# DEVELOPMENT OF COMPONENT-BASED INFORMATION SYSTEMS

SERGIO DE CESARE
MARK LYCETT
ROBERT D. MACREDIE
EDITORS

AMIS

ADVANCES IN MANAGEMENT INFORMATION SYSTEMS
VLADIMIR ZWASS SERIES EDITOR

# DEVELOPMENT OF COMPONENT-BASED INFORMATION SYSTEMS

SERGIO DE CESARE
MARK LYCETT
ROBERT D. MACREDIE
EDITORS

AMIS

ADVANCES IN MANAGEMENT
INFORMATION SYSTEMS
VLADIMIR ZWASS SERIES EDITOR

M.E.Sharpe
Armonk, New York
London, England

**Library of Congress Cataloging-in-Publication Data**

References to the AMIS papers should be as follows:

Janssen, M., and Wagenaar, R. Business engineering of component-based systems. S. de Cesare, M. Lycett, and R.D. Macredie, eds., *Development of Component-Based Information Systems: Advances in Management Information Systems,* Volume 2 (Armonk, NY: M.E. Sharpe, 2005), 166–186.

ISBN 0-7656-1248-8
ISSN 1554-6152

Printed in the United States of America

The paper used in this publication meets the minimum requirements of American National Standard for Information Sciences Permanence of Paper for Printed Library Materials, ANSI Z 39.48-1984.

∞

BM (c) 10 9 8 7 6 5 4 3 2 1

Chapter 11

# TRADING FOR COTS COMPONENTS TO FULFILL ARCHITECTURAL REQUIREMENTS

Luis Iribarne, José María Troya, and Antonio Vallecillo

***Abstract:*** *Component-based development (CBD) moves organizations from application development to application assembly, involving the use of third-party, prefabricated pieces (commercial off-the-shelf components, COTS) and spiral development methodologies. Although a software component market is still quite slow to develop, effective use of software components is slowly becoming a valid technology for the building of software systems. Moreover, the complexity of the applications is continuously growing, and the amount of the information about components is becoming too large to be handled by human intermediaries. Therefore, automated trading of components will play a critical role in CBD. This chapter underlines the need of linking three areas of the COTS CBD: the documentation and specification of COTS components, the description of COTS-based software architectures, and the trading processes for COTS components. A trading-based development method (TBDM), a three-tier method to build software applications as an assembly of COTS software components, is presented. A sample implementation is illustrated.*

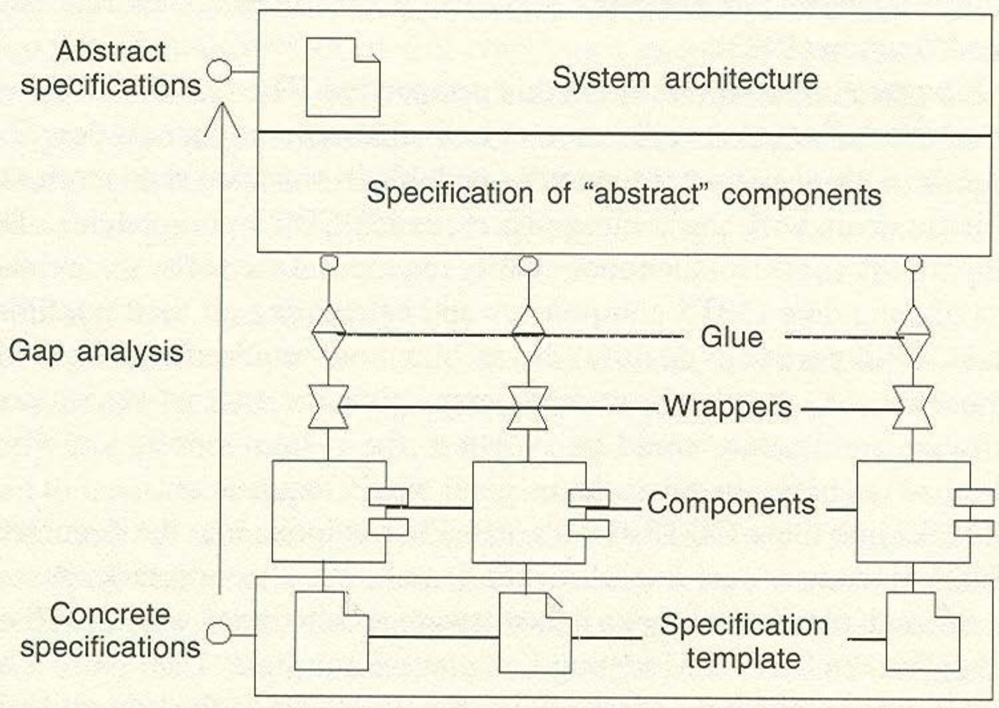
***Keywords:*** *Component Trading, Automated Trading, Trading-Based Development Method, Commercial Off-the-Shelf Components, Architecture*

## INTRODUCTION

In the last decade, component-based development (CBD) has produced a great 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. Being currently more a purpose to achieve than a reality, this approach 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 final goal is to be able to reduce development times, costs, and effort, while improving the flexibility, reliability, and reusability of the final application due to the reuse of software components already tested and validated.

This approach is challenging some of the current software engineering methods and tools. For instance, the traditional top-down development method is not transferable to component-based development. This method is based on successive refinements of the system requirements until a suitable concrete component implementation of the final application is reached. In CBD, the system designer has also to take into account the specification of predeveloped COTS components available in software repositories, which must be considered even when building the initial requirements of the system, incorporating them into all phases of the development process (Mili et al. 1995; Robertson and Robertson 1999). Here, system architects, designers, and builders

Figure 11.1 **Fulfilling Components' Abstract Specifications From Concrete Specifications**



must accept the trade-offs among three main concerns: user's requirements, system architecture, and COTS products (Garlan et al. 2002; Ncube and Maiden 2000).

Current solutions are usually based on spiral methodologies—see, for example, Nuseibeh 2001— which progressively develop detailed requirements, architectural specifications, and system designs, by repeated iterations. These solutions are also related to the so-called gap analysis (Cheesman and Daniels 2001), shown in Figure 11.1. Here, the abstract software architecture of the system is first defined from the user's requirements, which describe the specification of "abstract components" and their relationships. These abstract components are then matched against those "concrete components" available in software repositories. The matching process produces a list of the candidate components that could take part in the application: both because they provide some of the required services, and because they may fulfill some of the user's (extrafunctional) requirements such as price and security limitations. With this list, the system architecture is reexamined in order to accommodate as many candidates from the list as possible. Then the system requirements are matched against those provided by the architecture, and revised if needed. Finally, wrappers may be used to adapt the selected COTS components (hiding extra services not required, or adapting their interfaces for compatibility or interoperability reasons), and some glue language can be used to compose and coordinate the component interactions (see Figure 11.1).

In this new setting, the real search and selection processes of COTS components have become the cornerstone of any effective COTS development. These processes currently face serious limitations, generally for two main reasons. First, the information available about the components is not detailed enough for their effective selection. In this case, the black-box nature of COTS components hinders the understanding of their internal behavior. Moreover, only functional properties of components are usually taken into account, while some other information crucial to component selection is missing, such as protocol or semantic information (Vallecillo et al. 1999), or nonfunctional requirements (Chung et al. 1999; Rosa et al. 2001).

Second, the search, selection, and evaluation criteria are usually too simplistic to provide practical utility. These component searching and matching processes are delegated to traders (usually based on human factors) that do not provide all the functionality required for an effective COTS component trading in open and independently extensible systems (e.g., the Internet), as discussed in Vasudevan and Bannon (1999).

There are at least two important issues in this perspective. The first one deals with a common and standard documentation and specification of both abstract components described in software architectures and those third-party components available in software repositories.

The second issue deals with the trading processes for COTS components. These processes may implement partially (or fully) the functionality required in the software architecture, by constructing the list of candidate COTS components and calculating all their possible and different combinations that fulfill partially (or fully) the architectural requirements from the assembly of candidate components. As a result of a trading perspective, the alternative component combinations of the software architecture could be shown to the system's designer, who could decide which of the alternatives better matches the original user's requirements.

This chapter looks into these COTS-CBD areas. First, it looks into the documenting of COTS components. COTS documents are useful for the search and selection tasks associated with the trading service. Second, the chapter looks into software architectures with COTS components. In this case, we think that the Unified Modeling Language-Real Time (UML-RT) is a suitable notation for COTS software architecture descriptions. Finally, to solve the gap analysis problem, we propose two associated processes: COTStrader (Iribarne et al. 2001) and COTSconfig (Iribarne et al. 2002). The COTStrader process is a tool that extends the Open Distributed Processing (ODP) trading service (ISO/IEC 1997) to look for COTS components. The COTSconfig process is a composition function that calculates all the possible component combinations from those stored in the list of candidates generated by the trader. In this chapter, we explain the connection of the above areas, mainly focusing on the trading service for COTS components. Accordingly, the chapter presents a trader-based development method to fulfill architectural requirements. This method is mainly focused on an experimental framework to test and justify the validity and the usefulness of trading processes in CBD.

The rest of the chapter is organized in six sections. The first one describes a COTS document and a simple example of a COTS-based application to illustrate the proposal. The second section describes some features of a trading process for commercial components. The third section describes a three-tier method to build systems with commercial components. Tier one looks at defining the system's requirements using a software architecture. Tier two looks at searching and selecting components that meet the architectural requirements using the trader service COTStrader. And tier three looks at producing configurations of the software architecture (COTSconfig) from those components found by the trading service. The fourth section describes the technology used to develop the processes associated with the proposed method. Then, the fifth section describes the related works. Finally, the sixth section contains some concluding remarks.

## DOCUMENTING COTS COMPONENTS

COTS components are coarse-grained components that integrate several services and offer many interfaces. Component capabilities and usages are specified by interfaces. An interface is "a service abstraction defining the operations that the service supports, independently of any particular implementation" (Szyperski 1998).

Interfaces can be described using many different notations, depending on the information that we want to include, and the level of detail of the specification. In the Common Object Request Broker

Architecture (CORBA), an interface consists of the supported object public attributes, types, and methods. The Component Object Model (COM) follows a similar approach, but components may have several interfaces, each one describing the signature of the supported operations. The CORBA Component Model (CCM) also considers that component interfaces may describe not only the services they support, but also the ones they require from other components during their execution.

The current approaches at the signature level use the Interface Definition Language (IDL) to describe interfaces, which guarantee interoperability at this level among heterogeneous components. Interfaces can be written in different languages using different object models, living in different machines, and using different execution environments. Some IDL examples are those defined by CORBA, COM, and CCM.

On top of the signature level, the semantic level deals with the "meaning"—that is, the behavior (Leavens and Sitaaman 2000)—of the operations, though much more powerfully than mere signature descriptions. Behavioral semantics of components present serious difficulties when they are applied to large software systems: the computational complexity of proving behavioral properties of the components and applications hinders the interface's practical utility.

The semantic level can usually be described by using formal notations that range from the Larch family of languages (Dhara and Leavens 1996) using pre- and post-conditions and invariants to algebraic equations (Goguen et al. 1996), or refinement calculus (Mikhajlova 1999).

Finally, the protocol level just deals with the components' service access protocols, that is, the partial order in which components expect their methods to be called, and the order in which they invoke other methods (Vallecillo et al. 2000). This level, identified by Yellin and Strom (1997), provides more powerful interoperability checks than those offered by the basic signature level. Of course, it does not cover all the semantic aspects of components, but it is not weighed down with the heavy burden of semantic checks. At the protocol level, the problems can be more easily identified and managed, and practical (tool-based) solutions can be proposed to solve them.

At the protocol level, most of the current approaches enrich the IDL description of the components' interfaces with information about protocol interactions, using many different notations: finite-state machines (Yellin and Strom 1997), Petri nets (Bastide et al. 1999), temporal logic (Han 1999), or pi-calculus (Milner 1993).

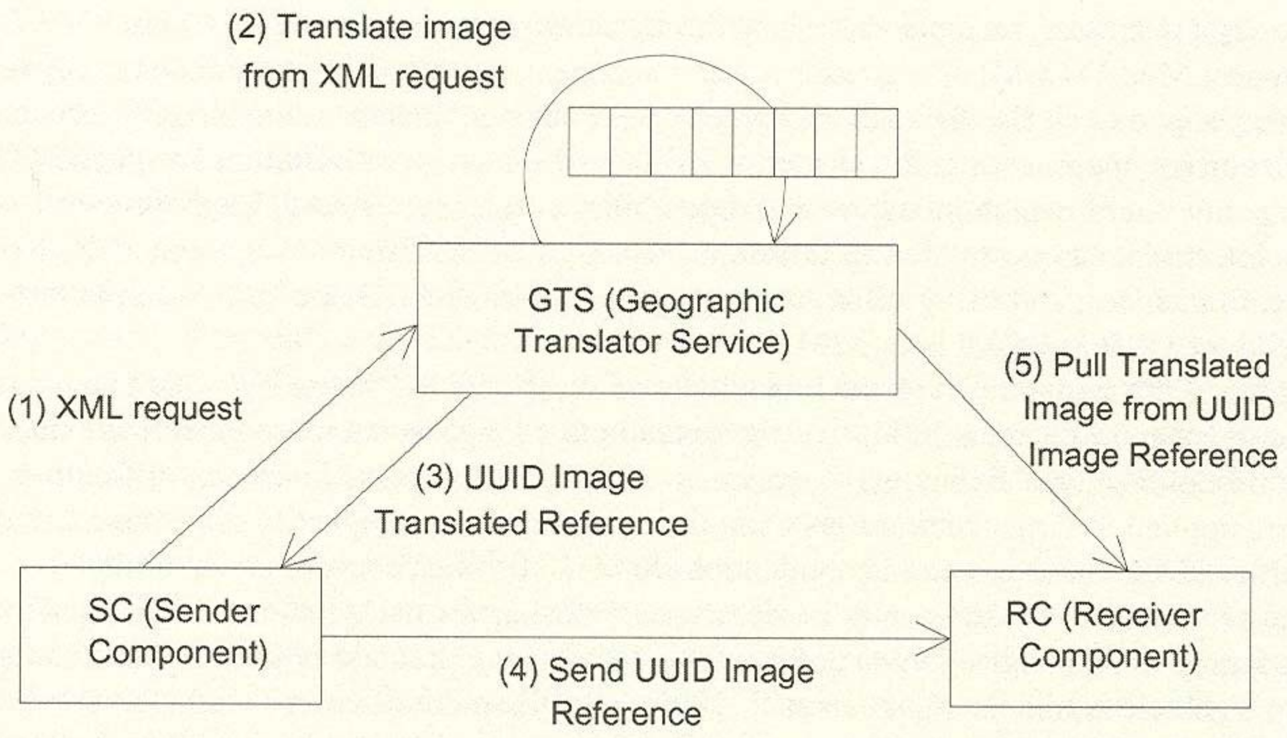
## An Example Using COTS Components

In order to illustrate our proposal, we will introduce a simple example that comes from the distributed Geographic Information Systems (GIS) arena. It consists of a common translation service between spatial images, usually known as Geographic Translator Service (GTS). Briefly, a sender component needs to send a formatted image to a receiver component, but instead of sending it directly, the sender uses a translator service to deal with all issues related to the image format conversion and compression. This simplifies both the sender and the receiver, taking away all those format-compatibility issues from them.

The way the service works is shown in Figure 11.2. First, the Sender forwards a translation 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>
```

Figure 11.2 **A Geographic Translator System (GTS) Example**


(2) Translate image from XML request
GTS (Geographic Translator Service)
(1) XML request
(3) UUID Image Translated Reference
(5) Pull Translated Image from UUID Image Reference
SC (Sender Component)
RC (Receiver Component)
(4) Send UUID Image Reference

Following this request, the GTS component downloads a zip-compressed DWG image from the http site. Then it generates a DXF image file with the same name and stores it in a buffer. After that, GTS associates a unique identifier UUID to the translated DXF image. Finally, it returns the identifier to the sender component, which gets the required image by pulling the converted file from the GTS buffer.

For our discussion, we will just concentrate on the GTS subsystem. It uses a main component called Translator that provides the target translation service (see Figure 11.6).

## COTS Documents

Similarly to software components, a mechanism to document commercial components is very useful in tasks of search, selection, evaluation, and assembly of components. For these processes, it is very important to use complete, concise, and unambiguous specifications of components in order to guarantee successful COTS software development.

A commercial component document (like an identity credential and a contract) could be used by developers to describe particular issues of the target component to build. System architects could also use this kind of information, recovered in a commercial component document, to describe the components' architectural requirements in the software architecture.

A COTS document is a Document Type Definition (DTD) template based on the World Wide Web Consortium's (W3C) XML Schema language, used to describe COTS components. Figure 11.3 shows an instance of an XML-based COTS document for a simple software component called Translator, which translates between spatial image formats.

A COTS document deals with four kinds of information: functional, nonfunctional, packaging, and marketing. This kind of information can help to determine the component requirements when describing the software architecture and can be used as the information type that a trader manages.

The first part describes the functional (i.e., computational) aspects of the services provided by

Figure 11.3 **A COTS Document Example**

```
<COTScomponent name="Translator" xmlns="http://www.cotstrader.com/COTS-
XMLSchema.xsd" . . . >
    <functional>
        <providedInterfaces>
            <interface name="Translator">
                <description notation="CORBA-IDL">
                interface Translator {
                    boolean translate(in string request, out string UDDI);
                    boolean get(in long UDDI, out string URL); };
                </description>
                <exactMatching href=" . . . /servlet/CORBA.exact"/>
                <behavior notation="Larch-JML">
                    <description> . . . </description>
                    <exactMatching href=" . . . /servlet/LarchJML.exact"/>
                    <softMatching href=" . . . /servlet/LarchJML.soft"/>
                </behavior>
            </interface>
        </providedInterfaces>
        <requiredInterfaces>
            <interface name="FileCompressor"> . . . </interface>
            <interface name="ImageTranslator"> . . . </interface>
            <interface name="XDR"> . . . </interface>
            <interface name="XMLBuffer"> . . . </interface>
        </requiredInterfaces>
        <choreography>
            <description notation="pi-protocols" href=" . . . /translatorProtocol.pi" >
            <exactMatching href=" . . . /servlet/PI.exact"/> <softMatching href=" . . . /servlet/
PI.soft"/>
        </choreography>
    </functional>
    <properties notation="W3C">
        <property name="capacity"> <type>xsd:int</type> <value>20</value> </property>
        <property name="isRunningNow"> <!—dynamic property—>
            <type>xsd:bool</type> <value href="http:// . . . /servlet/GTS.running"/> </property>
        <property name="keywords"> <type>xsd:string</type>
            <value> spatial image, format, conversion, compression, GIS </value> </property>
    </properties>
    <packaging>
        <description notation="CCM-softpkg" href=" . . . /AnImplementation.csd"/>
    </packaging>
    <marketing>
        <expirydate>2004–2-11</expirydate> <certificate href=" . . . /card.pgp" />
        <vendor>
            <companyname> . . . </companyname> <address> . . . </address> . . .
        </vendor>
    </marketing>
</COTScomponent>
```

the component, including both syntactic (i.e., signature) and semantic information. Unlike most object services (which contain only an interface), functional definition will host the set of interfaces offered by the component, and the set of interfaces that any instance of the component may require from other components when implementing its supported interfaces. The documentation may also list the set of events that the component can produce and consume, as defined in component models such as CCM, Enterprise JavaBeans (EJB), or Enterprise Distributed Object Computing (EDOC) (OMG 2001).

Since we did not want to commit to any particular notation of expressing the functional information contained in the XML templates, the "notation" attribute can be present in many fields of a COTS document, with several predefined values. This attribute can be used by clients and servers to agree with the values they want to use.

Behavioral information can be described directly in the document, or in a separate file by the "href" attribute, or even omitted (behavioral information is optional; only signature information is mandatory).

A third (optional) element called "choreography"—commonly known as protocol information (Canal et al. 2000; Vallecillo et al. 2000)—deals with the semantic aspects of the component that globally describe its behavior and that cannot be captured by the semantics of the individual interfaces. A protocol refers to the relative order in which the component expects its methods to be called and the way it calls other components' methods.

Syntactical (i.e., the interfaces), behavioral, and protocol descriptions can be referred to a couple of (optional) procedures that will allow the trader to do the matchmaking programs. In this example, LarchJML.exact is the name of a program that is able to decide whether two behavioral descriptions A and B, written in Larch-JML (Leavens et al. 1999), satisfy that A can replace B. Analogously, a second program, LarchJML.soft, is the one in charge of implementing the soft matchmaking process.

The second part of the COTS component template describes nonfunctional aspects (e.g., Quality of Service (QoS), nonfunctional requirements, etc.) in a similar way to ODP, that is, by means of service properties (ISO/IEC 1997). In order to deal effectively with nonfunctional requirements, we use some principles taken from a qualitative approach called nonfunctional requirements (NFR) Framework (Chung et al. 1999). This approach is based on the explicit representation and analysis of nonfunctional requirements. Considering the complex nature of nonfunctional requirements, we cannot always say that nonfunctional requirements are completely accomplished or satisfied. We have studied the importance of nonfunctional information and how to include it into COTS documents (Iribarne, Vallecillo et al. 2001).

The ODP way has been adapted to describe nonfunctional properties, that is, using "properties." They are the usual way (name, type, and value) in which the nonfunctional aspects of objects, services, and components are expressed in the literature. Dynamic properties can also be implemented in this approach, indicating the reference to the external program that will evaluate their current values. Keyword-based searches are allowed too, and they use the special "keywords" property. Complex properties and traceability can also be considered in a COTS document (see Iribarne, Vallecillo et al. 2001).

The third part contains the packaging information to download, deploy, and install the COTS component that provides the required service. It includes implementation details, context and architectural constraints, and so on. In this example, the CCM "softpackage" (OMG 1999) description style is used (see Figure 11.3). This information allows us to describe resources, configuration files, the location of different implementations of the component for different operating systems, the way those implementations are packaged, the resources and external programs they need, and so on.

Finally, some other nontechnical details of the service, and the component implementing it, are described in the marketing section, which includes licenses, certificates, vendor information, and so on.

## TRADING FOR COTS COMPONENTS

Trading is a well-known concept for searching and locating services. In a trader-based architecture, a client component that requires a particular service can query a matchmaking agent—called the trader—for references to available components that provide the required kind of service. Moreover, enhanced traders with quality-of-service (QoS) facilities can provide the means of self-configuring multimedia applications. In this chapter, we will just concentrate on COTS component trading. However, most of our discourse is also applicable to all disciplines in which trading is required.

After analyzing specific characteristics of CBD in open systems, we present in Table 11.1 the features and characteristics that traders should have in order to provide an effective COTS component trading service in these open environments.

Existing traders mostly follow the ODP model (ISO/ITU-T 1996), and therefore they present some limitations when their features are compared against the previous list of requirements in Table 11.1. Based on the experience obtained from the existing industrial implementations of the trading service (e.g., Distributed Object Group and IONA [2001]), and based on some closely related works (e.g., Merz et al. [1994] and Vasudevan and Bannon [1999]), we can see that current traders:

- use homogeneous object models only
- use direct federation
- do not allow service composition or adaptation
- work with "exact" matches at the signature level only
- do not allow multiple interfaces
- are based on a push model only

COTStrader, a trader that can overcome these limitations, is specifically designed to deal with COTS components in open environments. COTStrader uses two kinds of templates to register and look for components: a COTScomponent template similar to the "COTS Document" and a COTSquery template, respectively.

In order to import (or get) a service from the repository, the client uses a COTSquery document, which contains the selection criteria that must be used by the trader to look for services (COTS components). Using this kind of document, the trader covers the repository, looking for similar COTS documents. The trading process returns a list of candidate documents, accomplishing the search criteria fixed in the client query document. Figure 11.4 shows a COTSquery example of searching one (or more) Translator component(s).

As we can see here, a COTSquery template consists of five main parts. The main features of the required service can be directly described into the document or even with an additional COTScomponent document referred inside the "COTSdescription" part, in the COTSquery document. This COTScomponent document is the one used by the trader to match it with the candidate document being analyzed.

Table 11.1

**Features of a Trading Process for Commercial Components**

| Features | Description |
|---|---|
| Heterogeneous | A trader should not restrict itself to a particular component or object model, but it should be able to deal with different component models and platforms, such as CORBA, EJB, COM, .NET, etc. |
| Federation | Cooperating traders can federate using different strategies. The direct federation approach requires the traders to directly communicate (and know about) the ones they federate with. In the repository-based federation, multiples traders read and write to the same service offer repository. |
| Search engines | 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 of classes in a domain-specific property space. |
| Softmatches | Traditional exact matches between imports and exports are very restrictive in real situations, in which more relaxed matching criteria should be used. Therefore, partial matches should be allowed when building the list of candidate components. |
| Extensible and scalable | Component behavior, NFRs, QoS, marketing information, and semantic data should also be considered. The information managed by the trader should be able to be extended by users in an independent way, and still the trader should be able to use all its functionality and capabilities. |
| Compose and adaptation | Current traders focus on one-to-one matches between client requests and available service instances. A compositional trader should also consider one-to-many matches, in which composing several available services, which together provide the services, can also fulfill a client request. |
| Multiples interfaces | Components simultaneously offer several interfaces and, besides the services, should be defined in terms of sets of interfaces. This fact has to be specially considered when integrating components, since conflicts may appear between components offering common interfaces. |
| Subtyping | 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 type subtyping (or type conformance). Subtyping needs to be defined in order to cope with syntactic and semantic information, protocols, QoS, etc. |
| Store and forward | If a trader cannot fully satisfy a request, either it automatically replies back to the client with a denial (automatic behavior) or it stores the request and postpones the reply until a suitable service provider is found (store and forward). |
| Push and pull | In a push model, exporters directly contact the trader to register their services with it. Bots and search engines, used to enhance current COTStrader, use a push model, crawling the Web looking for services and "pushing" them into the traders. |
| Heuristics and metrics | Users should be able to specify heuristic functions and metrics when looking for components, especially in the case of soft matchmaking. |
| Delegation | If traders cannot resolve requests they should be able to delegate them to other (smarter) traders. Delegation of the complete request or just parts of it is desirable. |

Figure 11.4 **A Query Template Example**

```
<COTSquery name="TranslatorQuery">
      <COTSdescription href="http:// . . . /Translator.xml"/>
      <functionalMatching>
         <interfaceMatching> <exactMatching href=" . . . /servlet/exactmat"/> </
interfaceMatching>
         <choreographyMatching> <softMatching/> </choreographyMatching>
      </functionalMatching>
      <propertyMatching>
         <constraints notation="XQuery"> (capacity <= 17) </constraints>
         <preferences notation="ODP">random</preferences>
      </propertyMatching>
      <packagingMatching notation="XQuery">
         description/notation ="CCM-softpkg" and description/implementation/os/
name="WinNT"
      </packagingMatching>
      <marketingMatching notation="XQuery">vendor/companyname="IBM"</
marketingMatching>
</COTSquery>
```

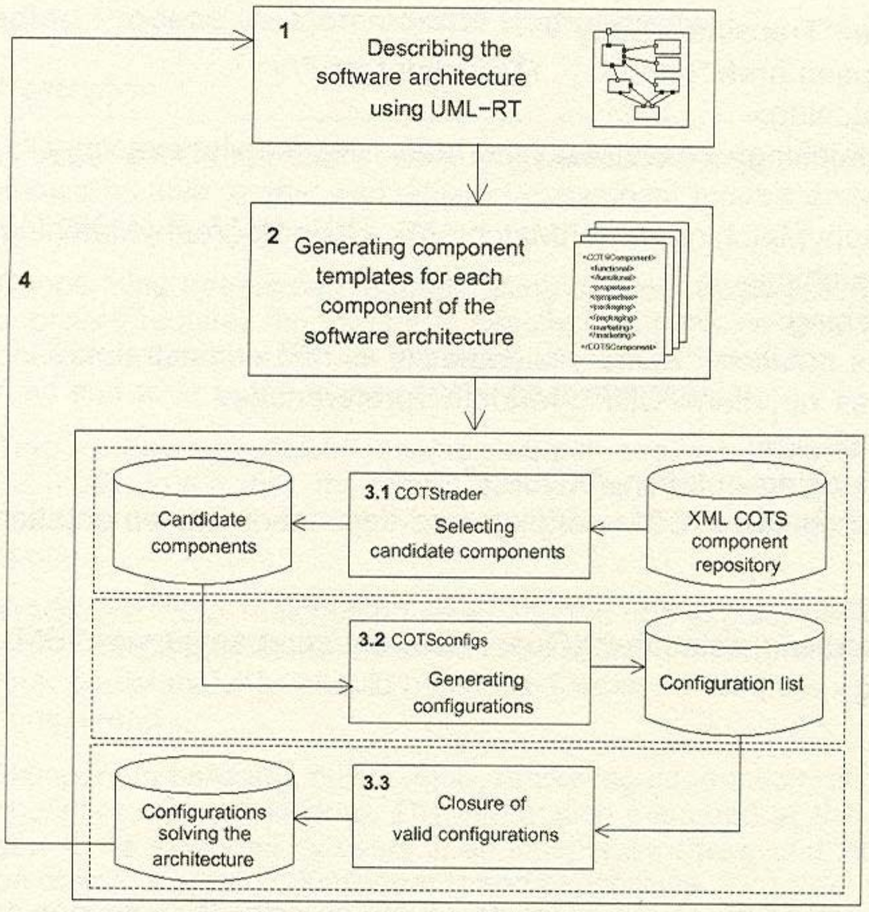
The other four parts of the COTSquery document describe the selection criteria to be used. In the functional part, the client may specify whether the matchmaking process should be *exact* or *soft,* and optionally the matchmaking program to be used (the program originally stated in the target COTS component description is ignored) (Zaremski and Wing 1995). For example, in the query template shown in Figure 11.4, we can see how a soft matching action is only desired at a protocol (choreography) level, whereas at the syntactical level an exact matching action and a program are desired by the client.

If a client does not offer a soft or exact matching program in the query document, the trader looks for it inside the document of the candidate component (candidate document) that is being analyzed from the trader's repository. Otherwise, the trader looks for default matching programs related to it. The candidate document is refused by the trader if an exact matching was required and no program was found. The candidate document is included in the candidate list (to be returned) if a soft matching was required, but no program was found.

Property-based matching is done in the usual way by ODP traders, using constraints and preferences. Constraints are boolean expressions consisting of service property values, constants, relational operators (<, >=, =, !=), logical operators (not, and, or), and parentheses, which specify the matching criteria to include a component in the trader's list of candidates for the current search. We have used the notation defined in the W3C's XML QueryAlgebra proposal to write the expression. On the other hand, the preferences can sort out the list of candidates according to a given criterion using terms like "first," "random," "min(*expr*)" and "max(*expr*)," where *expr* is a simple mathematical expression involving property names (ISO/ITU-T 1996).

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

Figure 11.5 **The TBDM Architecture**


1
Describing the software architecture using UML-RT
2
Generating component templates for each component of the software architecture
4
Candidate components
3.1 COTStrader
Selecting candidate components
XML COTS component repository
3.2 COTSconfigs
Generating configurations
Configuration list
3.3
Closure of valid configurations
Configurations solving the architecture

## PROCESSES FOR A TRADING-BASED DEVELOPMENT METHOD

To assess the usefulness of the trading process for CBD, it has been integrated into a method to build COTS-based systems by automating most of the search and selection activities. In order to produce systems with commercial components, architects, designers, and builders must accept the trade-offs among three main concerns: user requirements, system architecture, and COTS products.

In this section, we will introduce all the processes dealing with a trading-based development method (TBDM), that is, a software development proposal to build COTS-based systems that require the use of quick prototypes of the system's software architecture. This method tries to solve an important problem at design level, known as gap analysis in the CBD literature (Cheesman and Daniels 2001). The main purpose is to approach the architectural design requirements of the components and those related with particular implementations available in the market of software components (i.e., commercial components). Some important tasks, tools, and methods, very common in requirements engineering (requirement elicitation, analysis, specification, and validation), are beyond the scope of this chapter, which is rather focused on the design level.

Figure 11.5 shows the schema of all connected tasks that conform to the automated method. The process initially describes the software architecture (SA) of the system, which defines its high-level structure, exposing its organization as a collection of interacting components (step 1). The SA decomposes the application requirements into a hierarchical criteria set, which usually includes component functionality, extrafunctional requirements, architectural constraints, and other nontechnical factors such as vendor guarantees and legal issues.

Subsequently, the process continues by extracting the component requirement information from the SA (step 2). This step produces a list of abstract services, which can easily be expressed by means of COTScomponent documents (i.e., abstract documents).

Using this information (i.e., the list of COTScomponent documents), the next step begins with the activities of component selection. In this case, the COTStrader can be queried to look for those COTS components that provide the desired requirements (step 3.1). The trader returns a list of candidate components to each abstract document. A candidate list contains the collection of those particular documents (i.e., COTS documents of existing implemented components) that match with the information of the abstract document being considered.

The process keeps on trying to build the system from these lists of candidate components. The COTSconfig process (step 3.2) carries out this purpose by calculating a list of all possible combinations between the candidate components. A valid component combination represents a closed configuration of the system that partially (or fully) solves the abstract component requirements of SA. A closed configuration does not require any external component to work, that is, all services required by its constituent components are provided by a component in the configuration. Therefore, a new process deals with the closing of configurations (step 3.3).

Finally, a list of all closed configurations is shown to the system designer (step 4), so that a decision can be made as to (1) the best configuration that matches the user requirements, (2) which components are still missing and hence need to be developed, and (3) the extent to which the initial software architecture should be changed (and whether it is worth changing) in order to accommodate to the COTS components found. The selection, validation, evaluation, and adaptation of configuration activities are beyond the scope of this chapter. Instead, we focus on the assessment of the usefulness of integrating a COTS trader with CBD methodologies (e.g., Capability Maturity Model (CMM) or Rational Unified Process (RUP) approaches using software components).

The following sections describe these steps in more detail, and the Geographic Translation Service example is used to illustrate the trader-based development method. The discussion that follows will refer to Figure 11.5.

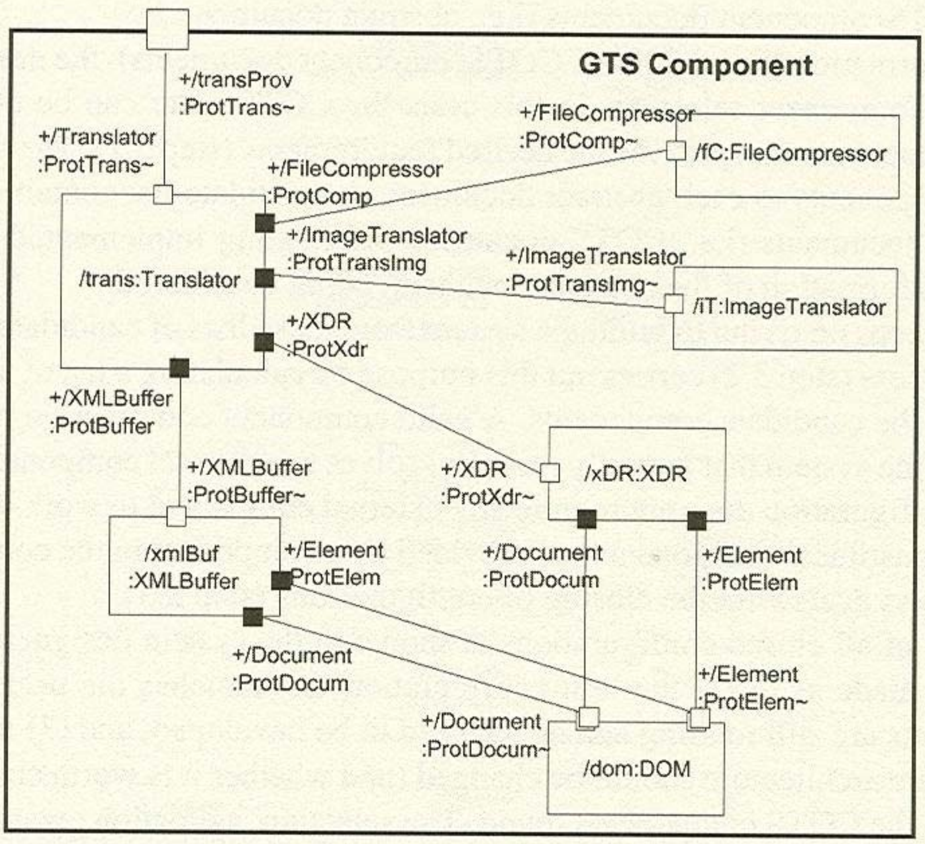
## Describing the Software Architecture

Complex software systems require expressive notations to represent their architecture. In general, the software architecture defines its high-level structure, exposing its organization as a set of interacting components.

Traditionally, specialized architecture description languages (ADL) have been used to provide a formal description of the structure and the behavior of an application's architecture (Medvidovic and Taylor 2000). The formality and the lack of visual support of most ADLs, however, have encouraged the quest for more user-friendly notations. In this respect, the Unified Modeling Language (UML) notation is clearly the most promising candidate, since it is familiar to developers and, to a certain extent, nontechnical people, it offers a close mapping to implementations, and it has commercial tool support.

The problem is that the current definition of UML does not offer a clear way of encoding and modeling the architectural elements typically found in the architectural description languages as discussed by Garlan et al. (2002) and Medvidovic et al. (2002), for instance. Until the new UML 2.0 is released (which is expected to support component-based development and run-time architectures of applications), the only widely accepted proposal for documenting software architectures available now is probably UML-RealTime (UML-RT) (Selic and Rumbaugh 1998).

Figure 11.6 **The GTS Software Architecture in the UML-RT Notation**



UML-RT is a visual modeling language with formal semantics for specifying, visualizing, documenting, and automating the construction of complex, event-driven, and distributed real-time systems. UML-RT uses some graphical notations to describe the software architecture, such as capsules (i.e., components), ports (i.e., provided and required interfaces), protocols (i.e., choreography), and connectors (to bind components through the ports).

Figure 11.6 shows the GTS software architecture using the UML-RT notation. This figure represents the first stage (step 1) of the TBDM method (shown in Figure 11.5).

As we can see, a general capsule is used to describe the whole software architecture. A UML-RT capsule can be composed of one or more capsules (i.e., components in our case). Please notice that the GTS software architecture capsule represents the GTS component capsule in Figure 11.2.

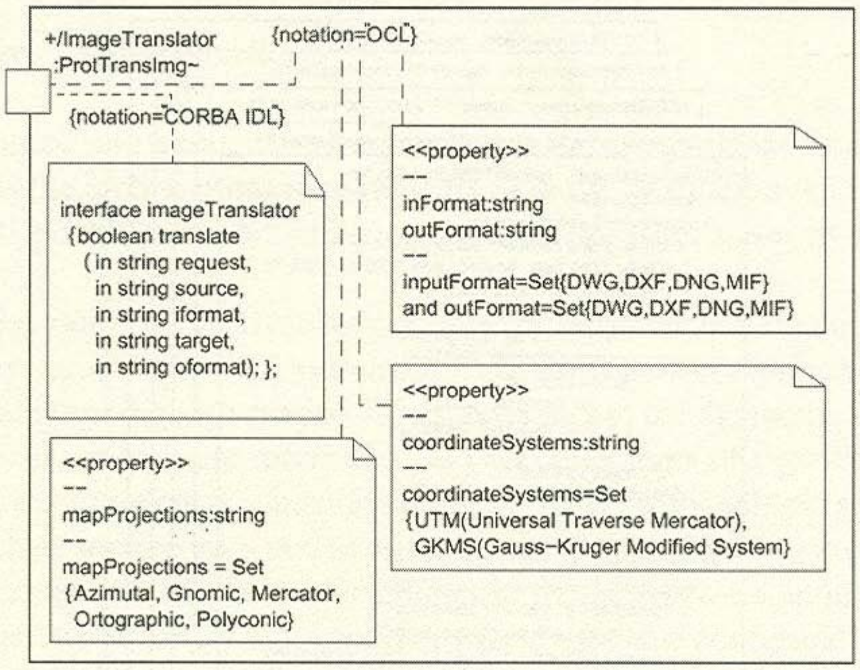
The +/transProv:ProtTrans~ port represents the boundary that communicates both the sender and receiver components with the inner part of the GTS capsule. This port connects directly with the +/translator:ProtTrans~ dual port, which is a part of the /trans:Translator component.

The GTS software architecture is composed of six components (i.e., capsules). The main component is /trans:Translator, which requires four additional components:

1. a file compressor called /fC:FileCompressor
2. a translator of spatial images called /iT:ImageTranslator
3. a component for intermediate representation of data called /xDR:XDR
4. a buffer called /xmlBuf:XMLBuffer

The GTS subsystem also requires two document object model (DOM) interfaces called Document and Element. They are used by the XDR and XMLBuffer components to support XML formats. These interfaces are available in software packages such as IBM XML4J or Sun JAXP.

Figure 11.7 **A Capsule Describing the Component Requirements**



In the GTS software architecture, only an instance of the base component DOM is used, but Document and Element ports are duplicated to handle both /xDR:XDR and /xmlBuf:XMLBuffer components.

Moreover, there is some important information that must be recovered inside a capsule. This information, which is recovered by UML notes and tagged values inside a capsule (i.e., component), is related with that included in a COTS document. For example, Figure 11.7 shows the internal requirements imposed on the ImageTranslator component. To simplify, we just represent the signature level of interfaces and nonfunctional information, but the remaining information of a COTS document can be represented in a similar way.

The signature level of interfaces can be described by some particular IDL notations. Specifically, the CORBA IDL notation is used to describe interfaces. This description is directly included inside a UML note and connected with the corresponding port (a UML-RT port refers to interfaces). Also, an external tagged value—which is connected with the IDL note—determines the notation type to describe the interface's signature level (e.g., notation *n* = "CORBA IDL").

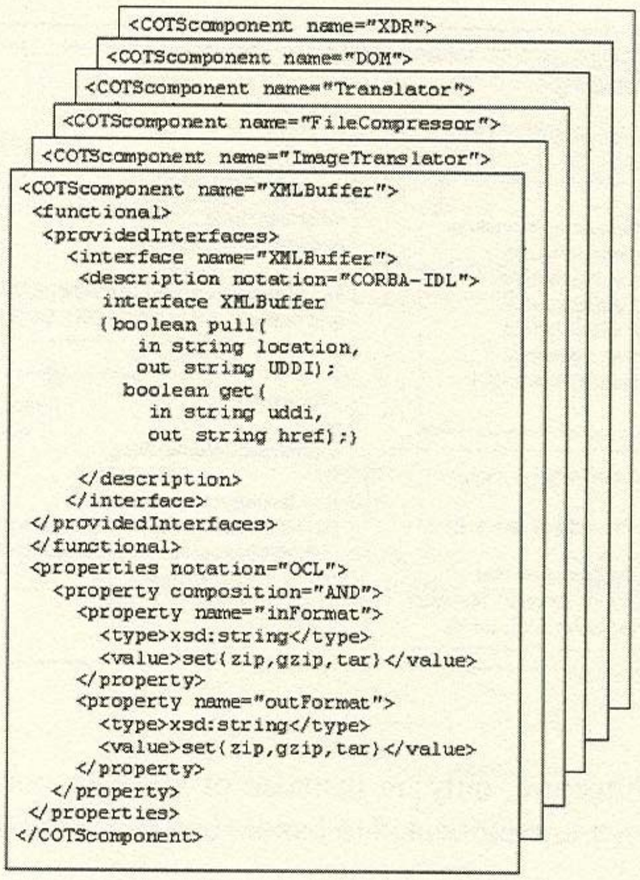
Properties are also described in a separate note. A property description begins with the "<<property>>" stereotype name. Next, the property description is indicated with a particular notation. As the interface shows, the description notation is represented by using an external tagged value. For example, the ImageTranslator capsule describes three properties in separate notes. It uses an external tagged value (notation = "OCL") connected with each property note. A "—" symbol is used to separate several parts in a property description note: (1) the "<<property>>" stereotype header; (2) the declaration of those types by the object constraint language (OCL) property description; and (3) the body of the property description.

Once the software architecture is drawn—using the Rational Rose RealTime package in our case—the information about the components, the services they offer and require, and their properties is extracted from the UML-RT diagram (i.e., capsule information, as we have discussed).

This process represents the second stage (step 2) of the TBDM method (Figure 11.5).

For that purpose, we have a process that parses the files produced by Rational Rose RealTime and generates a list of COTS documents (i.e., COTSdocument templates) with the description of

Figure 11.8 **A Template List of COTS Components**



```
<COTScomponent name="XDR">
<COTScomponent name="DOM">
<COTScomponent name="Translator">
<COTScomponent name="FileCompressor">
<COTScomponent name="ImageTranslator">
<COTScomponent name="XMLBuffer">
 <functional>
  <providedInterfaces>
    <interface name="XMLBuffer">
     <description notation="CORBA-IDL">
       interface XMLBuffer
       {boolean pull(
           in string location,
          out string UDDI);
         boolean get(
           in string uddi,
           out string href);}

     </description>
    </interface>
 </providedInterfaces>
 </functional>
 <properties notation="OCL">
   <property composition="AND">
     <property name="inFormat">
       <type>xsd:string</type>
       <value>set(zip,gzip,tar)</value>
     </property>
     <property name="outFormat">
       <type>xsd:string</type>
       <value>set(zip,gzip,tar)</value>
     </property>
   </property>
 </properties>
</COTScomponent>
```

the components found in the software architecture. Figure 11.8 shows the list of six COTScomponent templates generated for the GTS software architecture (shown in Figure 11.6).

In addition to this, we have a generic tool that processes XML Metadata Interchange (XMI) files and produces the COTScomponent templates, since we did not want to commit to any particular tool or graphical notation.

## Looking for COTS Components (Trading)

Besides obtaining the architectural needs, the next stage of the TBDM method looks over the component search and selection activities (step 3.1). This stage deals with the trader process (COTStrader). The trader generates a list of candidate components from the repository when it is queried with a client component template to be searched. In our case, the trader service is queried just six times using those document templates with the architectural requirements. These were extracted from the software architecture in the previous step. This search process will generate a list of candidate components for each query requested.

The trading service was described in the "Trading for COTS Components" section. As we could see there, the matching operations, which look for COTS components and generate a list of candidate components, start with soft matches (basically, just by looking for keywords). As the software architecture gets progressively refined, these initial soft matches get more and more exact in each iteration (see again Figure 11.5). Typical (and 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 level of matching is very useful in theory, 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, or mention their semantics.

**Building Alternative Implementations**

As previously discussed, the trading service produces a list of candidate components (i.e., candidate list) that fulfills the architectural requirements. A separate process (COTSconfig) generates the collection of all possible combinations of components from those in the list of candidate components (step 3.2).

Although a configuration should resolve partially (or fully) the architectural requirements, not all of them are really valid to build the system. A COTS software component is identified by two sets of incoming and outgoing information flows, basically at the syntactic, semantic, and protocol level (i.e., services), which are referred to as two collections of provided and required services. Nevertheless, a component combination (configuration) with multiple services may generate some problems, such as service gaps and overlaps. Gaps happen when none of the components in a configuration provides any of the services required by the software architecture. On the contrary, overlaps happen when two or more components in the same configuration provide the same service. The aim is to find those configurations without service gaps or service overlaps.

Here below, we will discuss an algorithm that generates valid configurations. In order to explain this algorithm, we will use a particular component notation, focused only on the provided and required services of the component (i.e., on the functional information), but not on the properties, packaging, or marketing information of a COTS component (as there is no influence in a configuration). In relation to the functional information, the algorithm considers the services independently of its information level (i.e., the syntactic, semantic, or protocol level).

To simplify the discussion of the configuration algorithm, let us now consider a component $C$ by two sets of services $C = (P,R)$, where $P$ is the set of supported (or provided) services ($P = \{P_1, \ldots, P_n\}$), and $R$ is the set of required services, $R = \{R_1, \ldots, R_m\}$. For simplicity, we are writing $C.P$ and $C.R$ to refer to both sets of services. At the signature level, $P_i$s and $R_j$s represent standard interfaces (e.g., CORBA or COM interfaces) composed just of a set of public attributes and methods. At the protocol level, $P_i$ and $R_j$ describe a "role." At the semantic level, they correspond to a description of an interface decorated with semantic information (e.g., with pre- and postconditions).

The GTS example will be used to explain the configuration algorithm. We are supposing that the COTStrader process has generated the list of eight candidate components shown in the right column of Figure 11.9. For the algorithm, $C_B(A)$ refers to the candidate list, $B$ refers to the trader repository, and $A$ refers to the software architecture. On the left column, we are show the six components of the GTS subsystem (see Figure 11.6).

For simplicity, we are using just two characters to name components and services. For example, the FileCompressor component will be written as $FC$, and a service as $P_{FC}$. Both $C_2$ and $C_6$ candidate components require two external services, $R_{TL}$ and $R_{FL}$ respectively. The first one represents a Tool interface, containing a collection of methods to transform spatial images (e.g., sizing, rotation, rolling, and so on). The second one represents a Filter interface, containing a collection of methods with some special effects on spatial images (e.g., noise effect, thresholding, edge detect, shading, segmentation, and so on). If these components are then considered for a configuration, this must be closed first in order to produce a working application. The last step of the process deals with this task (step 3.3 in Figure 11.5), closing the configurations with regard to the repository $B$.

Figure 11.9 **The Components of the GTS Architecture Against the Candidate List**

| Component name | GTS architecture | $C_B$(GTS): Candidate components |
|---|---|---|
| FileCompressor | $FC=\{P_{FC}\}$ | $C_1=\{P_{EL},P_{DO}\}$ |
| ImageTranslator | $IT=\{P_{IT}\}$ | $C_2=\{P_{EL},P_{DO},R_{TL}\}$ |
| XDR | $XD=\{P_{XD},R_{EL},R_{DO}\}$ | $C_3=\{P_{FC}\}$ |
| XMLBuffer | $BF=\{P_{BF},R_{EL},R_{DO}\}$ | $C_4=\{P_{FC},R_{EL},R_{DO}\}$ |
| DOM | $DM=\{P_{EL},P_{DO}\}$ | $C_5=\{P_{IT}\}$ |
| Translator | $TR=\{P_{TR},R_{FC},R_{IT},R_{XD},R_{BF}\}$ | $C_6=\{P_{IT},P_{FC},R_{FL}\}$ |
| | | $C_7=\{P_{BF},P_{TR},R_{EL},R_{DO}\}$ |
| | | $C_8=\{P_{XD},R_{EL},R_{DO}\}$ |

The configuration algorithm (i.e., COTSconfig process) tries to build a set ($S$) with all the possible configurations obtained from the candidate list $C_B(A)$, which was previously generated by the trading process (i.e., COTStrader process). Figure 11.10 shows a backtracking algorithm that implements this process. It produces—from the set of candidates, $C_B(A)$, and from the application A—the set S of valid configurations (line 11). The initial invocation of the algorithm is $S$ = Æ, Sol = Æ, and configs(1,Sol,S). Each configuration (line 9) is generated by trying all candidates, incorporating those interfaces $C_i.P_j$ not yet included in $A$, and discarding those already considered (lines 8 and 10). When the algorithm finishes, $S$ contains all configurations. Because of the way in which the algorithm works, no service gaps or overlaps may occur, and therefore, it produces some valid configurations.

The complexity of this algorithm is $O(L2^n)$, where $n$ is the number of interfaces offered by all candidate components in $C_B(A)$, and $L$ is the complexity of the substitutability operator used (i.e., at the signature level, protocol level, or semantic level operator). To reduce the exponential complexity, we could change the algorithm into a "branch and bound" one, which uses some upper bounds to prune many of the options in the exploration tree, thereby improving notably the execution time of the algorithm.

Once all configurations have been generated, we need to close them in order to get a "complete" application (step 3.3). The closure process of a given configuration can be carried out by applying any of the existing algorithms that calculate the transitive closure of a set (i.e., a configuration) with regard to another bigger set (in this case, the repository). Therefore, we may need to invoke the COTStrader again to look for those external services until we get a closed configuration.

Figure 11.11 shows some results of the GTS example, generated by the configuration algorithm from the candidate list, shown in Figure 11.9. Here, only twenty-four configurations are valid, although other discarded configurations are shown for completeness. Columns 2 to 9 (labeled $C_1$–$C_8$ indicate the services provided by the components in each combination. Column 10 (labeled configurations) describes the configuration, hiding the appropriate service too (e.g., we represent the hiding services as "$C_3$–$\{P_{DO}\}$"). Columns 11 and 12 (labeled Res. and Cd) indicate whether the configuration respects the application's structure and whether it is a closed configuration, respectively. As for that, an application is closed if all services required by its constituent components can be served internally, that is, without requiring external services from the components outside the application. Note that valid configurations may not be closed. Although they do not contain gaps with reference to the original services specified in the architecture, configurations may still contain a COTS component that requires some external services not contemplated

Figure 11.10 **Obtaining All Valid Configurations**

```
function configs(i,Sol,S)
        // 1? i ? size(C_B(A)) traverse the repository
        // Sol is the configuration being built
        // S contains the set of all valid configurations A
        if i ? size(C_B(A)) then
                for j := 1 to size(C_i.P) do // all service in C_i
                        // we try to include C_i.P_j service in Sol
                        if {C_i.P_j} ? Sol.P = Ø then // C_i.P_j { EMBED Equation.3 }
                                Sol := Sol { EMBED Equation.3 }{C_i.P_j};
                                if A.P{ EMBED Equation.3 }Sol.P then // Is Sol a
                                        S := S { EMBED Equation.3 }Sol; // if so, it
                                else // but if there are still service gaps . . .
                                        configs(i,Sol,S); // search in C_i . . .
                                Endif
                                Sol := Sol – {C_i.P_j};
                        endif
                endfor
                configs(i+1,Sol,S); // Next in C_B(A).
        endif
endfunction
```

in the original design. This situation is not common in real applications. For instance, if we install a software component in our computer, we will soon realize that it needs another additional (and apparently unrelated) component, which should be installed for the application to work.

The "respect" and "closure" concepts, together with the collection of operators used by the configuration algorithm, are defined in Iribarne et al. (2002).

For instance, configuration 1 contains all candidate components, except $C_2$, $C_4$, and $C_6$, and each component provides just one interface, except $C_1$, which offers two. This configuration is closed and it respects the application structure. Given the twenty-four configurations, five of them are closed, and twenty respect the structure.

Now it is a decision of the system's designers to select the configuration that best suits their requirements from this list of valid configurations or to revisit the original architecture.

It is important to observe that the process described here has been defined for complete applications. However, it could also be used for some parts of an application. In this way, we could allow the designer to decide which parts of the whole application to implement with COTS components from the repository, applying the process just to the selected parts.

On the other hand, the application of gap analysis is an important feature to assess the difference between stated requirements and existing components, making a compromise to the requirements in order to deliver the solution in a faster and cheaper way (Cheesman and Daniels 2001).

## TECHNOLOGY USED

All the processes described in the trader-based development method have been implemented in Java. Rational Rose RealTime has been used to describe the software architecture, which adopts Bran Selic's original UML-RT version. The W3C's XML 1.0 and XML Schema notations have

Figure 11.11 **Some Results of the Configs Algorithm for the GTS Example**

| | C1 | C2 | C3 | C4 | C5 | C6 | C7 | C8 | Configurations | Res. | Cd. |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | $P_{EL}, P_{DO}$ | – | $P_{FC}$ | – | $P_{IT}$ | – | $P_{TR}, P_{BF}$ | $P_{XD}$ | $C_1, C_3-\{P_{DO}\}, C_5, C_7, C_8$ | true | true |
| 2 | $P_{EL}, P_{DO}$ | – | $P_{FC}$ | – | – | $P_{IT}$ | $P_{TR}, P_{BF}$ | $P_{XD}$ | $C_1, C_3-\{P_{DO}\}, C_6-\{P_{FC}\}, C_7, C_8$ | true | false |
| 3 | $P_{EL}, P_{DO}$ | – | – | $P_{FC}$ | $P_{IT}$ | – | $P_{TR}, P_{BF}$ | $P_{XD}$ | $C_1, C_4, C_5, C_7, C_8$ | true | true |
| 4 | $P_{EL}, P_{DO}$ | – | – | $P_{FC}$ | – | $P_{IT}$ | $P_{TR}, P_{BF}$ | $P_{XD}$ | $C_1, C_4, C_6-\{P_{FC}\}, C_7, C_8$ | true | true |
| 5 | $P_{EL}, P_{DO}$ | – | – | – | $P_{IT}$ | $P_{FC}$ | $P_{TR}, P_{BF}$ | $P_{XD}$ | $C_1, C_5, C_6-\{P_{IT}\}, C_7, C_8$ | true | true |
| 6 | $P_{EL}, P_{DO}$ | – | – | – | – | $P_{IT}, P_{FC}$ | $P_{TR}, P_{BF}$ | $P_{XD}$ | $C_1, C_6, C_7, C_8$ | false | true |
| – | $P_{EL}$ | $P_{DO}$ | $P_{FC}$ | – | – | – | – | – | NONE: $P_{IT}$, $P_{BF}$, $P_{XD}$ missing (gaps) | false | false |
| 7 | $P_{EL}$ | $P_{DO}$ | $P_{FC}$ | – | $P_{IT}$ | – | $P_{TR}, P_{BF}$ | $P_{XD}$ | $C_1-\{P_{DO}\}, C_2-\{P_{EL}\}, C_3, C_5, C_7, C_8$ | true | false |
| 8 | $P_{EL}$ | $P_{DO}$ | $P_{FC}$ | – | – | $P_{IT}$ | $P_{TR}, P_{BF}$ | $P_{XD}$ | $C_1-\{P_{DO}\}, C_2-\{P_{EL}\}, C_3, C_6-\{P_{FC}\}, C_7, C_8$ | true | false |
| 9 | $P_{EL}$ | $P_{DO}$ | – | $P_{FC}$ | $P_{IT}$ | – | $P_{TR}, P_{BF}$ | $P_{XD}$ | $C_1-\{P_{DO}\}, C_2-\{P_{EL}\}, C_4, C_5, C_7, C_8$ | true | false |
| 10 | $P_{EL}$ | $P_{DO}$ | – | $P_{FC}$ | – | $P_{IT}$ | $P_{TR}, P_{BF}$ | $P_{XD}$ | $C_1-\{P_{DO}\}, C_2-\{P_{EL}\}, C_4, C_6-\{P_{FC}\}, C_7, C_8$ | true | false |
| 11 | $P_{EL}$ | $P_{DO}$ | – | – | $P_{IT}$ | $P_{FC}$ | $P_{TR}, P_{BF}$ | $P_{XD}$ | $C_1-\{P_{DO}\}, C_2-\{P_{EL}\}, C_5, C_6-\{P_{IT}\}, C_7, C_8$ | true | false |
| 12 | $P_{EL}$ | $P_{DO}$ | – | – | – | $P_{IT}, P_{FC}$ | $P_{TR}, P_{BF}$ | $P_{XD}$ | $C_1-\{P_{DO}\}, C_2-\{P_{EL}\}, C_6, C_7, C_8$ | false | false |
| 13 | $P_{DO}$ | $P_{EL}$ | $P_{FC}$ | – | $P_{IT}$ | – | $P_{TR}, P_{BF}$ | $P_{XD}$ | $C_1-\{P_{EL}\}, C_2-\{P_{DO}\}, C_3, C_5, C_7, C_8$ | true | false |
| 14 | $P_{DO}$ | $P_{EL}$ | $P_{FC}$ | – | – | $P_{IT}$ | $P_{TR}, P_{BF}$ | $P_{XD}$ | $C_1-\{P_{EL}\}, C_2-\{P_{DO}\}, C_3, C_6-\{P_{FC}\}, C_7, C_8$ | true | false |
| 15 | $P_{DO}$ | $P_{EL}$ | – | $P_{FC}$ | $P_{IT}$ | – | $P_{TR}, P_{BF}$ | $P_{XD}$ | $C_1-\{P_{EL}\}, C_2-\{P_{DO}\}, C_4, C_5, C_7, C_8$ | true | false |
| 16 | $P_{DO}$ | $P_{EL}$ | – | $P_{FC}$ | – | $P_{IT}$ | $P_{TR}, P_{BF}$ | $P_{XD}$ | $C_1-\{P_{EL}\}, C_2-\{P_{DO}\}, C_3, C_6-\{P_{FC}\}, C_7, C_8$ | true | false |
| 17 | $P_{DO}$ | $P_{EL}$ | – | – | $P_{IT}$ | $P_{FC}$ | $P_{TR}, P_{BF}$ | $P_{XD}$ | $C_1-\{P_{EL}\}, C_2-\{P_{DO}\}, C_5, C_6-\{P_{IT}\}, C_7, C_8$ | true | false |
| 18 | $P_{DO}$ | $P_{EL}$ | – | – | – | $P_{IT}, P_{FC}$ | $P_{TR}, P_{BF}$ | $P_{XD}$ | $C_1-\{P_{EL}\}, C_2-\{P_{DO}\}, C_6, C_7, C_8$ | false | false |
| – | – | $P_{EL}, P_{DO}$ | $P_{FC}$ | – | – | – | $P_{TR}, P_{BF}$ | $P_{XD}$ | NONE: $P_{FC}$, $P_{IT}$ missing (gaps) | false | false |
| 19 | – | $P_{EL}, P_{DO}$ | $P_{FC}$ | – | $P_{IT}$ | – | $P_{TR}, P_{BF}$ | $P_{XD}$ | $C_2, C_3, C_5, C_7, C_8$ | true | false |
| 20 | – | $P_{EL}, P_{DO}$ | – | – | – | $P_{IT}$ | $P_{TR}, P_{BF}$ | $P_{XD}$ | $C_2, C_3, C_6-\{P_{FC}\}, C_7, C_8$ | true | false |
| 21 | – | $P_{EL}, P_{DO}$ | – | $P_{FC}$ | $P_{IT}$ | – | $P_{TR}, P_{BF}$ | $P_{XD}$ | $C_2, C_4, C_5, C_7, C_8$ | true | false |
| 22 | – | $P_{EL}, P_{DO}$ | – | $P_{FC}$ | – | $P_{IT}$ | $P_{TR}, P_{BF}$ | $P_{XD}$ | $C_2, C_4, C_6-\{P_{FC}\}, C_7, C_8$ | true | false |
| 23 | – | $P_{EL}, P_{DO}$ | – | – | $P_{IT}$ | $P_{FC}$ | $P_{TR}, P_{BF}$ | $P_{XD}$ | $C_2, C_5, C_6-\{P_{IT}\}, C_7, C_8$ | true | false |
| 24 | – | $P_{EL}, P_{DO}$ | – | – | – | $P_{IT}, P_{FC}$ | $P_{TR}, P_{BF}$ | $P_{XD}$ | $C_2, C_6, C_7, C_8$ | false | false |
| – | – | – | – | – | – | – | $P_{TR}, P_{BF}$ | $P_{XD}$ | NONE: $P_{EL}$, $P_{DO}$, $P_{FC}$, $P_{IT}$ missing (gaps) | false | false |

been used to write the COTScomponent documents and develop the template schemas (the grammar of a COTS document), respectively. The COTStrader service has been implemented by using the IONA's ORB ORBacus and IBM XML4J, and the latter implements the trader repository.

A Web client and some servlets (working on the Apache Tomcat WebServer and under a Linux/Redhat platform) have also been implemented to support trading service from the Web site of our trading-based development method (www.cotstrader.com).

Finally, well-known formalizations have been used to implement certain parts of the proposal: for example, the object constraint language notation to describe the properties of a COTS template, and Leavens' JML notation (Leavens et al. 1999) to describe the semantic specification of COTS component interfaces (the "behavior" tag in Figure 11.3). Finally, a subset of the pi-calculus notation (Milner 1993) has been used to describe protocols (the "choreography" tag in Figure 11.3).

## RELATED WORK

The contributions presented in this chapter are related to two main research lines: research based on component acquisition and research based on the building of systems from commercial components.

### Component Acquisition

Component acquisition is related to the component search and selection processes from software repositories. These studies take into account architecture requirements (also known as applications engineering) and the component specifications available in well-known software repositories (also known as domain engineering).

Several studies focus on component acquisition. Here, we are underlining three of them. First, Rolland (1999) proposes a technique that captures requirements through transition diagrams (called maps) based on four basic models: the "as-is model, "to-be" model, COTS model, and integrated match model. This approach, however, does not propose any particular way of specifying COTS components, nor does it give any indication of how to carry out the syntactic and semantic matchmaking process between components.

Second, Goguen et al. (1996) present and discuss a set of criteria for searching and selecting components from a repository. However, this technique deals only with components that offer simple interfaces, and therefore the problems of service gaps and overlaps do not appear.

Finally, Seacord et al. (2001) propose a process to identify component ensembles that satisfy the specification of system requirements focused on a knowledge basis of the integration rules of the system. This technique does not make use of a trading model, as discussed in the section titled "Trading for COTS Components."

### Building COTS-Based Systems

Another related research line deals with processes aimed at developing software applications with commercial components. First, Seacord et al. (2001) propose some processes for the identification of components, focusing on the knowledge of the integration rules of the system. This proposal lacks a concrete way of documenting commercial components; however, it is one of the few works that deal with real examples of commercial components. This work is developed in the COTS-Based Systems (CBS) Initiative at the Software Engineering Institute (SEI) at Carnegie Mellon University (Pittsburgh, Pennsylvania).

Second, the Concepts to Appliccation in System-Family Engineering (CAFÉ) and Engineering Software Architecture and Platforms System Families (ESAPSb) projects (available at www.esi.es/Cafe and www.esi.es/esaps) can be cited. Although there are several lines of interest, we emphasize the work of Cherki et al. (2001). The authors describe a platform called Thales for the building of software systems based on COTS parts. This proposal also makes use of Rational tools to define the software architecture, and a diagram of classes prevails instead of UML-RT, as carried out in the trader-based development method. This work also lacks trading processes for COTS components.

## CONCLUSION

CBD deals with the building of software systems by searching, selecting, adapting, and integrating COTS software components. Although a software component market has been quite slow to develop, we are perceiving how the effective use of software components is slowly becoming a valid technology for the building of software systems (e.g., Componentsource is successfully selling and licensing many components every month). Moreover, the complexity of the applications is continuously growing, and the amount of information available is becoming too large to be handled by human intermediaries. Therefore, automated trading processes seem to play a critical role in CBD.

In the present chapter, a trading-based development method (TBDM) has been proposed to build software applications as an assembly of COTS software components. This work is due in part to the need of linking three areas of COTS component-based development: (1) the documentation and specification of COTS components, (2) the description of software architectures with COTS components, and (3) the trading processes for COTS components.

To document COTS components, we have first proposed the use of XML document templates (called COTScomponent) for the specification of commercial components. In the case of software architectures, we think that the UML-RT notation is suitable to describe software architectures with COTS components. Finally, we have discussed that the current trading processes are not sufficient to support COTS components, given that there is no connection with existing software architectures. TBDM, an automated method to build COTS systems, solves this problem. It mainly uses two functions (or stages): (1) the COTStrader function, a tool that extends the ODP trading service to look for COTS components, and (2) the COTSconfig function, a tool that generates combinations of components (called configurations) from those in the candidate list—found previously by the COTStrader function. The configurations are taken as solutions that fulfill the architectural requirements of those abstract components defined in the software architecture of the system. The software architecture's description represents an early stage of the TBDM method, which uses the UML-RT notation.

As a future line, metrics and heuristics should be considered so that the trader can generate ordered sequences of candidate components based on certain criteria established by the user or the administrator of the trader community. A connection with other traders should also be implemented to search or register components in a federated trading. Finally, a COTS trader should also be in accordance with some Web services technologies, integrating it with WebServices Definition Language (WSDL) documents, the Universal Description, Discovery, and Integration (UDDI) repositories, and the Semantic Web.

## ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees for their insightful comments and suggestions. This work has been partially supported by Spanish CICYT Projects TIC2002–04309-C02–02 and TIC2002–03968.

## REFERENCES

Bastide, R., O. Sy, and P. Palanque. 1999. Formal specification and prototyping of CORBA Systems. *Lectures Notes in Computer Science* 1628: 474–494.

Canal, C., L. Fuentes, J.M. Troya, and A. Vallecillo. 2000. Extending CORBA interfaces with protocols. *Computer Journal* 44, no. 5: 448–462.

Cheesman, J., and J. Daniels. 2001. *UML Components. A Simple Process for Specifying Component-Based Software.* Boston, MA: Addison-Wesley.

Cherki, S., E. Kaim, N. Farcet, S. Salicki, and D. Exertier. 2001. Development support prototype for system families based on COTS. *TR ESAPS Project,* www.esi.es/esaps.

Chung, L., B. Nixon, E. Yu, and J. Mylopoulos. 1999. *Non-Functional Requirements in Software Engineering.* Boston, MA: Kluwer Academic Publishers.

Dhara, K.K., and G.T. Leavens. 1996. Forcing behavioral subtyping through specification inheritance. In *18th International Conference on Software Engineering,* 258–267. Berlin, Germany. Los Alamitos, CA: IEEE Computer Society Press.

Garlan, D., S. Chen, and A. Kompanek. 2002. *Reconciling the* needs of architectural descriptions with object-modeling notations. *Science of Computer Programming,* 44, no. 1: 23–49.

Goguen, J., D. Nguyen, J. Meseguer, D. Zhang, and V. Berzins. 1996. Software component search. *Journal of Systems Integration* 6, no. 9: 93–134.

Han, J. 1999. Semantic and usage packaging for software components. Paper presented at ECOOP'99 Workshop on Object Interoperability, Lisbon, Portugal

IONA. 2001. ORBacus trader: ORBacus for C++ and Java. *Technical Report,* IONA.

Iribarne, L., J.M. Troya, and A. Vallecillo. 2001. Trading for COTS components in open environments. In *27th Euromicro Conference,* 30–37. Los Alamitos, CA: IEEE Computer Society Press.

———. 2002. Selecting software components with multiple interfaces. In *28th Euromicro Conference,* 26–32. Los Alamitos, CA: IEEE Computer Society Press.

Iribarne, L., A. Vallecillo, C. Alves, and J. Castro. 2001. A non-functional approach for COTS components trading. Paper presented at the *Workshop on Requirements Engineering,* Buenos Aires, Argentina.

ISO/IEC. 1997. Reference model for open distributed processing. ISO/IEC DIS 13235, ITU-T X.9tr, ISO/IEC/ITU-T.

ISO/ITU-T. 1996. Information technology–Open Distributed Processing–ODP trading function. Rec., ISO/IEC DIS 13235, ITU-T X.9tr, ISO.

Leavens, G.T., L. Baker, and C. Ruby. 1999. JML: A notation for detail design. In *Behavioral Specifications of Businesses and Systems,* ed. H. Kilov, B. Rumpe, and I. Simmonds, 175–188. Boston, MA: Kluwer Academic.

Leavens, G.T., and M. Sitaraman. 2000. *Foundations of Component-Based Systems.* Cambridge: Cambridge University Press.

Medvidovic, N., D.S. Rosenblum, J.E. Robbins, and D.F. Redmiles. 2002. Modeling software architectures in the Unified Modeling Language. *ACM Transactions on Software Engineering and Methodology* 11, no. 1: 2–57.

Medvidovic, N., and R.N. Taylor. 2000. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering* 26, no. 1: 70–93.

Merz, M., K. Muller, and W. Lamersdorf. 1994. Service trading and mediation in distributed computing systems. In *14th International Conference on Distributed Computing Systems,* 450–457. Los Alamitos, CA: IEEE Computer Society Press.

Mikhajlova, A. 1999. Ensuring Correctness of Object and Component Systems. PhD diss. Åbo, Finland: Åbo Akademi University.

Mili, H., F. Mili, and A. Mili. 1995. Reusing software: Issues and research directions. *IEEE Transactions on Software Engineering* 21, no. 6: 528–562.

Milner, R. 1993. The polyadic pi-calculus: A tutorial. In *Logic and Algebra of Specification,* ed. F.L. Bauer, W. Brauer and H. Schwichtenberg, 203–246. Berlin, Germany: Springer-Verlag.

Ncube, C., and N. Maiden. 2000. COTS software selection: The need to make tradeoffs between system requirements, architectures and COTS components. Paper presented at the *COTS Continuing Collaborations for Successful COTS Development.* Limerick, Ireland.

Nuseibeh, B. 2001. Weaving together requirements and architectures. *IEEE Computer* 34, 3: 115–117.

OMG. 1999. *The CORBA Component Model,* www.omg.org, Object Management Group.

———. 2001. *A UML Profile for Enterprise Distributed Object Computing V1.0.*, www.omg.org/technology/documents/formal/edoc.htm, Object Management Group.

Robertson, S., and J. Robertson. 1999. *Mastering the Requirements Process.* Harlow, UK: Addison-Wesley.

Rolland, C. 1999. Requirements engineering for COTS based systems. *Information and Software Technology* 41, no. 14: 985–990.

Rosa, N.S., C.F. Alves, P.R.F. Cunha, and J.F.B. Castro. 2001. Using non-functional requirements to select components: A formal approach. Paper presented at *Fourth Iberoamerican on Requirements Engineering and Software Environments.*

Seacord, R.C., D. Mundie, and S. Boonsiri. 2001. K-BACEE: Knowledge-based automated component ensemble evaluation. In *27th Euromicro Conference,* 56–62. Los Alamitos, CA: IEEE Computer Society Press.

Selic, B., and J. Rumbaugh. 1998. Using UML for modeling complex real-time systems, www-106.ibm.com/developerworks/rational/library/139.html.

Szyperski, C. 1998. *Component Software: Beyond Object-Oriented Programming.* Harlow, UK: Addison-Wesley.

Vallecillo, A., J. Hernandez, and J.M. Troya. 1999. Object interoperability. *Lectures Notes in Computer Science* 1743: 1–21.

Vallecillo, A., J. Hernandez, and J.M. Troya. 2000. New issues in object interoperability. *Lectures Notes in Computer Science* 1964: 256–269.

Vasudevan, V., and T. Bannon. 1999. WebTrader: Discovery and programmed access to Web-based services. Paper presented at the *Eighth International WWW Conference,* Toronto, Canada.

Yellin, D.M., and R.E. Strom. 1997. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems* 19, no. 2: 292–333.

Zaremski, A.M., and J.M. Wing. 1995. Signature matching: A tool for using software libraries. *ACM Transactions on Software Engineering and Methodology* 4, no. 2: 146–170.