Towards a Semantically Enhanced Component Trader Architecture

S. Terzis and P. Nixon
Computer Science Department,
University of Dublin, Trinity College
Dublin 2, Ireland
{Sotirios.Terzis, Paddy.Nixon}@cs.tcd.ie

Abstract: Component-oriented software seems promising in providing the dynamism and flexibility necessary to realize the Virtual Enterprise (VE) model for business organisations. According to the VE model, and from the perspective of component-oriented programming, a virtual enterprise system could be constructed by marshalling the necessary services through composition of available components selected from a collection provided by the participating enterprise systems to form the new virtual enterprise system. One of the central issues for component software is the location of components. In large-scale open systems trading is already used as a service location mechanism. This is considered the only way to manage services where complete knowledge of the system is both unreasonable and unrealistic. Providing trading mechanisms appropriate for component development requires a move from appearance based (interface) to behavior based (semantic) trading. We present a semantically enhanced component trader architecture that enables this move, which builds on ideas from component location in software reuse and conceptual structures from knowledge engineering.

1. Introduction

1.1 Motivation

Through the years advances in software have been changing the way in which industry, government and academia work. While at the same time, the emergence of new organizational models, which they made feasible, have been producing new challenges for software development. Under this perspective, it is hardly surprising to see that the emergence of the Virtual Enterprise (VE) model for business organisations produced yet another set of challenges for software development. The VE model is based on the idea that business organisations should be able to marshal more resources than they have available without the need for expansion. It aims to provide the responsiveness that business require in highly changeable market conditions where market opportunities emerge rapidly. Although, a number of technological phenomena and business trends make the VE model both feasible and appealing, the effectiveness and efficiency of the model depends on the development of global information infrastructures. These infrastructures should span organizational boundaries and support flexible, dynamic and seamless composition of services from a worldwide pool.

It seems that the only reasonable way to achieve the flexibility and dynamism required by the VE model is through a new perspective on enterprise system development. The decomposition of current enterprise systems into services will provide the necessary framework within which this new perspective can be built. This framework should provide the necessary flexibility by allowing the dynamic composition and configuration of services. It should also provide the necessary dynamism through on-the-fly exploitation of flexibility by allowing on-the-fly re-composition and reconfiguration of services. Finally, since the number of these components and services will be significant, the framework should also be scalable (see [1] for a more detailed discussion).

Inside this framework the development of a new system is a process of dynamically selecting the appropriate services and composing them into a new configuration. This process presents a number of challenges to traditional development techniques. Traditionally system development is based on the assumption of equally trusted and equally under control system parts. But, this assumption leads to monolithic systems, which are too rigid and unmanageable. Thus, traditional development techniques are unsuitable to support the new framework. From a purely technical perspective, development inside this framework seems to hinge on issues as diverse as service selection through high-level semantic service descriptions, on-the-fly re-configuration, novel programming language structures, and the close integration of trust and trust management into the fabric of applications [2].

1.2 Paper Outline

In this paper we focus on the issue of service selection through high-level semantic service descriptions. We propose the development of a semantically enhanced component trader architecture as a basis for the development framework presented above. The development of the architecture is based on two assumptions; first that the notion of service trading as was introduced in open distributed systems (see section 2.1 below), can provide the basis for a service selection framework. Second that component-oriented software (see section 2.2 below) could provide the basis for the realization of services. Discussion about other aspects of the framework can be found elsewhere [3, 4].

We start in section 2 by examining the scope and the implications of our two basic assumptions through a presentation of the history and the background of dynamic service location and component software. This presentation determines the goals and defines the requirements of the semantically enhanced component trading. Then, in section 3, we present our semantically enhanced trader architecture and we discuss some issues related to its implementation. We also determine our evaluation process for the trader. In section 4 we conclude by presenting the intended contribution of our work and a summary of our findings. Finally, we propose directions for future work.

2. Background

2.1 Dynamic Service Location

According to the new development framework described in section 1.1, the first step in the process of developing a new system is the selection of appropriate services from a wide collection of available ones. The need for a service location facility is a

crucial issue. A similar problem is the location of services in large-scale open distributed systems, which attracted a lot of research interest in the past. A number of different approaches, like name, directory and trading services have been proposed, each of them exhibits certain advantages that make it particularly suitable for specific applications. A look into each of them will help us determine which one is more appropriate as a basis for our new development framework.

A name service allows the location of services based on their names. The advantages of such a technique are first that names are easier to remember than references to the actual services, second it allows the decoupling of service providers and service consumers. But, at the same time the use of names is also its main disadvantage as the size of the name space increases, developers are unable to keep track of the plethora of available names. Adding a structure to the name space could help which led to the idea of a directory service. Structuring the name space makes the location of the appropriate name easier. To make that clearer, think of how problematic would be the search for a phone number in phone book that is not organized alphabetically or under any other scheme. Moreover a directory service is really scalable as the Internet host-naming scheme proves. But, a directory service is still based on the association of names to services. As a consequence in both name and directory service the whole component location process is based on two assumptions, first that the names are intuitive on the nature of the underlined service. Second, the naming scheme deployed in structuring the name space is based on simple principles that make name construction easy. As the number of available services in the system grows, the validity of both assumptions weakens. To make things worse, if available services are not predefined and complete knowledge of the system is impossible, as is the case in open distributed systems; a naming scheme simply does not work.

Trading services were developed to answer this problem. A trading service is based on the notion of matching service offers with service requests. It solves the problems of naming schemes by employing a service typing hierarchy. The service location does not require knowledge of the service name just knowledge of the developer needs (specification of a service type). A trading service seems the only reasonable way to manage the development of large-scale compositional systems, where knowledge of the whole system is both impossible and unrealistic. A good analogy to make the difference between trading and directory services clearer is the phone book. The use of a directory service corresponds to the use of the white pages, while the use of a trading service corresponds to the use of the yellow pages. Since the new development framework described above aims to large-scale systems where the number of available service is significant, trading seems the most appropriate basis for the dynamic service location.

A good definition of trading, which we adopt in this paper is the following: "*the activity of choosing services, such that they match some service requirements. The choice is based on the comparison of the specification of a service required (provided by a prospective consumer) and the service specification supplied by service providers or their agents*" [5]. The ANSA model [6] for open distributed processing introduced the notion of a trading facility [5]. The idea of a trading function was adopted as part of the basic reference model of open distributed processing (ODP), which is an international standard [7, 8]. The Object Management Group (OMG) provides also a specification of a

trading service as part of the object services of its common object request broker architecture (CORBA) [9, 10]. Both the ODP and the OMG trading facilities were greatly influenced by the ANSA specification. The OMG object trading service is certainly the most influential due to the success of CORBA. There are currently a number of commercial implementations available, like the OrbixTrader [11], ORBacus Trader [12], etc. For this reason, the rest of the discussion on trading is based mainly on the OMG object trading service.

2.2 Component Software

Component software, as the term implies, is software that is constructed through the composition of software components [13]. According to the description of the new development framework (see section 1.1), the defining characteristic of services is their ability to be composed and configured into a system making the relation between service and components apparent. Although the componentization of services is not the main focus of this paper, a look into component software will first demonstrate that following a component-based approach presents further advantages than just supporting the new development framework. And second will provide us with a definition of components and some insight on the currently available technology upon which our proposed solution will be based.

Component software emerged as a solution to the “software crisis” problem [14]. The term “software crisis” refers to the fact that software construction remains still a high risk and high cost process. Software development techniques seem unable to provide the necessary productivity increases to keep up with the need for more and higher quality software (see [15] for recent manifestations of the problem). The basic idea behind component software is that software systems should be developed by composing prefabricated components in a way analogous to the way hardware systems are developed by plugging together circuit boards.

The use of prefabricated components has a number of advantages. First, the cost and risk for each particular system drops significantly. The reason for this is that the risk in using prefabricated components is lower since they have been already tested in other systems while the cost of their development is distributed over a number of different projects. The fact that the components are reused from system to system increases confidence in them since errors are more likely to be detected and corrected. As the development cost and time for the components is spread into more than one project, the search for optimal solutions and implementations becomes feasible. Finally, there is strong belief that components are a natural step into maturity for software engineering, in the same way as it was for other engineering disciplines.

Through the years a number of software development techniques aiming to component software have been proposed, examples are modular programming [16] and object-oriented programming [17], but they failed to reach the full potential of component software. The last few year, the success of component-based technologies like ActiveX controls [18], JavaBeans [19] and Enterprise JavaBeans [20] combined with the emergence of distributed object frameworks like CORBA [9] and COM [21, 22], and architectures like Jini [23], stimulated interest in component software [13, 24]. The question of what are the components is at the center of component software research.

There have been a number of different definitions of components (you can find a collection of them in [13]), each with its advantages and disadvantages, focusing on different aspects of software artifacts. In this paper we adopt the following one: "*A software component is the unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties*" [13]. This definition captures all the defining characteristics of component (unit of composition, is subject to composition by third parties, etc) and guarantees that each component could be reused in a meaningful way by people other than its producers. As is expected, this definition has significant consequences in the way that components and trading can be combined.

Before we proceed to see the way components and trading can be combined; we need to look a bit in the way the services of our framework are mapped into components. The framework, presented in section 1.1, requires the decomposition of current systems into a number of services. This system decomposition is done according to different levels of abstraction (see Figure 1). At the basic level the system is a collection of services. Services are functionally cohesive groups of basic components. These basic components form the lowest level of system decomposition and exhibit all the characteristics of the definition presented above. At a higher level, services themselves are combined to form a whole system. This means that services themselves are components at a different abstraction and granularity level, and they also conform to the definition. Finally, although we could consider whole systems themselves as components at a higher abstraction, system integration, level; we limit ourselves to the two lower levels.

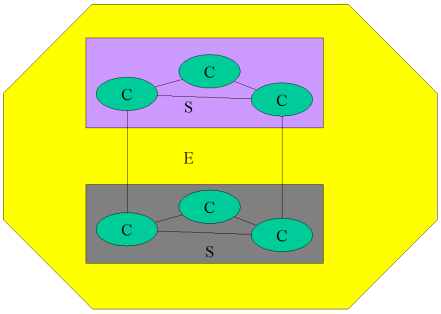

Figure 1: Level of abstraction for system decomposition.

2.3 Component Trading

In order to determine how trading and components can be combined we need to have a closer look into trading and examine what are the implications of the component definition presented in the previous section for the trading process.

2.3.1 A closer look into Trading

Trading, as we mentioned, is based on the notion of matching service offers and service requests. The component responsible for the maintenance of the trading space and the matching of offers and requests is called a trader. Current traders organize services in a service type hierarchy. Each service type defines an interface that everyone using the

service, either service provider or service consumer must follow. They also allow the association of a number of properties as attribute value pairs with each service type. The use of service types and the association of properties for them allow the expression of quite refined service offers and requests in an easy way.

Service providers are called exporters. An exporter exports a service offer by specifying its service type and service properties. The service provider can also refine its offer by specifying a number of service offer properties. Service consumers are called importers. An importer requests a service by specifying the type of the service and constraints on the properties of the type. The importer might also be able to specify a search scope and limit the part of the trading space that wants to be searched. In addition it might be allowed to specify selection criteria for the selection of offers when more than one meets the specified type and property constraints and it could specify the search method that is going to be used by the trader[1].

The combination of the trading policy of the trader, the export policy of the exporter and the import policy of the importer determine the service matching process. Central in the service type matching process is the notion of interface conformance. Interface conformance is based on the "no surprise rule", which states that two interfaces are conformant if components that support one of them can also use the other and no unexpected input or output requirements and termination conditions will emerge. On the other hand, the property matching is based on a lexicographic or arithmetic match of attributes and their values.

Since type conformance depends on the interface, the description of interfaces is crucial in trading. At the basic level the description of an interface is based on determining the signature[2] for every operation of the interface. In most cases this is not sufficient. For example, in the C math library nearly two thirds of the functions have the same signature double → double. In order to overcome this problem the name of the method is considered part of the signature. Since exceptions are valid termination conditions, thrown exceptions are also considered part of an operation's signature. For the specification of operation signatures and interfaces in general interface definition languages (IDLs) have been developed (e.g. CORBA/IDL [25]).

The fact that the "no surprise rule" is at the heart of trading means that for current traders the only relationship between interfaces that can be supported is the supertype subtype relationship. The fact that current trading is IDL-based, means that it is limited only to the appearance of services ignoring their behavior. Subsequently, the only kind of checking that current trader can support is the interface conformance between offers and their respective service type usually supported through an interface repository. In conclusion the above characteristics make current trading syntactic in nature.

2.3.2 Components and Trading

From the discussion in the previous section two are the main issues in combining components and trading. The first issue is the definition of a component description language, which will play a role analogous to that of IDL in current trading. The second issue is then, starting from the component description language to determine the

[1] The ability to specify the search method as part of a request first appeared as part of the ODP trading function specification [8], but is not supported by the OMG object trading service [10].
[2] The signature of an operation is the type of its parameters and of its return value.

component matching process. This matching process should be based on some notion of "component type conformance", which will reflect the substitutability property between conformant components. In attacking both issues the definition of components presented in section 2.2 is crucial.

According to the definition there are three characteristics, which are crucial for components and have significant implications when we try to combine components and trading: (a) "… *explicit context dependencies…*", (b) "…*contractually specified interface…*" and (c) "…*subject to composition by third parties…*". The first two characteristics have implications mainly related to the component description language, while the last one mainly to the matching process. It is worth noting that although this grouping of the characteristics might imply that the two issues are orthogonal, this is not the case. In fact as it will become clear from the following discussion there are a lot of interaction points between the two. In this paper, our analysis focuses mainly on the matching process. A detailed analysis about the implications of the first two characteristics for the component description language can be found in [26], here we just present the conclusions.

The conclusions of [26] were that in order to make context dependencies explicit, the component description language should include besides the interface provided by the component also the interface required by the component[3]. It should also include a description of the produced and consumed events, as well as description of exposed states[4]. Moreover, it should support the expression of relationships between components. There are already component description languages that provide the ability to express all the parts mentioned above, for example [27, 28]. The meaning of the term contractually specified interface depends on the assumed scope of contracts. The no surprise rule offers some kind of contract between interacting parties, no surprise input, output or termination condition will occur. This notion of contract is supported sufficiently by current traders. A definition of contract like the one in contract-based programming [29] is not supported by current traders and as a consequence the support of pre-/post-conditions and invariants becomes a necessity.

The requirement for composition by third parties has significant implications for trading and raises doubts about its suitability, at least in its current form. A third party is a side that does not have any knowledge of both the component's internals and the internal of the component's clients [13]. So, component trading is challenged to locate components whose composition would be meaningful without looking at their internals. As a consequence the component description language must be powerful enough to capture the behaviour encapsulated in the internals of the components. As it is pointed out in [30, 31] interface definition languages, and subsequently current traders, are unable to provide this level of expressive power.

2.3.2.1 Component Retrieval in Software Reuse

There has been significant research in encoding component behaviour in the component retrieval phase of software re-use [32]. The proposed solutions fall into three

[3] The required interface includes the interface definition of all the operations that the component invokes on other components.

[4] In some cases the result of an operation depends on the state that the component was when the operation was invoked. In this case, it is important for the component to make visible this state.

categories [33]: text-based, lexical descriptor based and specification based. Text based use the textual representation of a component as an implicit description of its behaviour, while employing arbitrarily complex string matching expressions to retrieve required components. Although text based solution have low maintenance cost and are easy to introduce, a textual representation does not guarantee sufficient information for the classification and in fact could be misleading. Lexical descriptor based use key phrases, which are constructed from a predefined vocabulary provided by subject experts, to describe what a component is. This technique can be extended to describe a number of different aspects of the component, leading to a technique commonly called multi-faceted classification [34]. The use of key phrases, which are assigned by subject experts, makes the method sounder and more complete. But, the construction of the predefined vocabulary is not a trivial task and there is also an ambiguity on the type of semantics (computational or application ones) that the vocabulary should describe. Finally, specification based use a specification language, whose semantics define the classification and retrieval scheme. In fact, "s*pecification-based retrieval comes closest to achieving full equivalence between what a component is and does and how it is encoded [described]*" [33]. There are a number of specification methodologies, which deploy various degrees of formality, from informal ones [35, 36] to formal ones [37, 38]. Specification based approaches are in general more powerful than both text and lexical descriptor based ones, mainly because of the wide range of formality they offer, making them preferable as a basis for our component trading approach.

2.3.2.2 The Issue of Formality

Since our component trading is based on specification-based classification and retrieval, we need to determine the appropriate level of formality. This involves a trade-off between precision and usability. Formality provides precise, complete and consistent descriptions of components, besides "t*he only way to eliminate ambiguity is to be formal*" [39]. But, the complexity of formal specification languages makes them difficult to use limiting their popularity [40]. Specification matching at a high level of formality has been studied in the past and can provide various kinds of matching [41], while at the same time the high level of formality provides more alternatives even at the signature matching level [42]. The matching process at this level requires support from a theorem prover, like [43]. Theorem-proving techniques are too complex to allow efficient component retrieval in a large component collection [44] and they are can not be fully automated. To make things worse formal specifications induce significant maintenance costs. These costs could be minimised if the specification of components is done before their implementation, usually though this is not the case. So, although a highly formal approach does not seem ideal, some elements of formality are necessary in order to limit ambiguity.

Another way to limit ambiguity is through the use of conceptual structures and knowledge representation techniques [45]. In fact in [46] they combined trading and semantic networks [47] in order to allow the trader to support the cognitive domain of application users and learn new ways to describe services. While in [48] they used a linguistic ontology to allow users to overcome vocabulary inconsistencies in describing and phrasing requests for components. Conceptual structures of particular interest are concept ontologies. An ontology is collection of concepts together with their definition

and a number of relationships between these concepts. The fact that each concept is associated with its definition and relationships with the others can guarantee that there are no misconception hiding in its use as part of a component description [49]. Ontologies are recognised to be particularly suitable in tackling semantic interoperability problems [50, 51] particularly during system integration [52], and are currently used in a number of applications [53, 54]. Finally, another similar approach is to use standard concepts with well-defined meaning. This is the basis for business objects [31, 55, 56].

In conclusion, specification based retrieval techniques are appropriate as the basis for our component trading. The main issue is the level of formality of the specifications. Although, making formal the semantics of the specifications eliminates ambiguity it also makes them difficult to use. In order to balance these conflicting requirements we accept a lower level of formality while at the same time we use ideas from knowledge management and the kind construct [57] in particular to recover some of the lost ground in our fight against ambiguity. The kind is a system-modelling construct, which acts as a syntactic and semantic bridge between types [57]. So, a popular modelling language like UML [36] in combination with a specification technique based on concepts, like the one described in [58] and the kind construct can be used[5]. It should be noted that even if a specification technique based on concepts was not followed at the construction of the system we could use reverse knowledge engineering to extract it [60].

3. Trader Architecture

From the discussion in the previous section we can see that in order for trading to support service selection through high-level semantic service descriptions we need to change its syntactic nature to incorporate semantics. This change requires a move from the appearance of services to their behavior, which is achieved through two steps. The first step is the replacement of interface definitions with component descriptions. Component descriptions capture more behavioral aspects of a service than interface definitions because besides the provided interface they include the required interface, produced and consumed events, exposed states and relationships with other components. The second step is the replacement of service type conformance with specification matching. While service type conformance organizes service types according to interface subtyping, specification matching uses plug-compatibility [41] and behavioral subtyping [61] instead.

These two steps lead to a new kind of trading, which we call semantically enhanced component trading. We call it component trading, because of the use of component descriptions (first step) and semantically enhanced, because of the use of specification matching (second step), which based on semantic information about the components[6]. With the introduction of semantically enhanced component trading we aim to improve both precision and recall during the service selection process. Improvements in precision will come as a result of the increased confidence on the results of the selection process that specification matching offers. While improvements in recall will come as result of a combination of relaxed matches that specification matching supports with a component composition facility that supports various signature matches [42].

[5] It should be noted that although UML is not formal has a formal part the object constraint language [59].
[6] We call it semantically enhanced instead of semantically based because the degree at which we are based on semantics depends on the level of formality we adopt for specification matching.

The overall trader architecture is presented in Figure 2. The trader accepts requests for components and returns the matching component offers. The request processing is supported by a relationship service, which has a role analogous to the service subtyping hierarchy supported by current traders. During request processing the matching components are determined, which can be either simple or composite. A simple component corresponds directly to component offers from the trading space, while a composite one consist of a set of offers for simple components and a composition strategy. The set of offers and the composition strategy are passed to the composition facility to create the composite component. The component offers that form the trading space are analogous to the service offers of current traders. Finally, the role of the component description repository is analogous to the interface repository of current traders, ensuring that component offers comply with the component description associated with them. The use of the component description repository although it provides a higher degree of confidence during trading, is not necessary.

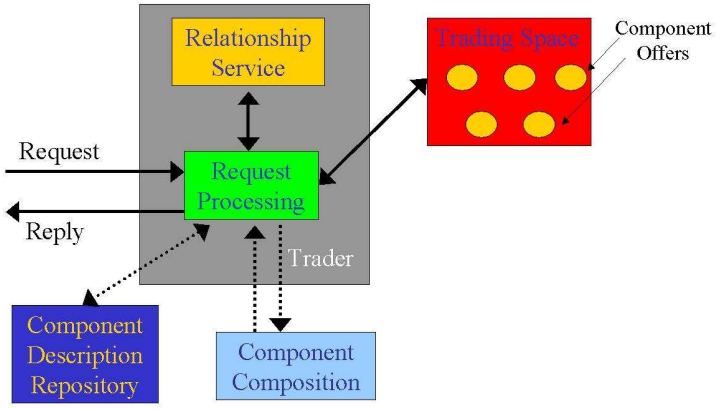

Figure 2: Semantically Enhanced Component Trader Architecture

In the following subsections we give a more detailed presentation and we discuss some implementation issues related to certain parts of the architecture. We close this section with a description of the evaluation process that will be followed to determine the degree at which the architecture achieves its goals.

3.1 Querying Interface

The first part of the architecture that needs further clarification is the querying interface. How are the requests to and the replies from the trader formatted? Current traders organize services in service types and requests are based on the name of the service type. The main advantage of this support is that it makes the matching process a lot more efficient, which is particularly important if the trader is used at runtime. In order to both maintain this advantage and also overcome the strict requirements of runtime trading we introduce two separate querying interfaces, one for runtime and one for development time trading. For each component description we associate a name and queries are based on that name. As we are going to see in following sections, the trader

organizes component offers based on the description that they comply with, while it also maintains a graph of relationships between the various component descriptions. The relationship service of the architecture maintains this graph.

The formation of the replies depends on the querying interface used to submit the request. If the request was submitted using the runtime interface then the trader responds in a way similar to current traders. It either returns a set of valid component references for the matching offers, which can be subsequently used to invoke operations on these components, or it throws an exception to indicate any problems during the matching process. If a reference is returned, then the trader needs to ensure that the "no surprise rule" holds, meaning that returned references comply with the interfaces expected by the requester. As a consequence the matching process during runtime is more restricted. That is particularly true in the case of composite components, because besides the composition strategy the trader needs also matching offers for all the components involved.

If the request was submitted using the development time querying interface, then the "no surprise rule" is relaxed, allowing more flexibility to the trader. The trader returns incomplete composite components in addition to complete simple or composite components. In which case the reply contains the set of component description names, the matching offers for each description name and the composition strategy. This way the developer can identify the missing parts and provide them. In addition, the fact that we are not restricted by the "no surprise rule" means that we can support relaxed matches [41]. In which case, the matched components or the application under development may require adaptations [62]. Finally, at development time the trader could be also used as a tool to inspect the trading space. In which case, related components are also returned.

3.2 Matching Process

It was mentioned above that the matching process depends on the relationship service, which holds a graph with the relationships between the various component descriptions in the system. During the matching process the description name that was requested (see section 3.1) is used to locate the corresponding node of the graph. Then, all the appropriate links from that node are followed in order to determine other possible matches. This traversal of the graph is exhaustive.

The graph consists of a set of nodes, which in order to accommodate both simple and composite component matching, are divided into simple and composite respectively, and a set of edges, which in order to accommodate the two querying interfaces are divided into matching and related links. Every node contains the name of the corresponding component description associated with it. If a component description repository is employed then every node contains also a reference in the repository for the corresponding description. Simple nodes contain also the set of component offers that comply with the description, while composite nodes contain a composition strategy and the set of simple nodes that it uses.

Nodes are connected to each other with links[7]. Two nodes are connected with a matching link if their component descriptions are either equivalent, in which case the link is bi-directional, or a behavioral supertype – subtype relationship holds between them, or finally the two descriptions are plug-compatible. In both of the last two cases the link is unidirectional indicating the direction of the corresponding relationship. Consequently,

[7] Note though that the graph does not have to be fully connected.

matching links represent a strong relationship and are the ones that are followed during runtime querying. Two nodes are connected with a related link if their corresponding component descriptions are conceptually related; both are of the same kind [57]. Related links are always bi-directional and they are followed only during development time querying.

Since the trading process depends completely on the graph, its construction should guarantee the semantics of matching. At first for each new component offer we have to ensure that it complies with the component description of the node it is associated with. This requires first retrieving the component description from the repository and then inspecting the sources of the component to determine its compliance. The way this problem is handled in current traders is by keeping with each service object a link to the interface repository for its interface definition. This approach requires that the trading and the component development are parts of a unified framework. If trading and component development are separate, then the simplest solution is to assume that the offer complies with the description, which the approach that current traders follow when the interface repository is not accessible. A more advanced solution is to use validity evidence as part of the component offer [63], which could include a formal proof of correctness, a comprehensive test suite, or references to other people who have used it successfully. Similar approaches can be followed in dealing with issues of how to guarantee that the composition strategy of composite nodes is meaningful and that the links are valid. In general as we saw in section 2.3.2.2, these issues could be dealt at a middle level, between the simple assumption and the formal proof, by employing information extracted from the design of components, e.g. a UML diagram with its corresponding OCL expressions.

In a more advanced approach the construction of composite nodes and edges could be automated. Automation can be achieved, by employing extensively either automated proofing techniques or reasoning techniques from knowledge representation. For the automated construction of composite nodes the work on software architectural styles [64, 65, 66] could provide guidelines for composition strategies. Although this approach is particularly interesting and will provide a very powerful trading facility, it is a very big issue and requires separate investigation. So, we leave it out from our focus for the moment, but we consider it for future research.

3.3 Component Composition

In general component composition is another very big issue, which requires extensive investigation on its own. Although in this work we are not going to look into component composition in detail, we need to provide some basic component composition functionality in order to support the composite nodes described in section 3.2. As we saw the composite nodes include a set of component offers and a composition strategy. The component composition facility should use the composition strategy and the component offers to produce a composite component [67]. Two are the issues, first how do we construct the composite component and second how we describe the composition strategy.

The composition strategy requires the definition of a composition language that allows the description of the composition process. The various software architectural styles could provide guidelines for possible kinds of compositions. The more kinds of

compositions supported the more complex the composition language is. Since, component composition is not our main focus the architecture will support composition based on the pipeline architectural style [65] and on the techniques used in dealing with object interoperability problems [68]. At a later advanced stage a modular composition language will be supported. The Vanilla framework will provide the necessary support [3]. Finally, the underling mechanism for the composition will be based on techniques like Dynamic Skeleton and Interface Invocation, which allow invocation and service of calls without knowledge of the calling interface supported by the other side. They are part of the CORBA specification [9].

3.4 Trader Evaluation Process

In section 3 we determined as a goal for our architecture to improve both recall and precision in the retrieval of components. The improved precision will come as a result of the higher certainty in the matching process that the use of the component description graph provides. While the improved recall will come as a result of the introduction of composite nodes and the composition facility. The evaluation process should mainly focus in demonstrating the validity of these two hypotheses.

Demonstrating the validity of these two hypotheses through a formal proof or quantitative study is not feasible because we do not have a formal theory to support our architecture and anyway we do not follow a strictly formal approach for the reasons we discussed in section 2.3.2.2. Besides we do not have available a large collection of components to measure precision and recall. So, the demonstration will be based in examples. We will try to show through selected examples that our trader architecture first finds matches that current traders do not. And second that it picks up inconsistencies between component descriptions and eliminates the corresponding matches while current traders do not. For the first part examples in the lines of the one presented in the foundation of the kind construct [57] should be developed and tested. While for the second part abstract data types examples seem to dominate the relevant literature e.g. [41, 61]. An enterprise system application scenario will demonstrate the advantages of our trader architecture in service selection through high-level semantic service descriptions, which was our initial motivation. Finally, some study of the performance of the trader should provide indications on the practicality of the architecture.

4. Conclusions and Future Work

From the above discussion we identify contributions of this work in a number of different areas. First the completion of this work will provide a basic implementation of the semantically enhanced component trader architecture. This trader could be used in the development of applications, allowing developers to take advantage of its improved component retrieval capabilities. Second, it will provide an understanding of how existing notions like trading, specification matching and conceptual construct can be combined together and applied inside a component-oriented development. Finally, a framework for the study of a variety of issues related to component-oriented development through the modular design of the trader will be available. These issues include, but are not restricted to, component description languages, various component matches and component composition.

In conclusion, we contend that the future of enterprise system development requires a new perspective on software development, according which the construction of new enterprise systems is a process of dynamically selecting the appropriate services and composing them into a new configuration. A combination of trading with component software provides the basis for this new perspective. But, this combination requires from trading a move from the appearance to the behavior and leads to a new kind of trading; a semantically enhanced component trading. Semantically enhanced component trading improves both precision and recall in component retrieval. It improves precision by employing semantic information in component specifications either formally or through the use of conceptual constructs. While it improves recall through a component composition facility.

This work will be extended in various ways in the future. First, a number of advanced features that were mentioned above, but are not included in this work could be studied and implemented. These features include support for automatic link and composite node creation. Another feature that could be added is a query by example mechanism. As it is pointed out in [69] the use of names, service type names in current trading, component description names in semantically enhanced component trading, as the basis for querying in current trader can be a problem when the number of the names becomes high. Allowing queries to provide a component description of their own, which the trader will try to match it to the descriptions it already has can eliminate this problem. The addition of such a feature will make the trader a more powerful software development tool. Another feature that could be added is a browser for the component description repository, which will expose the component description graph of the trader to the developers allowing them to manipulate it directly. Finally, use the trader as a framework for the study of the various aspects of component-oriented development. This requires a highly modular trader implementation, which is another issue for future development.

5. References

1. S. Terzis, P. Nixon, V. Wade, S. Dobson and J. Fuller, “The Future of Enterprise Groupware Applications”, in proceedings of the 1st International Conference on Enterprise Information Systems (ICEIS’99), Setubal, Portugal, March 1999.
2. P. Nixon, V. Wade, S. Dobson, J. Fuller and S. Terzis, “Designing Components for a Virtual Organization: A Case Study”, in proceedings of the Object, Components and the Virtual Enterprise Workshop of the International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA’98), Vancouver, Canada, October 1998.
3. S. Dobson, P. Nixon, V. Wade, S. Terzis and J. Fuller, “Vanilla: an open language framework”, in proceedings of the 1st International Symposium on Generative and Component-Based Software Engineering (GCSE 99), Erfurt, Germany, September 1999.
4. P. Nixon, V. Wade, S. Terzis, M. O’Connell and S. Dobson, “The Virtues Architecture: A Software Infrastructure for Business-to-Business E-Commerce”, Virtues Project Technical Report, February 2000.
5. J. Deschrevel, “The ANSA Model for Trading and Federation”, Architecture Report APM. 1005.01, Approved 15 July 1993.

6. R. Van der Linden, “An Overview of ANSA”, Architecture Report APM. 1000.01, Approved 15 July 1993.
7. Open Distributed Processing, “Reference model: Overview”, ISO/IEC 10746-1:1998.
8. Open Distributed Processing, “Trading function: Specification”, ISO/IEC 13235-1:1998
9. Object Management Group (OMG), “CORBA/IIOP 2.3.1 Specification”, October 1999, available at http://www.omg.org/corba/corbaiiop.html
10. Object Management Group (OMG), “CORBAservices: Common Object Services Specification”, December 1998, available at http://www.omg.org/corba/sectran1.html.
11. IONA Technologies, “OrbixTrader - User’s Guide”, 1997, information available at http://www.iona.com/products/orbixenter/trader/index.html.
12. Object Oriented Concepts, “ORBacus Trader version 1.0.2”, 1999, information available at http://www.ooc.com/trader/.
13. C. Szyperski, “Component Software: Beyond Object-Oriented Programming”, ACM Press, Addison-Wesley, 1998.
14. M.D. McIlroy, “Mass produced software components”, P. Naur and B. Randell (Eds.) Proceedings of the NATO Conference on Software Engineering, Garmish, Germany, October 1968, NATO Science Committee, Brussels (published as a book in 1976).
15. N. Gross, M. Stepanek, O. Port and J. Carey, “Software Hell”, Business Week – European Edition, December 6 1999, available online at http://www.businessweek.com/datedtoc/1999/9949.htm
16. D.L. Parnas, P.C. Clements and D.M. Weiss, “The modular structure of complex systems”, IEEE Transactions on Software Engineering, vol. 11, no. 3, March 1985.
17. B.J. Cox, “Planning the software industrial revolution”, IEEE Software, vol. 7, no. 6, November 1990.
18. Microsoft, “ActiveX Controls”, available at http://www.microsoft.com/com/tech/activex.asp.
19. Sun Microsystems, “JavaBeans”, available at http://java.sun.com/beans.
20. A. Thomas, “Enterprise JavaBeans Technology - Server Component Model for the Java Platform”, Patricia Seybold Group, 1998, available at http://java.sun.com/products/ejb/white_paper.html.
21. Microsoft, “Microsoft Component Services: Server Operating System, A Technology Overview”, available at http://www.microsoft.com/com/wpaper/compsvcs.asp
22. Microsoft, “Component Object Model – COM”, available at http://www.microsoft.com/com/tech/com.asp.
23. K. Arnold, B. O’Sullivan, R.W. Scheifler, J. Waldo, A. Wollrath, “The Jini Specification”, The Jini Technology Series, Reading, Massachusetts, Addison-Wesley, 1999.
24. O. Nierstrasz, D. Tsichritzis (ed.), “Object-Oriented Software Composition”, The Object-Oriented Series, Prentice Hall, 1995.
25. J. Siegel, “CORBA fundamentals and programming”, Wiley 1996.
26. S. Terzis and P. Nixon, “Component Trading: The basis for a Component-Oriented Development Framework”, in J. Bosch, C. Szyperski and W. Weck (Eds.) Proceedings of the 4th International Workshop on Component-Oriented Programming, Research Report 17/99, Dept. of Computer Science and Software Engineering, University of Karlskrona/Ronneby.

27. Object Management Group (OMG), "CORBA 3.0 New Components Chapters – CORBA Component Model FTF Draft ptc/99-10-04", available at http://www.omg.org/cgi-bin/doc?ptc/99-10-04.
28. J.R. Kiniry, "CDL: A Component Description Language", presented to the Advanced Topics Workshop (ATW) of the 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'99), March 1999, available at http://www.cs.caltech.edu/~kiniry/projects/papers/kiniry/COOTS99_ATW/cdl/cdl.html
29. B. Meyer, "Applying Design by Contract", IEEE Computer, vol. 25, no. 10, October 1992.
30. J.H. Gennari, A.R. Stein and M.A. Musen, "Reuse For Knowledge-Based Systems and CORBA Components", in proceedings of the 10th Knowledge Acquisition for Knowledge-Based Systems Workshop (KAW'96), Banff Canada, November 1996, available at http://ksi.cpsc.ucalcary.ca/KAW/KAW96/gennari/
31. T. Digre, "Business Object Component Architecture", IEEE Software, vol. 15, no. 5, September/October 1998.
32. R. Rada, "Software Reuse", Intellect, 1995.
33. H. Mili, F. Mili and A. Mili, "Reusing Software: Issues and Research Directions", IEEE Transactions On Software Engineering, vol. 21, no. 6, pp. 528 – 562, June 1995.
34. R. Prieto-Diaz and P. Freeman, "Classifying Software for Reusability", IEEE Software, vol. 4, no. 1, January 1987.
35. R.J. Wirfs-Brock and R.E. Johnson, "Surveying Current Research in Object-Oriented Design", Communications of the ACM, vol. 33, no. 9, September 1990.
36. G. Booch, J. Rambaugh and I. Jacobson, "Unified Modeling Language User Guide", Addison-Wesley, 1997.
37. B. Potter, J. Sinclair and D. Till, "An Introduction to Formal Specification and Z", Prentice-Hall, 1991.
38. J.C. Wing, E. Rollins and A.M. Zaremski, "Thoughts on a Larch/ML and a new application for LP", in proceedings of the 1st International Workshop on Larch, 1993.
39. R. Wieringa, "A survey of Structured and Object-Oriented Software Specification Methods and Techniques", ACM Computing Surveys, vol. 30, no. 4, December 1998.
40. J.R Kiniry, "The Specification of Dynamic Distributed Component Systems", M.Sc. Thesis, Department of Computer Science, California Institute of Technology, also Technical Report CS-TR-98-08, July 1998, available at ftp://ftp.cs.caltech.edu/tr/cs-tr98-08.ps.Z
41. A. Moormann Zaremski and J.M. Wing, "Specification matching of software components", in proceedings of the 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering, October 1995.
42. A. Moorman Zaremski and J.M. Wing, "Signature Matching: A Tool for Using Software Libraries", ACM Transactions on Software Engineering and Methodology, vol. 4, no. 2, April 1995.
43. S.J. Garland and J.V. Guttag, "A guide to LP, the Larch Prover", Report 82, DEC Systems Research Center, Palo Alto, California, December 1991.
44. J. Penix and P. Alexander, "Efficient Specification-Based Component Retrieval", Accepted to Automated Software Engineering, June 1997, Revised August 1997, originally submitted July 1996, available at http://ic-www.arc.nasa.gov/ic/projects/amphion/people/penix/research/rebound/

45. J. Sowa, “Conceptual Structures: information processing in mind and machine”, Addison-Wesley, 1984.
46. A. Puder, S. Markwitz and F. Gudermann, “Service Trading Using Conceptual Structures”, in proceedings of the 3rd International Conference on Conceptual Structures (ICCS’95), Santa Cruz, California, August 1995.
47. F. Lehmann, “Semantic Networks”, Computers and Mathematics with Applications, vol. 23, no. 2-5, June 1992.
48. S. Borgo, N. Guarino, C. Masolo and G. Vetere, “Using a Large Linguistic Ontology for Internet-Based Retrieval of Object-Oriented Components”, in proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering (SEKE’97), Madrid Spain, June 1997.
49. M. Ushold and M. Gruninger, “Ontologies: principles, methods and applications”, The knowledge Engineering Review, vol. 11, no. 2, June 1996.
50. S. Terzis and P. Nixon, “Semantic Trading: Tackling Interoperability Problems during System Integration”, A. Vallecillo, J. Hernandez and J.M. Troya (eds.), “Object Interoperability”, Dept. of Languages and Computer Science, University of Malaga.
51. S. Heiler, “Semantic Interoperability”, ACM Computing Surveys, vol. 27, no. 2, June 1995.
52. C. Schlenoff, R. Ivester and A. Knutilla, “A robust process ontology for manufacturing systems integration”, February 1999, available at http://www.ontology.org/main/papers/psl.html.
53. H. Smith and K. Poulter, “The Role of Shared Ontology in XML-Based Trading Applications”, February 1999, available at http://www.ontology.org/main/papers/cacm-agents99.html.
54. B. Chandrasekaran, J. R. Josephson and V. R. Benjamins, “What Are Ontologies, and Why Do We Need Them?”, IEEE Intelligent Systems, vol. 14, no. 1, January/February 1999.
55. O. Sims, “Business Objects: delivering co-operative objects for client-server”, IBM McGraw-Hill Series, McGraw-Hill, 1994.
56. C. Casanave, “Business-Object Architectures and Standards”, January 1999, available at http://www.dataaccess.com/dat/BOPaper.htm.
57. J.R Kiniry, “A New Construct for Systems Modeling and Theory: The Kind”, Technical Report CS-TR-98-14, Computer Science Department, California Institute of Technology, October 1998, available at ftp://ftp.cs.caltech.edu/tr/cs-tr98-14.ps.Z
58. J. Martin and J.J. Odell, “Object-Oriented Methods: A Foundation”, UML edition, Prentice-Hall, 1998.
59. J.B Warmer and A.G. Kleppe, “The Object Constraint Language: precise modeling with UML”, Addison-Wesley, Reading, Massachusetts, 1999.
60. P. Devanbu, R. Brachman, P. Selfridge and B. Ballard, “LaSSIE: A Knowledge-based Software Information System”, Communication of the ACM, vol. 34, no. 5, May 1991.
61. B.H. Liskov and J.M. Wing, “ A behavioral notion of subtyping”, ACM Transactions on Programming and Systems, vol. 16, no. 6, November 1991.
62. J. Bosch, “Adapting Object-Oriented Components”, in Proceeding of the 2nd International Workshop on Component-Oriented Programming (WCOP’97),

Jyvaskyla, Finland, September 1997, ed. W. Weck, J. Bosch and C. Szyperski, Turku Center for Computer Science.
63. M. Chandy, P. Sivilotti and J. Kiniry, “A Cottage Industry of Software Publishing: Implications for Theories of Composition”, in proceedings of the 3rd International Workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA'98), Orlando, USA, April 1998, available at http://www.infospheres.caltech.edu/papers/fmppta98/paper.html.
64. D. Garlan and M. Shaw, “An Introduction to Software Architecture”, in V. Ambriola and G. Tortora (ed.), Advances in Software Engineering and Knowledge Engineering, Series on Software Engineering and Knowledge Engineering, vol. 2, World Scientific Publishing Company, 1993.
65. M. Shaw, “Some Patterns for Software Architecture”, 2nd Annual Conference on Pattern Languages of Programming, September 1995, available at http://www.cs.cmu.edu/afs/cs/project/vit/www/paper_abstracts/PLoP95.html.
66. M. Shaw and P. Clements, “A Field Guide to Boxology: Preliminary Classification of Architecture Styles for Software Systems”, in proceedings of the 1st International Computer Software and Applications Conference (COMPSAC97), August 1997.
67. O. Nierstrasz, D. Tsichritzis, V. de Mey, and M. Stadelmann, “Objects + Scripts = Applications”, in proceedings of the Esprit 1991 Conference, Dordrecht Germany, 1991.
68. D. Konstantas, “Object Oriented Interoperability”, in Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP’93), Kaiserslauten, Germany, July 1993, also in “Visual Objects”, ed. D. Tsichritzis, CUI, University of Geneva, 1993.
69. A. Cox, “An Exploration of the Reuse Techniques to the Location of Services in a Distributed Environment”, M.Sc. Thesis, Dept. of Computer Science, Trinity College Dublin, September 1994.