start with 'soft' matches (basically, by looking for keywords only) and get more and more 'exact' in each iteration, as the

software architecture gets progressively refined. Typical (increasingly stronger) levels of matching are: keywords, marketing and packaging information (operating systems, component models etc.), quality properties, interface names, interface operations and behavioral and semantic information. Although the latter matchings are in theory very useful, our experience shows that it is difficult to go beyond the level of looking for quality properties. Software vendors do not even include the names of the interfaces that provide their services, not to mention their semantics [10].

5.5. Building 'configurations'

Traditionally, the search and matching processes of components have been defined on a one-to-one basis [30, 31, 37], whereby each component implements just one service, and requires none. However, this is not the common case in most real applications; in general, COTS components are coarse-grained components that integrate several services and offer several interfaces. Think for instance of an Internet navigator or a word processor: apart from their core services they also offer many other services, such as Web page composition, spell check etc.

The trader can help to find and locate those components that implement any of the services specified in the system architecture. In this step, we face the problem of defining those 'combinations' of the components found by the trader that may implement the system (or parts of it). Of course, there are usually many different ways of combining the components found, since more than one component may

offer the same service (as may usually happen in the case of complex components providing several services). Let us call configurations to every alternative combination that can be built with the components found by the trader [15].

Not all configurations are 'valid' for building the system. The goal is to find those configurations with no service gaps and no service overlaps. Gaps happen when any component in a configuration provides one of the services required by the architecture. On the contrary, overlaps happen when two or more components in the same configuration provide the same service.

The task of building valid configurations from the set of candidate components found by the trader is not an easy task, especially when components may offer and require more than one service simultaneously. The idea is to start exploring all the possible alternatives, and discard those with gaps or overlaps.

A backtracking algorithm to build valid configurations has been reported in a separate work [15], together with a more rigorous definition of the concepts 'valid configuration', 'closure' etc. The implementation of the algorithm is now part of the tool suite that accompanies the trader, and can be obtained at the COTStrader Web site. In [15] we also discuss the process of ordering the valid configurations according to some criteria, and how they are presented to the software architect, for him/her to decide which is the most suitable for his/her system.

In our example, we had a COTS component repository built from the information provided by different software component vendors. The trader found eight components which provided one or more of the seven services required by the components specified in the architecture. Out of the $2^8 = 256$ possible combinations, 24 were valid configurations of which only five were closed. ([39] contains a complete description of this example.)

It is now up to the system architect to decide, based on the list of configurations produced, how to proceed. For instance, the system designer may decide to use one of the configurations obtained, it being the best one that matches his/her requirements. But he/she may also decide to review the initial architecture in order to accommodate it to the components found if any configuration really satisfies the requirements. With this new architecture, he/she may start the process again.

6. RESULTS AND DISCUSSION

In this section we will discuss some of the issues that currently have an impact on the applicability of our proposal in commercial environments. The first issue is about the documentation of components. There are several proposals for documenting them, using different notations and strategies [3]. Most of the approaches agree on the basic information that needs to be captured in order to build component-based systems (e.g. [10, 43, 44, 45]. However, few of the proposals are supported by tools, and probably none is widely accepted by the industry for documenting commercial software components.

In order to validate our proposal we tried to build the GIS example application described in Section 5.1 using COTS components. First of all, we surveyed the Web sites of different software component providers (such as IBM, Sun, ComponentSource, Flashline and OpenSource RedHat Community), trying to fill-in our component templates with the information available about the components they sell or license. The study was conducted during the first half of 2002, and the repository built with these samples can be found at http://www.cotstrader.com/samples/templates/repository.

The results clearly show the gap between what is claimed by the research community about the information needed for component reuse (especially if we want to have automated support), and the (scarce) information provided by software component vendors. More precisely, we found that only 25–40% of the information described in our templates was available from the vendors.

- `<marketing>` Most of this information is available from most vendors.
- `<packaging>` Basically, only deployment characteristics are available: CPUs, operating systems etc.
- `<properties>` Extra-functional information was difficult to find, apart from some of the supported features, and some very specific characteristics. For instance, in the case of the spatial image conversion components (such as BBN's OpenMap or ESRI's MapObject) there was information available about the supported conversions (DXF, DWG, MIF, …), map projections (Orthographic, Polyconic, Azimuthal, …) or the coordinate systems handled (UTM, GKM, ECEF). But very little extra-functional information was found about the quality of service provided, for instance.
- `<functional>` Apparently, the most 'technical' and 'easy-to-provide' information. Most academic proposals for effective CBSD provide successful solutions based on this kind of information (component interfaces and behavioral semantics), without which (automated) CBSD seems unaffordable. However, that was surprisingly the most difficult information to be found. The very few vendors that provided some functional information described just (some of) the names of the supported interfaces, but nothing about their operations, or their protocols or semantics.

These results evidence the need of better component documentation, if effective component search and selection processes are to be achieved. Our evidence shows that we are currently far from this, since we cannot even count with the functional information required for checking whether a given COTS component fits into a software architecture described by the interfaces and method names of its constituent components.

Finally, another issue worth mentioning is that trading can be particularly effective in the case of large companies working in very specific environments, and/or using product-line architectures [46] that rely on in-house repositories of components previously developed for similar applications.

7. RELATED WORK

Two main research lines can be related to ours: the design of software component traders, and the documentation of COTS components so that they can be fully integrated into Software Engineering practices and methodologies.

Most of the existing traders (e.g. [20, 21, 22, 23, 24, 25, 26]) follow the ODP model [19] (which is also the one adopted by the OMG), and therefore have similar features, advantages and disadvantages, as we have already discussed in Section 2.2. An approach that somehow distinguishes it from the rest, and which claims that an enhanced model for trading is needed for Internet-based software development is called WebTrader [11]. This proposal uses XML to describe component services, but the problem is that the information it manages about components is somehow limited (actually, it is very close to the information handled by the standard ODP trader).

A proposal very similar to ours is called Component-Xchange [44], which also uses an XML-based specification language for components, and implements a component broker over the Internet. However, it is quite focused on e-commerce, forcing components to be license-aware and imposing the trader to be involved in the transaction. Furthermore, it does not deal with any behavioral description of the semantics of the components and has limited support for some of the features required for COTStrader, such as implementing a pull-based model and store-and-forward requests.

Agora is an interesting search engine for components developed at the SEI, that provides agents that crawl the Web for components [45]. The components descriptions are indexed and stored in the search engine. However, Agora deals only with the syntactic aspects of components, without considering the rest of their aspects (packaging, marketing, properties etc.).

There is also a directory service for WebServices, called (Universal Description, Discovery, and Integration UDDI, http://www.uddi.com). The companies that are willing to advertise a WebService they provide, must register at a (UDDI Business Registry UBR), which is the repository used by UDDI to locate WebServices. A UBR repository contains data about the registered companies structured in white, yellow and green pages. White pages contain information about the company providing the service. Yellow pages group companies according to the kind of services they provide. Finally, green pages contain the references to the WebServices Definition Language (WSDL) description of the services provided by a company, i.e. the technical description of the offered WebServices.

Apart from using these technical descriptions, searches are focused on aspects of location, binding and communication of WebServices. Precisely, one of the limitations of UDDI comes from the fact that these technical descriptions rest on WSDL, which only allows the capture of functional information about the services being described, and that too only at the signature level (it does not allow any behavioral or protocol descriptions). Furthermore, extra-functional information (e.g. quality-related attributes) about

services cannot be captured with WSDL either, hindering the automated assessment and selection of WebServices based on extra-functional requirements and architectural constraints. In this sense, UDDI provides a very useful and complete directory service, but it cannot really be regarded as a trader: a trading service can be considered as an advanced directory service which allows attribute-based search [47]. Besides, as a directory service, UDDI does not currently implement two features common to traders: federation and query propagation. Although the UDDI specification allows the concept of 'affiliation', it does not really contemplate the possibility of federation of UDDI repositories. Query propagation between different UBRs is not allowed either.

Regarding the documentation of components, this is currently a hot topic, and many authors are making different proposals [48]. To mention some of them, IBM is working on a proposal to document their large-grained components (www.ibm.com/software/components), and there are several good proposals from SEI (www.sei.cmu.edu) claiming better component documentation. Han [43] has defined some component specification templates in a joint project with Fujitsu, Australia, which provide semantic information for proper usage and selection of components, in addition to their standard signature description. Bastide *et al*. [49] and Canal *et al*. [32] are among the authors that propose IDL extensions in order to deal with protocol information using Petri nets and π-calculus respectively. Finally, Alves *et al*. [8] are working on the extra-functional side of components, trying to add this kind of information to commercial components, with the goal of relating it to the architectural specification of the applications, too. Our proposal for documenting components just tries to provide a common template to integrate most of these different notations, aiming at being flexible and versatile enough to host all of them.

8. CONCLUSIONS AND FUTURE WORK

CBSD aims at building software systems by searching, selecting, adapting and integrating (COTS) software components. In a world where the complexity of the applications is continuously growing, and the amount of available information is becoming too large to be handled by human intermediaries, automated trading processes need to play a critical role.

In this paper we have analyzed the features that COTS components traders should have in open systems, and presented COTStrader, an Internet-based trader for COTS components that addresses the heterogeneity, scalability and evolution of COTS markets. Furthermore, we have shown how it can be integrated into some kinds of spiral CBSD methodologies, providing partially automated support for COTS components search and selection processes.

There are several possible extensions to our work. In the first place, the COTStrader fulfils some of the requirements described in Section 2.1, but does not implement all of them. Some issues remain open, such as the dynamic service composition and adaptation, the definition and usage

of heuristic functions or the query delegation. 'Semantic' trading is not covered by our proposal either, since we do not currently deal with concepts, ontologies or knowledge-based trading.

We also need more tools to automate the matchmaking operators at the protocol and semantic levels. Compatibility and replaceability of services are two of the key issues in CBSD, together with predictable assembly, and therefore the automated support for them seems to be crucial.

Moreover, the emergence of Semantic Web cannot be ignored. Part of our current efforts is focused on enhancing the COTStrader to deal with the new Web advances, integrating our proposal with their notations (WSDL, WSFL, RDF), resources and repositories, so they can be successfully handled by the COTStrader.

Likewise, we are building bridges to other existing traders (especially to CORBA ones). The idea is to be able to connect COTStrader with other trading services tied to particular component technologies, aiming at the provision of a more complete and global COTS trading service.

ACKNOWLEDGEMENTS

REFERENCES

[1] Mili, H., Mili, F. and Mili, A. (1995) Reusing software: issues and research directions. *IEEE Trans. Softw. Eng.*, **21**, 528–562.

[2] Robertson, S. and Robertson, J. (1999) *Mastering the Requirement Process*. Addison-Wesley and ACM Press.

[3] Wallnau, K. C., Hissam, S. A. and Seacord, R. C. (2002) *Building Systems from Commercial Components*. Addison-Wesley, Boston.

[4] Tran, V. and Liu, T. (1997) A procurement-centric model for engineering component-based software systems. In *Proc. Fifth Int. Symp. on Assessment of Software Tools*, Pittsburgh, PA, USA, June 3–5, pp. 70–79. IEEE Computer Society Press.

[5] Garlan, D., Allen, R. and Ockerbloom, J. (1995) Architectural mismatch: why reuse is so hard. *IEEE Softw.*, **12**, 17–26.

[6] Ncube, C. and Maiden, N. (2000) COTS software selection: the need to make tradeoffs between system requirements, architectures and COTS components. In *COTS Workshop: Continuing Collaborations for Successful COTS Development*, Limerick, Ireland, June 4–5. ACM Press.

[7] Vallecillo, A., Hernández, J. and Troya, J. M. (2000) New issues in object interoperability. In *Object-Oriented Technology: ECOOP 2000 Workshop Reader*, Lecture Notes in Computer Science, **1964**, Sophia Antipolis and Cannes, France, June 12–16, pp. 256–269. Springer-Verlag.

[8] Alves, C. F., Rosa, N. S., Cunha, P. R. F., Castro, J. F. B. and Justo, G. R. R. (2001) Using non-functional requirements to select components: a formal approach. In *Proc. Fourth Iberoamerican Workshop on Requirements Engineering and Software Environments (IDEAS'01)*, San José, Costa Rica, April 3–6. Instituto Technológico de Costa Rica.

[9] Chung, L., Nixon, B. A., Yu, E. and Mylopoulos, J. (1999) *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers.

[10] Bertoa, M. F., Troya, J. M. and Vallecillo, V. (2003) A survey on the quality information provided by software component vendors. In *Proc. 7th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2003)*, Darmstadt, Germany, July 21–25, pp. 25–30. Universidade Nova de Lisboa.

[11] Vasudevan, V. and Bannon, T. (1999) Webtrader: discovery and programmed access to web-based services. *Poster at the 8th Int. WWW Conf. (WWW8)*, Toronto, Canada, May 11–14. University of Toronto.

[12] Nuseibeh, B. (2001) Weaving together requirements and architectures. *IEEE Comp.*, **34**, 115–117.

[13] Szyperski, C. (1998) *Component Software. Beyond Object-Oriented Programming*. Addison-Wesley Professional.

[14] Brown, A. W. and Wallnau, K. (1999) The current state of CBSE. *IEEE Softw.*, **15**, 37–46.

[15] Iribarne, L., Troya, J. M. and Vallecillo, A. (2002) Selecting software components with multiple interfaces. In *Proc. 28th Euromicro Conf.*, Dortmund, Germany, September 4–6, pp. 26–32. IEEE Computer Society Press.

[16] Nierstrasz, O. (1995) Regular types for active objects. In Nierstrasz, O., and Tsichritzis, D. (eds), *Object-Oriented Software Composition*. Prentice-Hall, pp. 99–121.

[17] Leavens, G. T. and Sitaraman, M. (2000) *Foundations of Component-Based Systems*. Cambridge University Press.

[18] Canal, C., Pimentel, E. and Troya, J. M. (2001) Compatibility and inheritance in software architectures. *Sci. Comput. Program.*, **41**, 105–138.

[19] ISO/IEC 13235, ITU-T X.9tr (1996) *Information Technology—Open Distributed Processing—ODP Trading Function*. International Organization for Standarization and International Telecommunication Union.

[20] IONA (2000) *ORBacus trader. ORBacus for C++ and Java*. Object Oriented Concepts, Inc., IONA. http://www.ooc.com/ob.

[21] PrismTech (2001) *Trading Service—White Paper*. PrismTech OpenFusion, Enterprise Integration Services. http://www.prismtechnologies.com.

[22] Shmidt, D. C. (2001) *AceORB (TAO). The Adaptative Communication Environment*. Department of Computer Science and Engineering, Washington University, USA. http://www.cs.wustl.edu/~schmidt/TAO.html.

[23] Bearman, M. (1997) *Tutorial on ODP Trading Function*. Faculty of Information Sciences Engineering, University of Canberra, Australia.

[24] Beitz, A. and Bearman, M. (1995) An ODP trading service for DCE. In *Proc. First Int. Workshop on Services in Distributed and Networked Environments (SDNE)*, Prague, Czech Republic, June 27–28, pp. 42-49. IEEE Computer Society Press.

[25] Kutvonen, L. (1996) Overview of the DRYAD trading system implementation. In *IFIP/IEEE Int. Conf. on Distributed Platforms*, New York, USA, May 21–23, pp. 314–326. Chapman and Hall.

[26] Merz, M., Muller, K. and Lamersdorf, W. (1994) Service Trading and Mediation in Distributed Computing Systems. In *Proc. 14th Int. Conf. on Distributed Computing Systems*, Poznan, Poland, June 21–24, pp. 450–457. IEEE Computer Society Press.

[27] OMG ad/2001-08-19 (2001) *A UML Profile for Enterprise Distributed Object Computing V1.0.* Object Management Group.

[28] Dhara, K. K. and Leavens, G. T. (1996) Forcing behavioral subtyping through specification inheritance. In *Proc. 18th Int. Conf. on Software Engineering (ICSE-18)*, Berlin, Germany, March 23–30, pp. 258–267. IEEE Computer Society Press.

[29] Mili, R., Desharnais, J., Frappier, M. and Mili, A. (2000) Semantic distance between specifications. *Theoret. Comp. Sci.*, **247**, 257–276.

[30] Zaremski, A. M. and Wing, J. M. (1995) Signature matching: a tool for using software libraries. *ACM Trans. Softw. Eng. Methodol.*, **4**, 146–170.

[31] Zaremski, A. M. and Wing, J. M. (1997) Specification matching of software components. *ACM Trans. Softw. Eng. Methodol.*, **6**, 333–369.

[32] Canal, C., Fuentes, L., Pimentel, E., Troya, J. M. and Vallecillo, A. (2003) Adding roles to CORBA objects. *IEEE Trans. Softw. Eng.*, **29**, 242–260.

[33] ISO/IEC 10746:1-4, ITU-T X.901-904 (1997) *RM-ODP. Reference Model for Open Distributed Processing*. International Organization for Standardization and International Telecommunication Union.

[34] Iribarne, L., Vallecillo, A., Alves, C. and Castro, J. (2001) A non-functional approach for COTS components trading. In *Proc. Fourth Workshop on Requirements Engineering (WER'01)*, Buenos Aires, Argentina, November 22–23, pp. 124–138. Universidad Technológica Nacional de Buenos Aires.

[35] OMG TC/2002-06-65 (2002) *The CORBA Component Model*. Object Management Group.

[36] Bertoa, M. F. and Vallecillo, A. (2002) Quality attributes for COTS components. *I+D Computación*, **1**, 128–144.

[37] Goguen, J. A., Nguyen, D., Meseguer, J., Luqi, Zhang, D. and Berzins, V. (1996) Software component search. *J. Syst. Integration*, **6**, 93–134.

[38] Cheesman, J. and Daniels, J. (2001) *UML Components. A Simple Process for Specifying Component-based Software*. Addison-Wesley.

[39] Iribarne, L., Troya, J. M. and Vallecillo, A. Trading for COTS components to fulfil architectural requirements. In Lycett, M. G., de Cesare, S. and Macredie, R. D. (eds) *Development of Component-Based Information Systems*. M. E. Sharpe, Inc., to appear.

[40] Medvidovic, N. and Taylor, R. N. (2000) A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, **26**, 70–93.

[41] Selic, B. and Rumbaugh, J. (1998) Using UML for modeling complex real-time systems. Available at http://www.rational.com/media/whitepapers/umlrt.pdf.

[42] Bastos, L. and Castro, J. (2002) An event based layered architecture for bank system. In *Proc. Fifth Iberoamerican Conf. on Requirements Engineering and Software Environments (IDEAS'02)*, La Havana, Cuba, April 23–26, pp. 138–148. University of La Havana.

[43] Han, J. (2000) Temporal logic based specifications of component interaction protocols. In Vallecillo, A., Hernández, J. and Troya, J. M. (eds) *Proc. ECOOP 2000 Workshop on Object Interoperability (WOI'00)*, Lecture Notes in Computer Science, **1850**, Sophia Antipolis and Cannes, France, June 12–16, pp. 43–52. Springer-Verlag.

[44] Varadarajan, S., Kumar, A., Gupta, D. and Jalote, P. (2002) ComponentXchange: an e-exchange for software components. In *Proc. IADIS Int. Conf. WWW/Internet 2002*, Lisbon, Portugal, November 13–15, pp. 62–72. IADIS Press.

[45] Seacord, R. C., Hissam, S. A. and Wallnau, K. C. (1998) Agora: a search engine for software components. *IEEE Internet Comput.*, **2**, 62–70.

[46] Bosch, J. (2000) *Design & Use of Software Architectures*. Addison-Wesley.

[47] Kutvonen, L. (1995) Achieving interoperability through ODP trading function. In *Second Int. Symp. on Autonomous Decentralized Systems (ISADS'95)*, Arizona, USA, April 25–27, pp. 63–69. IEEE Computer Society Press.

[48] Lüders, F., Lau, K.-K. and Ho, S. (2002) Specification of software components. In Crnkovic, I. and Larsson, M. (eds) *Building Reliable Component-based Systems*. Artech House, London, pp. 52–69.

[49] Bastide, R., Sy, O. and Palanque, P. (1999) Formal specification and prototyping of CORBA systems. In *Proc. ECOOP'99*, Lecture Notes in Computer Science, **1628**, Lisbon, Portugal, June 14–18, pp. 474–494. Springer-Verlag.