Component Location and the Role of Trading in Large Scale Distributed Systems

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 software is a promising approach for dealing with the problem of large scale distributed system development, because it decreases development costs and increases software quality. One of the central issues for component software in distributed systems is the location of components. In large scale distributed systems trading is already used as a service location mechanism. This is considered the only way to manage services in large scale distributed systems 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 behaviour based (semantic) trading. We present a semantically enhanced component trading architecture that enables this move.

1. Introduction

The last few years have placed component software and component-oriented development issues at the centre of research for both academia and industry [1, 2, 3, 4]. The reason for this is that despite of continuous progress in software engineering, software construction remains still a high risk and high cost process [6]. Software development techniques seem unable to provide the necessary productivity increases to keep up with the need for more and higher quality software. Set along side, distributed systems are considered more difficult to develop, because of the issues that their distributed nature introduces. Thus, it is not surprising that in large scale distributed systems the problem is exasperated. This problem is often referred to as "software crisis" and since its identification variations on the component solution have been proposed [5].

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. In addition, the fact that components are reused from system to system increases confidence in them since errors are more likely to be detected and corrected. Moreover, 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 in maturity for software engineering, in the same way as it was for other engineering disciplines.

Modular programming and object-oriented programming are early attempts to provide a component approach for software development. These techniques although they contributed in dealing with the software crisis problem, they failed to realise the full potential of component software. But the success of technologies like ActiveX controls [7], JavaBeans [8] and Enterprise JavaBeans [9] combined with the emergence of distributed object frameworks like CORBA [10] and COM [11], and architectures like Jini [12], stimulated interest in component software [1, 2, 4]. Despite this interest most issues (e.g. component description languages, component location and composition mechanisms) are still open for component software.

This paper focuses on the issue of component location in large scale distributed systems. In particular it examines what is the role of trading; a technique already in use in large scale distributed systems, as a component location mechanism. It starts by a short presentation of components and trading and how they can be combined, which leads to the realisation of the need for semantically enhanced component trading. Then, it looks into the closely related area of component location in software reuse in order to determine how a semantically enhanced component trader architecture should be constructed. It continues to present the architecture and discuss some

0-7695-0634-8/00 $10.00 © 2000 IEEE

issues related to its implementation. Finally, it concludes and gives some points for further investigation.

2. Components and Trading

In order to determine how components and trading can be combined we need to define what components and trading are. A good definition of trading 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*" [13]. The main advantage of trading is that it does not require knowledge of the services available in the system as naming or directory services do. It just requires knowledge of the developer needs (specification of a service type). Thus, a trading service seems the only way to manage the development of large-scale compositional systems, where knowledge of the whole system is both impossible and unrealistic. Trading was first introduced in the ANSA model [13] for open distributed processing, and was adopted as part of the international standard for a basic reference model of open distributed processing (ODP) [14]. To date the OMG object trading service [15] is certainly the most influential due to the success of the common object request broker architecture (CORBA) [10]. For this reason, the rest of the discussion on trading is based mainly on the OMG object trading service.

A good definition of components is the following: "*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*" [1]. This definition captures all the defining characteristics of components (unit of composition, is subject to composition, etc) and also guarantees that each component can be reused in a meaningful way by people other than its producers (explicit context dependencies, composition by third parties). As is expected, this definition has significant consequences in the way that components and trading can be combined.

According to the above definitions, three are the characteristics of components which have significant implications for trading: (a) "*… explicit context dependencies…*", (b) "*…contractually specified interface…*" and (c) "*…subject to composition by third parties…*". These characteristics have implication both for the way components are described (component description language) and the way components are matched (trading matching process). A detailed analysis of these implications can be found in [16], here we just outline the conclusions.

The first conclusion was 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, which includes the interface definition of all the operations that the component invokes on other components. Additionally, it should include the description of the produced and consumed events, as well as description of exposed states. It is important to make states visible, because in some cases the behaviour of a component depends on the state that the component was, when the operation was invoked. Moreover, it should support the expression of relationships between components. There are already component description languages under development, which provide the ability to express all the parts mentioned above, for example [17, 18]. Second, the meaning of the term contractually specified interface depends on the assumed scope for contracts. At one level, the "no surprise rule", which current traders abide to, offers some kind of contract between interacting parties, ensuring that no surprise input, output or termination condition occurs. At a different level current traders do not support a definition of contract like the one in contract-based programming [19], which requires the support of pre-/post-conditions and invariants. This observation in combination with the "composition by third parties" characteristic of components raises the issue of semantics in trading. Currently, trading is syntactic in nature; it focuses only to the appearance (interface) of component and ignores the behaviour (semantics). So, the main conclusion in [16] is that incorporation of semantics in trading is required, in order to move towards component trading.

2.1. Component retrieval in software reuse

The issue of how to describe the behaviour of components has been under investigation in software reuse, since it is essential for the component retrieval phase. Proposed solutions fall into three categories [20]: text-based, lexical descriptor based and specification based. Text based ones 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 solutions 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 ones use key phrases, which are constructed from a predefined vocabulary provided by subject experts, to describe the component. This technique can be extended to describe a number of different aspects of the component, leading to a technique commonly called multi-faceted classification. 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 a non-trivial task and there is also ambiguity associated with the type of semantics (computational or application ones) that the vocabulary should describe. Finally, specification based ones 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]*" [20]. There are a number of specification methodologies ranging from informal [21, 22] to formal [23, 24] ones. 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 component trading.

Selecting the appropriate level of formality involves a trade-off between precision and usability. Formality provides precise, complete and consistent descriptions of components; "t*he only way to eliminate ambiguity is to be formal*" [25]. But the complexity of formal specification languages makes them difficult to use limiting their popularity [26]. Specification matching at a high level of formality has been studied in the past and can provide various kinds of matching [27], while at the same time the high level of formality provides more alternatives even at the signature matching level [28]. The matching process at this level requires support from a theorem prover, like. Theorem-proving techniques are too complex to allow efficient component retrieval in a large component collection [29] and they can not be fully automated. To make things worse formal specifications introduce significant maintenance costs. These costs could be minimised if the specification of components is done before their implementation, but usually this is not the case. So, although a formal approach seems ideal in limiting ambiguity, because of the reasons discussed above it is not currently very popular.

The use of conceptual structures and knowledge representation techniques can also be used to limit ambiguity [30]. In fact in [31] they combined trading and semantic networks in order to allow the trader to support the cognitive domain of application users, through learning of new ways to describe services. While in [32] 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 a collection of concepts together with their definition and a number of relationships between them. The fact that each concept is associated with its definition and its relationships with the others can guarantee that there are no misconceptions hiding [33]. Ontologies are recognised to be particularly suitable in tackling semantic interoperability problems [34], particularly during system integration [35]. Another similar approach is to use standard concepts with well-defined meaning, which is the basis for business objects [36].

In conclusion, specification based retrieval techniques seem appropriate as the basis for our component trading. Although, making formal the semantics of the specifications eliminates ambiguity it also deters developers from using them. Ideas from knowledge representation techniques can also be deployed to limit ambiguity. The work on kind theory provides a system-modelling construct, which acts as a syntactic and semantic bridge between types [37]. So, a popular modelling language like UML [22] (although it is not formal has a formal part the object constraint language) extended with the kind construct in combination with a specification technique at a conceptual level, like the one described in [38] seem to be a reasonable balance. 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 [39].

3. Trader architecture

From the above discussion we can identify two steps that need to be taken to support component location through trading. The first step is the replacement of interface definitions with component descriptions, which capture more behavioural aspects of a component than mere interface definitions. The second step is the replacement of service type conformance, on which current trading is based, with specification matching. While service type conformance organises service types according to interface subtyping, specification matching uses plug-compatibility [27] and behavioural subtyping [40] instead.

These two steps lead to a new kind of trading, which we call semantically enhanced component trading. We call it semantically enhanced, because it incorporates semantic information about the component through the use of specification matching. 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.

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 a result of the combination of relaxed matches that specification matching supports with a component composition facility that supports various signature matches [28].

The overall trading architecture is presented in Figure 1. 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.

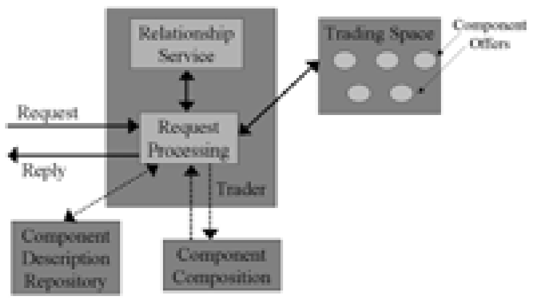

Figure 1. Semantically enhanced component trading architecture

3.1. Querying interface

One of the aspects of the architecture presented in Figure 1 that needs further clarification is the querying interface. This involves the formatting of requests to, and replies from, the trader. Current traders organise services in service types and requests are based on the name of the service type. The main advantage of this 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. The trader organises component offers based on the description that they comply with, while it also maintains a graph of relationships between the various component descriptions, which is maintained by the relationship service of the architecture.

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 ensures that the “no surprise rule” holds, meaning that returned references comply with the interfaces expected by the requester. As a consequence the matching process using the runtime interface is more restricted. This 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 offers and deal with them. In addition, the fact that we are not restricted by the “no surprise rule” means that we can support relaxed matches [27]. In which case, the matched components or the application under development may require adaptations. Finally, the development time querying interface can be also used to inspect the trading space, in which case, related components are also returned.

3.2. Matching process

In the description of the architecture it was mentioned 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 is used to locate the corresponding node of the graph. From each node all the other nodes that are connected to it are somehow related. So, an exhaustive traversal of the graph following all the appropriate links from that node determines all possible matches. The request could limit the search scope (the part of the graph that is traversed) by providing a limit on the number of matches or on the number of nodes considered, which is similar to current trading.

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. Additionally, if a component description repository is employed then every node contains also a reference in the repository for the corresponding description. Moreover, 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, the graph, though, does not have to be fully connected. Two nodes are connected with a bi-directional matching link if their component descriptions are equivalent or a unidirectional matching link if a behavioural supertype – subtype relationship holds between them, or the two descriptions are plug-compatible. Consequently, 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 [37]. Related links are always bi-directional and are followed only during development time querying.

Since the trading process depends completely on the graph, its construction should guarantee the semantics of matching. 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 the retrieval of the component description from the repository and the inspection of the component source code 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 and also that the source code for the components is available. If either of these requirements does not hold then the simplest solution is to assume that the offer complies with the description, an approach current traders follow when the interface repository is not available. A more advanced solution is to use validity evidence as part of the component offer [41], which could include a formal proof of correctness, a comprehensive test suite, or references to other people who have used the component 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 above, these issues are dealt with at a middle level, between the simple assumption and the formal proof, by employing information extracted from the design of components, for example a UML diagram with its corresponding OCL expressions.

In a more advanced approach the construction of graph composite nodes and edges can be automated. Automation can be achieved, by employing extensively either automated proofing techniques or reasoning techniques from knowledge representation. Additionally, for the automated construction of composite nodes the work on software architectural styles [42] can 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 further investigation.

3.3. Component composition

Although in this paper we are not looking into component composition issues, we provide some basic component composition functionality in order to support the composite nodes described above. In the description of the architecture we saw that the component composition facility uses the composition strategy and the component offers, which composite nodes keep, to produce composite component. This raises the issue of how we construct the composite component and how we describe the composition strategy.

The composition strategy requires the definition of a composition language to describe the composition process in terms of the various software architectural styles. Additionally, the techniques used in dealing with object interoperability problems [43] can be deployed too. The more kinds of compositions supported the more complex the composition language is and it seems reasonable to provide a modular composition language. The Vanilla framework [44] can provide the necessary support. Finally, the underling mechanism for the composition is based on techniques like Dynamic Skeleton and Interface Invocation, which are parts of the CORBA specification [10] and allow invocation and service of calls without knowledge of the interaction interface supported by the other party.

4. Conclusions and future work

In conclusion, despite progress in software engineering the development of large-scale distributed systems remains a high risk, high cost and problematic task. Component software has the potential to solve this problem, but most of the issues related to it are still open. Central to its use is the ability to locate from a set of working components the appropriate ones. At the same time, trading is the only reasonable way to manage component location in large-scale compositional systems, where complete knowledge of the system is both unreasonable and unrealistic. We propose semantically enhanced component trading as a way to address some of these issues and describe an appropriate trader architecture. Our architecture improves both precision and recall in component location by employing semantic information in component specifications through a combination of formal specifications and conceptual constructs, and a component composition facility. We have developed a component repository that stores the trading space and the component descriptions, and a basic composition facility. We are currently working on the implementation of the relationship service and the development of the graph for the matching process. Initial experimental results with our trader and repository are encouraging.

After the completion of the current work, we plan to extend it in various ways by introducing a number of advanced features. These features include:

- Support for automatic link and composite node creation.
- A query by example mechanism, which will allow queries to provide a component description of their own and the trader will try to match it to the descriptions it already has.
- 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, we intend to 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. C. Szyperski, "Component Software: Beyond Object-Oriented Programming", ACM Press, Addison-Wesley, 1998.
2. W. Kazaczynski and G. Booch, "Component-Based Software Engineering", IEEE Software, vol. 15, no. 5, September/October 1998.
3. P. Zave and M. Jackson, "A Component-Based Approach to Telecommunications Software", IEEE Software, vol. 15, no. 5, September/October 1998.
4. B. Meyer, "On To Components", IEEE Computer, vol. 32, no. 1, January 1999.
5. 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).
6. 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
7. Microsoft, "ActiveX Controls", available online at http://www.microsoft.com/com/tech/activex.asp.
8. Sun Microsystems, "JavaBeans", available online at http://java.sun.com/beans.
9. A. Thomas, "Enterprise JavaBeans Technology - Server Component Model for the Java Platform", Patricia Seybold Group, 1998, available online at http://java.sun.com/products/ejb/white_paper.html.
10. Object Management Group (OMG), "CORBA/IIOP 2.3.1 Specification", October 1999, available online at http://www.omg.org/corba/corbaiiop.html
11. Microsoft, "Component Object Model – COM", available online at http://www.microsoft.com/com/tech/com.asp
12. K. Arnold, B. O'Sullivan, R.W. Scheifler, J. Waldo, A. Wollrath, "The Jini Specification", The Jini Technology Series, Reading, Massachusetts, Addison-Wesley, 1999.
13. J. Deschrevel, "The ANSA Model for Trading and Federation", Architecture Report APM. 1005.01, Approved 15 July 1993.
14. Open Distributed Processing, "Trading function: Specification", ISO/IEC 13235-1:1998.
15. Object Management Group (OMG), "CORBAservices: Common Object Services Specification", December 1998, available online at http://www.omg.org/corba/sectran1.html.
16. 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.
17. Object Management Group (OMG), "CORBA 3.0 New Components Chapters – CORBA Component Model FTF Draft ptc/99-10-04", available online at http://www.omg.org/cgi-bin/doc?ptc/99-10-04.
18. 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 online at http://www.cs.caltech.edu/~kiniry/projects/papers/kiniry/COOTS99_ATW/cdl/cdl.html
19. B. Meyer, "Applying Design by Contract", IEEE Computer, vol. 25, no. 10, October 1992.
20. 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.
21. 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.
22. G. Booch, J. Rambaugh and I. Jacobson, "Unified Modelling Language User Guide", Addison-Wesley, 1997.
23. B. Potter, J. Sinclair and D. Till, "An Introduction to Formal Specification and Z", Prentice-Hall, 1991.
24. 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.
25. R. Wieringa, "A survey of Structured and Object-Oriented Software Specification Methods and Techniques", ACM Computing Surveys, vol. 30, no. 4, December 1998.
26. 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 online at ftp://ftp.cs.caltech.edu/tr/cs-tr98-08.ps.Z

27. 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.
28. 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.
29. 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 online at http://ic-www.arc.nasa.gov/ic/projects/amphion/people/penix/research/rebound/
30. J. Sowa, “Conceptual Structures: information processing in mind and machine”, Addison-Wesley, 1984.
31. 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.
32. 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.
33. M. Ushold and M. Gruninger, “Ontologies: principles, methods and applications”, The knowledge Engineering Review, vol. 11, no. 2, June 1996.
34. S. Heiler, “Semantic Interoperability”, ACM Computing Surveys, vol. 27, no. 2, June 1995.
35. C. Schlenoff, R. Ivester and A. Knutilla, “A robust process ontology for manufacturing systems integration”, February 1999, available online at http://www.ontology.org/main/papers/psl.html.
36. T. Digre, “Business Object Component Architecture”, IEEE Software, vol. 15, no. 5, September/October 1998.
37. J.R Kiniry, “A New Construct for Systems Modelling and Theory: The Kind”, Technical Report CS-TR-98-14, Computer Science Department, California Institute of Technology, October 1998, available online at ftp://ftp.cs.caltech.edu/tr/cs-tr98-14.ps.Z
38. J. Martin and J.J. Odell, “Object-Oriented Methods: A Foundation”, UML edition, Prentice-Hall, 1998.
39. 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.
40. B.H. Liskov and J.M. Wing, “ A behavioural notion of subtyping”, ACM Transactions on Programming and Systems, vol. 16, no. 6, November 1991.
41. 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 online at http://www.infospheres.caltech.edu/papers/fmppta98/paper.html.
42. 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.
43. 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.
44. 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.