

A study of the RM-ODP model trading function

Luis Iribarne¹, José M. Troya², and Antonio Vallecillo²

¹ Dpto. Lenguajes y Computación. University of Almería, Spain.

`liribarne@ual.es`

² Dpto. Lenguajes y Ciencias de la Computación. University of Málaga, Spain.

`{troya,av}@lcc.uma.es`

Resumen

Component-Based Software Engineering (CBSE) uses more and more “Commercial off-the-shelf” (COTS) components in early phases of the development of the software applications. Developers of software systems demand the presence of processes for searching and selection software components to fulfill architectural requirements of the future application. This work establishes the basis of a trader implementation for COTS components, focusing on the experience of known trading models, news models of computation, and all the potential that the Internet offers. In accordance with this, the ODP/OMG trading function is analyzed, underlining its limitations and requirements for a COTS component trader on open systems.

1. Introduction

Commercial components—called as COTS—are being used more and more in Component-Based Software Engineering (CBSE) for building software applications [Brereton et al., 1999, Brown and Wallnau, 1999, Ncube and Maiden, 2000]. In fact, this kind of component can have significant advantages when developing software applications, such as to reduce development costs, efforts and time, and increasing the reliability and flexibility of the final product.

Nevertheless, appropriated search and selection processes of COTS components have become the cornerstone of any effective COTS development. Those processes currently face serious limitations, though, generally due to two main reasons. First, the information available about the components is not detailed enough for their effective selection. And second, the search and evaluation criteria are usually too simple to provide practical utility. In the first case, the black-box nature of COTS components hinders the understanding of their internal behavior; besides, only functional properties of components are usually taken into account, whilst other information crucial to component selection is missing, such as protocol or semantic information [Vallecillo et al., 1999], or non-functional requirements [Rosa et al., 2001, Chung et al., 1999]. On the other hand, component searching and matching processes are delegated to traders; but the problem is that current traders do not provide all the functionality required for an effective COTS component trading in open and independently extensible systems (e.g., Internet), as discussed in [Vasudevan and Bannon, 1999].

In this work we analyse the standard ODP trading function and its limitations for COTS components. Likewise, we propose a list of requirements that a COTS trader must fulfill. This requirements have layed the foundations for an implementation of a trader for COTS components called **COTStrader**. At the present time, there is an implementation available at <http://www.cotstrader.com>.

This paper is structured in six sections, the first of one concerns to the present introduction. Section 2 exposes some important characteristics of the ODP trading function, such as *service types* and *service offers*, the *exporter* and *importer* roles, and *trader federation*. Section 3 analyses the limitations that the current ODP trading function presents in the field of the commercial components. These limitations have been classified as functional, extra functional and computational levels. Section 4 proposes a list of requirements to have a COTS trader. Finally, Section 5 covers some conclusions.

2. The ODP trading function

The Reference Model of Open Distributed Processing (RM-ODP) is a model that is being jointly developed by International Standard Organisation (ISO) and International Telecommunication Union (ITU-T). This model defends the transparent use of services distributed in platforms and heterogeneous networks and a dynamic location of these services. The trading function is one of the 24 functions of ISO/ITU-T ODP model [ISO/IEC-ITU/T, 1997]. This specification has been adopted by Object Management Group (OMG) with the name of *CosTrading* for the *CORBAservices* trading service. At the present time, several implementations of the CosTrading service are available in the market [AceORB, 2000, OpenORB, 2001, OOC, 2001, PrismTech, 2001].

2.1. Object roles

From the viewpoint of the object-oriented programming (OOP), a trading function, or simply trader, is a software object that serves as intermediary between objects that supply certain capacities—the services—and other objects that demand a dynamic use of these capacities. From the ODP viewpoint, those objects that supply capacities to other objects are called *exporters*. On the other hand, the objects that demand capacities to the system's objects are called *importers*.

Therefore, as we can see in Figure 1, a trading process client can have one of the following roles [ISO/IEC-ITU/T, 1997]:

- a) The **exporter** role. An object adopts the *exporter* role when it must announce a service to the system's objects. It is achieved by means of the `export()` method of the trader object, setting correctly the parameters to be able to carry out the storing of the service in the trader repository.
- b) The **importer** role. An object adopts the *importer* role when it requires certain services stored in the trader repository. It is achieved by means of the `query()` method of the trader object, establishing the parameters correctly.

Considering the Figure 1 as a diagram of transitions, an interaction sequence could be the following one: In the first place, a trader's client object adopts the **exporter** role because it wants to announce a service in the trader (1). After that (2), another trader's client object requires certain services that can be offered by the trader. This client object adopt the **importer** role, imposing its searching restrictions. As a result (3), the trader returns one or more object reference with the instances of the wished services, which indicate where those services are located within the distributed system. Finally (4), it is a decision of the client object to select one of the object references offered by the trader, and to connect directly to the object that offers the searched service.

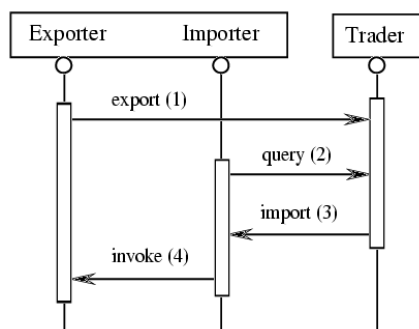


Figura 1: The ODP trading roles

In accordance with this, the applications of trader can be multiple, for example, for reusability of applications (legacy systems) [Mili et al., 1995], for collecting and classification offered capacities on distributed environments [Davidrajuh, 2000], or as a printer server buffer in network [OOC, 2001], among other applications.

2.2. Interfaces

A trader object has different interfaces that allow the interaction with client objects, emphasizing the *Register* and *Lookup* interfaces. The *Register* interface describes an **export** operation that allows the exporting task of services, among other operations, such as removing or modifying an existing service. The *Lookup* interface contains a *query* operation that allows the extracting of service references from the repository.

A trader object works with three kinds of independent repositories: (a) an interface repository *IR*, (b) a repository where the service types of the trader are stored *STR*, and (c) finally, a repository of offers *RO* where the clients register their offers as instances of services from service types in *SRT*. In the three following sections we are going to separately treat these information groups. We will also analyze a case example that illustrates the described behaviour.

A capacity or service exported by a client object must have associate a computational interface that responds to the functionality of the announced service. The trading function specification accepts the definition of interfaces in OMG IDL, and is stored in an independent *interface repository*, *IR*.

2.3. Service types

A trader can support different service types, all of them stored in an independent *service type repository*, *STR*. A trader service can be identified as: $\langle \text{TypeName}, \text{IDL}, \text{Properties} \rangle$, where **TypeName** is the announced service type, **IDL** is the name of the IDL interface that describes the computational signature of the announced service, and **Properties** is a list of properties that collects non-functional information not gathered by the interface of the announced service.

Using the ODP trading specification syntax, a service can be expressed as follows:

```

service <ServiceTypeName>[:<ServiceTypeName>[,<ServiceTypeName>]*]{
  interface <InterfaceTypeName>;
  [[mandatory] [readonly] property <Type> <PropertyName>;]*
}
  
```

In this notation we can observe several characteristics in an ODP trading service. A new service type can be created from service types already supported by trader, supporting multiple inheritance of service supertypes. Besides, a service type also requires the declaration of a single interface name that encapsulates the computational aspects of the capacity of the service to announce. A service can contain one or more properties or attributes that enclose non-functional aspects, which cannot be gathered by the interface. Each property can be considered as a short list: `<name,type,mode>`. In this case, `name` is the name that identifies the declared property, and `type` is the value type that accepts the property, for example a **string**, a **float**, or a **long** type, among other values. Although this information can not be reflected in the specification, most implementations of the ODP trading function use CCM CORBA::`TypeCode` types, such as the ORBacus [OOC, 2001], JavaORB [OpenORB, 2001] or TAO [AceORB, 2000] trading functions. Finally, `mode` represents the access way to the declared property. For example, a **mandatory** access means that a property value must be indicated when an offer of that service type is exported.

2.4. Service offers

A client object can export a capacity or service by means of the `Lookup::export` operation. A client object exports a service announcing an “offer” of service type supported by the trader. Then this offer is registered in an independent *repository of offers*, *RO*. A service offer can be identified as follows: `<TypeName,Location,Properties>`, where `TypeName` is the name of the announced service type, `Location` is the place where the object that implements the announcing service type is running, and `Properties` is a list of properties that fulfill the definition of the `TypeName` service type. In this case, a property is a couple `<Name,Value>`, where `Name` is the name of the property and `Value` is the value of this property.

2.5. An example

Let us suppose that we have a software component that implements different algorithms to translate image formats, such as **gif**, **jpg** or **png** formats. A simple interface for this example could be as follows:

```
module GifImageConverter {
    interface Formats {
        string giftjpg(in string name);
        string giftpng(in string name);
    }
    interface GifImageConverter: Formats {
        enum Format {jpg, png}
        string GifConvert(in string name, in string format);
    }
}
```

In this example, both `giftjpg` and `giftpng` methods use and return a `string` parameter to specify the path name where the image file is allocated.

Following the ODP service type definition syntax, this interface can be registered by trader as follows:

```
service GifConverter {
    interface GifImageConverter;
    mandatory property string otherformat;
    mandatory property boolean colour;
}
```

A service type called `GifConverter` is declared, encapsulating the `GifImageConverter` interface and declaring two additional properties: `otherformat` parameter is used to specify other formats of conversion; `colour` parameter is a boolean value that indicates if the offer accepts images in colour. Finally, client objects announce its particular implementations of `GifConverter` type in the trader. For example, the following represents three offers for the `GifConverter` service, which are announced by three independent clients.

```
offer GifConverter {      offer GifConverter {      offer GifConverter {
  Location object1;      Location object2;      Location object3;
  otherformat="";      otherformat="tif,bmp"  otherformat="bmp";
  colour=false;}      colour=true;}      colour=true;}

```

In this case, `Location` is an object reference that implements the service type supplied in the trader; `object1` accepts the conversion formats on the `GifConverter` service type and it does not support images in color. The other two objects supply implementations that accept additional conversion formats and images in colour.

2.6. Looking for offers

In previous sections we have discussed aspects related to the *exporter* role of an object. In this section we are going to discuss some details for the *importer* role.

The main responsibility of a trading service is to satisfy importer client requests, which are focussed on four issues, `<ServiceType,Constraints,Preference,Policies>`, where:

- (a) `ServiceType`, is the service type of those client offers stored in the trader and involved in the searching process. The searching conditions are established as follows.
- (b) `Constraints`, is a Boolean expression written in OCL language. For example, the expression “`colour==true or 'bmp' otherformat`” means that the trader must locate `GifConverter` offers, supporting colour images and conversion formats to `bmp` files.
- (c) `Preference`, indicates the order in which the offers are returned to the client. The ODP trading service supports five preferences: `max` (ascending), `min` (descendent), `with` (under condition), `random` (random) and `first` (in the natural order). For example, considering the `GifConverter` service type, a preference “`with(color==true)`” means that the trader will return firstly those offers that support images in colour, after evaluate the search with the imposed conditions in *Constraints*.
- (d) `Policies`, are the policies that suit the search. For example, there are policies that limit the cardinality of the offers involved in the trader searching process. Figure 2 shows the sequence of tasks that are achieved in an *importer* role. Here, the client can set different cardinalities to limit the number of offers that must be considered in each step.

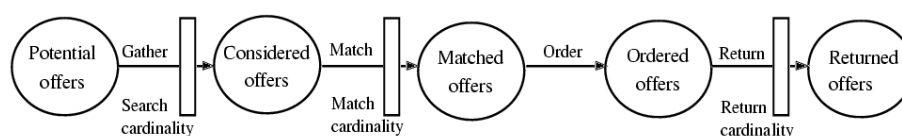


Figura 2: Sequence of tasks in the ODP trader importer role

2.7. Federation of traders

The ODP trading service specification allows to the system administrator to connect his trader with other well-known traders. This allows the propagation of requests in a network. When a importer client executes a query operation, the trader always looks for the offers in its repository of service offers. The trader can also send the request to the connected traders, locating new offers considering the same search conditions imposed on the target trader.

In this federation, a trader object can play the role of a client object over other trader. Furthermore, each trader imposes its internal policies, which can be propagated by the client in the query request. For example, as we can see in Figure 3, one of these policies limits the depth of the search or the number of traders in which the query request can be propagated (`hop_count`). The query policy value decreases in one when the query is propagated from a trader to other. This value must be less or equal to the policy value of the current trader.

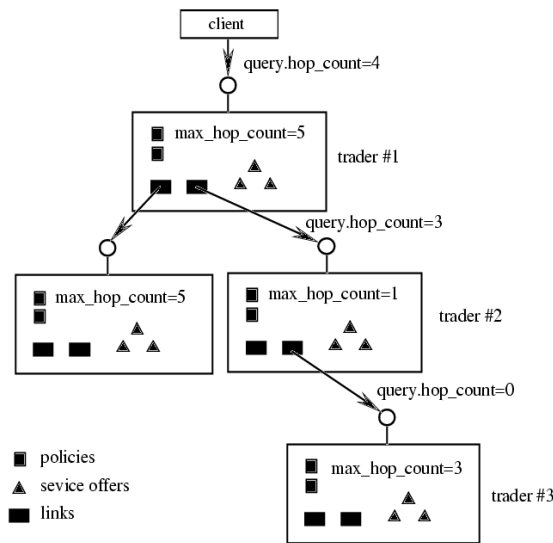


Figura 3: Propagation of a query in a ODP trader federation

3. ODP trader shortcomings

In this section we emphasize some limitations that the current ODP trading function presents for COTS components. These limitations have been detected on the basis of the obtained experience from current implementations of the trading service [AceORB, 2000, OpenORB, 2001, OOC, 2001, PrismTech, 2001] and other related work [Bearman, 1997, Beitz and Bearman, 1997, Merz et al., 1994]. Finally, we have detected a total of ten limitations, which we have cataloged in three classes: functional (limitation 1 to 8), extra-functional (9) and computational (10).

3.1. Shortcoming 1: OMG component model.

The ODP trading function specification uses the OMG component model, and it adopts the OMG Interface Description Language (IDL) to identify the service interfaces of the trader repository. As we have indicated in the introduction of this work, our purpose is to have a trader that supports COTS components for open systems. Therefore, the trading function must support several models of components (CORBA, EJB, .NET, and so on).

3.2. Shortcoming 2: IDL+properties is not enough.

As we have seen in Section 2, a service is identified by means of an interface that contains functional aspects, and a collection of properties that contains non-functional aspects. We think the tandem “IDL+properties” is not enough for COTS components. The importing tasks associated to the ODP trader only allows to set the query constraints on the properties of a trader service, but not on the properties’ IDL information. For example, our viewpoint COTS component model uses an IDL that gathers four types of information: `<functional, non-functional, architectural, non-technical>`. Then a client object should be able to achieve a query to the trader, imposing IDL query restrictions over these types of information.

3.3. Shortcoming 3: partial matching.

As we have seen, the ODP trader limits the constraint operations only on the properties of a service and not only on the interface. This forces that the trading function must achieve exact matchings between requests/offers. Nevertheless, in COTS ambients the trading function also allows to impose query restrictions on the IDL of a service offer, supporting *Partial Matches* [Zaremski and Wing, 1995]. For example, let us consider the `GifConverter` service type, which was covered in Section 2.5. As we have seen, this service type has the `GifImageConverter` interface, which contains two operations, `giftojpg()` and `giftopng()`. To simplify, we are going to define a service as an assembly of operations $C = \{O_1, O_2, \dots, O_n\}$, being O_i a service operation, and n the number of operations that this service includes. Therefore, for the example that we are treating, the `GifConverter` service type can be defined as $GifConverter = \{giftojpg(), giftopng()\}$. A trader with “partial matches” should return service offers of the `GifConverter` type for query requests that include the service $B = \{giftojpg()\}$, and also for query requests that include the service $C = \{giftojpg(), giftopng(), tex2html()\}$.

3.4. Shortcoming 4: behaviour with multiple interfaces.

A COTS component can have multiple interfaces. Although the ODP trading function associates an interface to a service and also it allows inheritance of services, it does not allow a registry of components with multiple services (multiple interfaces). For example, let us suppose that we have a component called `Converter` that translates `tex` file formats to HTML file formats, and it also translates those `gif` image format found in the `TeX` file to `jpg` image format. Let us suppose that this component can be identified by two interfaces, `TeXConverter` and `GifConverter`—such as we have treated in Section 2.2. Then, when a client object wants to announce this component in the trader, it must create a service for the interface `TeXConverter`, and also another service for the `GifConverter` interface. To conclude, a service for the `Converter` must be created, which inherits both previous services.

3.5. Shortcoming 5: one to one matching.

The client/trader interaction is a process that involves one-to-one service matchings. The ODP trading function does not allow to set selection criteria in matching processes to select more than one service. For example, in the previous example, a client object cannot specify query criteria simultaneously on the properties of the `TeXConverter` service type and on the properties of the `GifConverter` service type, and also considering these services as independent services. The client can query the `Converter` service type, which inherits both previous service types. However, this query involves one-to-one matching again. A trading process that support many-to-many matchings could be very useful to have compositional traders.

3.6. Shortcomings 6: the push model is not enough.

The ODP trading function uses a *push* model to store service offers in the repository of offers (*RO*) of the trader. The requests arrive to the trader without having knowledge of how, when and who makes them. However, in a COTS components environment on the Internet, the trading function should support a *pull* model, in which automated processes—in the field of the mobile agents these processes are called *bots*, in relation to robots or spiders—track the network searching COTS component offers. This is a process that follows a push-pull model.

3.7. Shortcoming 7: query based direct federation.

As we have seen in Section 2.7, the ODP trading function allows to connect a trader with others traders, and these also with others one, obtaining an interconnected collection of traders (a federation). Nevertheless, as shows Figure 3, these connections are unidirectionals. If we want that a trader *A* shares information with other trader *B*, and viceversa, a connection from *A* to *B* must be indicated, and another one from *B* to *A*. Furthermore, the current trading function does not take advantage of the federation concept because it does not achieve existence checking on the federation when exporting offers. Therefore, the ODP trading function uses a single import-oriented direct federation model. We thought that a COTS trader should support a similar behavior for the exporting process.

3.8. Shortcoming 8: reactive behavior.

Although in a trader federation each trader has internal policies, it follows a reactive behavior when a query request arrive to the trader. That is, the trader accepts a request, it executes the request, returns the results, and finalizes. However, in COTS component environments, the trader federation should adopt a trader collaborative behaviour.

3.9. Shortcoming 9: Quality of Service.

The ODP trading function does not provide criteria to control the quality of the service (QoS) of the trader information—the service offers achieve by vendor clients—neither to manage this information in repositories. The trader could require that a vendor client establishes an expiry field in the service registry to verify the validity period of the registered trader information. After that, the own client updates this service in trader, using a push model—in case she/he wishes to continue supplying his/her product—or even there could be an automated process that contacts vendors if the expiry field succeeded. If a validity confirmation is not received, then the service should be eliminated from the repository. The trader could also require other additional information, such as security, confidentiality, or market fields, among others.

3.10. Shortcoming 10: Other computational).

The ODP trading function also has several gaps on the COTS component field. The following enumerates some of them:

- (a) By means of the `Register::modify` operation, it is possible to modify a stored service offer in the repository of service offers (*RO*). There is not a similar function to modify a service type, once it has been stored in the repository of service types (*RST*).

- (b) A client stores a service offer in the *RO* using a service type that must exist previously in the *RTS*. In the ODP trading specification it is not clear who, how and when achieves the registry of a service type in the *RTS*.
- (c) The `query()` operation—that is related to the importer action—does not allow query simultaneously the properties of two or more service types.
- (d) In a trader federation does not exist policies that simultaneously affect to all traders of the federation. For example, if we want a cardinality of search of 10, we must indicate this to each trader of the federation.
- (e) When a service is exported, the trader only achieves existence checkings using the service type name, without considering the value on the service type. For example, let $A=\{IDL_A, \mathbf{a}, \mathbf{b}\}$ be a service type with an interface called (IDL_A) and two properties, (\mathbf{a} and \mathbf{b}). Let $A=\{IDL_A, \mathbf{c}, \mathbf{d}, \mathbf{e}\}$ be another service type that already exists in the *RTS*. The trader will consider that these two services are similar—their type names agree—and therefore the registry will not be achieved. Nevertheless, the trader can register a service as $B=\{IDL_A, \mathbf{a}, \mathbf{b}\}$, which is similar to A service.
- (e) There is a set of characteristics, not considered in the ODP trader specification, that must be established at the deployment time or at the activation time, such as the persistence of the trader, or the data update interval (commits) of the repositories.

4. Requirements for a COTS component trader

Considering the previous limitations, in this section we enumerate a list of requirements that a COTS trader for open systems should have. The requirements for a COTS trader are:

- (a) **Heterogeneous component model.** The trader must support components that accomplish with several component models existing in the market, and not only based in the OMG model.
- (b) **Virtual repository based federation.** The trader of COTS components must allow the connection with others traders to carry out component searching and storing operations.
- (c) **Compositional trader.** One-to-many matches must be allowed by a COTS trader, and not only one-to-one matches, similar to ODP trader. A compositional trader can resolve queries which contain restrictions over one or more interconnected components, representing the software architecture of an application that must be built.
- (d) **Multiple interfaces.** An aspect that characterizes COTS components is that they can have multiple interfaces, and therefore, a COTS trader must allow this extension.
- (e) **Partial matchings.** The trader must support partial and exact matchings between the component restrictions of a query request and those availables in the repository.
- (f) **Functional and extra-functional information.** The trader must support searching operations over the interface signatures and over any facilitated extra-functional information, such as non-functional, architectural, and nontechnical information.
- (g) **Metrics and heuristics.** The trader must establish the sorting sequences from the user criteria or the trader administrator criteria.

- (h) **Scalable.** The COTS trader must support component specifications from different vendors, and not only those service specifications supported by the trader.
- (i) **Based on concept.** The trader must support a behavior based on concept, allowing for example component searchings based on keywords, and not only based on properties.
- (j) **Postponing answers.** If the trader does not satisfy a request completely, this must support the possibility of postponing the result until all parts of this request had been covered or a condition had expired (or it had been annulled by the administrator of the trader).
- (k) **Using bots to support a push/pull model.** Additionally to the traditional ODP trader push model, a COTS trader needs of bot and search engine processes that locate COTS components within a determined environment, supporting a pull model that automatically registers COTS components in the trader.
- (l) **Delegation.** If the COTS trader cannot satisfy a part of a query, the trader must be able to delegate the query to one or more traders that solve the request.

5. Conclusions

In this work we have presented a study of the trading function of the reference model of ODP on the basis of the experience that we have obtained from well-known implementations on trading service [AceORB, 2000, OpenORB, 2001, OOC, 2001, PrismTech, 2001] and other works [Bearman, 1997, Beitz and Bearman, 1995, Merz et al., 1994]. As result of this study, we think that the current ODP trading function does not adjust itself to the necessities of a trader for COTS components. At this perspective, we have detected a total of ten limitations of the ODP trading function for COTS components on open systems, and we have cataloged these limitations in three classes: functional, extra-functional and computational limitations. As a conclusion to the work study, we have proposed a list of requirements that a COTS trader should have to work on open systems. This list of requirements has being the base of COTStrader [Iribarne et al., 2001, Iribarne et al., 2002], a trader for COTS components available at <http://www.cotstrader.com>.

Referencias

- [AceORB, 2000] AceORB (2000). The Adaptative Communication Environment (TAO). By Douglas C. Schmidt, University of California. <http://www.cs.wustl.edu/~schmidt/ACE>.
- [Bearman, 1997] Bearman, M. (1997). *Tutorial on ODP Trading Function*. Faculty of Information Sciences Engineering. University of Canberra. Australia.
- [Beitz and Bearman, 1995] Beitz, A. and Bearman, M. (1995). An ODP Trading Service for DCE. In *Second International Workshop on Services in Distributed and Networked Environment*.
- [Brereton et al., 1999] Brereton, P., Budgen, D., Bennet, K., Munro, M., Layzell, P., MaCaulay, L., Griffiths, D., and Stannett, C. (1999). The Future of Software. *Communications of the ACM*, 42(12):78–84.

- [Brown and Wallnau, 1999] Brown, A. W. and Wallnau, K. C. (1999). The Current State of CBSE. *IEEE Software*, 15(5):37–46.
- [Chung et al., 1999] Chung, L., Nixon, B., Yu, E., and Mylopoulos, J. (1999). *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers.
- [Davidrajuh, 2000] Davidrajuh, R. (2000). *Automating Supplier Selection Procedures*. PhD thesis, Narvik Institute of Technology. Norwegian University of Science and Technology.
- [Iribarne et al., 2001] Iribarne, L., Troya, J. M., and Vallecillo, A. (2001). Trading for COTS Components in Open Environments. In *27th Euromicro Conference*, pages 30–37, Warsaw, Poland. IEEE Computer Society Press.
- [Iribarne et al., 2002] Iribarne, L., Troya, J. M., and Vallecillo, A. (2002). Selecting Software Components with Multiple Interfaces. In *28th Euromicro Conference*, Dortmund, Germany. IEEE Computer Society Press.
- [ISO/IEC-ITU/T, 1997] ISO/IEC-ITU/T (1997). Information Technology – Open Distributed Processing – Trading function: Specification. ISO/IEC 13235-1, UIT-T X.950.
- [Merz et al., 1994] Merz, M., Muller, K., and Lamersdorf, W. (1994). Service Trading and Mediation in Distributed Computing Systems. In *14th International Conference on Distributed Computing Systems*, pages 450–457. IEEE Computer Society Press.
- [Mili et al., 1995] Mili, H., Mili, F., and Mili, A. (1995). Reusing Software: Issues and Research Directions. *IEEE Transactions on Software Engineering*, 21(6):528–562.
- [Ncube and Maiden, 2000] 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*.
- [OOC, 2001] OOC (2001). ORBacus Trader. ORBacus for C++ and Java. Technical report, OOC, Object Oriented Concepts, Inc. <http://www.ooc.com/ob>.
- [OpenORB, 2001] OpenORB (2001). The OpenORB web site. Distributed Object Group. <http://www.multimania.com/dogweb>.
- [PrismTech, 2001] PrismTech (2001). Trading Service - White Paper. PrismTech OpenFusion, Enterprise Integration Services. <http://www.prismtotechnologies.com>.
- [Rosa et al., 2001] Rosa, N. S., Alves, C. F., Cunha, P. R. F., and Castro, J. F. B. (2001). Using Non-Functional Requirements to Select Components: A Formal Approach. In *Fourth Iberoamerican on Requirements Engineering and Software Environments (IDEAS'2001)*. San José, Costa Rica.
- [Vallecillo et al., 1999] Vallecillo, A., Hernández, J., and Troya, J. M. (1999). Object Interoperability. In *Object-Oriented Technology: ECOOP'99 Workshop Reader*, number 1743 in LNCS, pages 1–21. Springer-Verlag.
- [Vasudevan and Bannon, 1999] Vasudevan, V. and Bannon, T. (1999). WebTrader: Discovery and Programmed Access to Web-Based Services. In *Poster at the 8th International WWW Conference (WWW8)*, Toronto, Canada. <http://www.objs.com/agility/tech-reports/9812-web-trader-paper/WebTrade%rPaper.html>.

[Zaremski and Wing, 1995] Zaremski, A. M. and Wing, J. M. (1995). Signature Matching: A Tool for Using Software Libraries. *ACM Trans. on Software Engineering and Methodology*, 4(2):146–170.