Issues in Component-Based Development: Towards Specification with ADLs

Rafael GONZÁLEZ
Departamento de Ingeniería de Sistemas, Universidad Javeriana
Bogotá, Colombia

and

Miguel TORRES
Departamento de Ingeniería de Sistemas, Universidad Javeriana
Bogotá, Colombia

ABSTRACT

Software development has been coupled with time and cost problems through history. This has motivated the search for flexible, trustworthy and time and cost-efficient development. In order to achieve this, software reuse appears fundamental and component-based development, the way towards reuse. This paper discusses the present state of component-based development and some of its critical issues for success, such as: the existence of adequate repositories, component integration within a software architecture and an adequate specification. This paper suggests ADLs (Architecture Description Languages) as a possible means for this specification.

Keywords: Component-Based Development and Software Component specification.

1. INTRODUCTION

An earlier version of this paper was presented at The 3rd International Conference on Computing, Communications and Control Technologies, held in Austin, Texas in July 2005. It has been updated and revised according to the progress we have had in carrying out our research.

We begin by reminding ourselves of some of the typical problems in the development of computer-based information systems: excessive customization (meaning, little or no reuse); complex integration and deployment; lack of coordination standards; lack of interoperability; difficulty in dealing with change; necessity of reducing time and development cost; demanding integration requirements; and lack of run-time flexibility [17]. Software Development has, however, continually evolved to deal with such challenges: from structures, to objects, to components, even perhaps to services. Regarding components, the body of knowledge and tools already has enough drive and potential to allow Component-Based Development (CBD) to deal with the ever-present difficulties mentioned above. This paper discusses the present state of component-based software development and some of its critical issues, such as: the existence of repositories, component integration within software architecture and an adequate ontology and specification for which ADLs may prove useful.

The rest of this paper is structured as follows. Section 2 will provide an overview of what a component is, how it is different from an object and its implications on the issue of software reuse. On the third section, component-based development will be briefly described, presenting it as a new software development paradigm which requires the existence of component repositories and alignment with software architecture and patterns in order to be effective. The fourth section highlights the main issue of this paper, that of component description and modeling; it distinguishes between formal and informal and between static and dynamic specifications, and presents some languages that exist to describe components. On section five, ADLs are presented as a possible alternative which may help in integrating software component description to architecture and pattern abstractions, possibly resulting in better CBD, which is the future research we point at in the final section.

2. FROM OBJECT-ORIENTED DEVELOPMENT TO COMPONENT REUSE

In this section we will describe what a component is and how it is different from an object (or class). After this, we will mention some of the advantages and issues of component-based development in terms of software reuse.

What is a Component?

The definition of a component is ample and potentially ambiguous, which is why, for software, the ontology of a component remains an open and critical issue towards the development of guidelines and practical efforts in component-based software development. From a logical perspective, a component is a way to model real-world concepts in a computer system's domain. This allows for decomposition of complex problems into entities, processes or transactions, for instance. From a "physical" (in the software sense) perspective, components are independent units of software, which implement logical abstractions. Thus, a component is any coherent design unit which may be packaged, sold, stored, assigned to a person or team (for development), maintained and, most importantly, reused [21]. Another cryptic and recursive definition of component is the one provided by Councill *et al.* [11]: "A software component is a software element that conforms to a components model and can be independently deployed and composed without modification according to a composition standard".

With the increasing interest in enterprise architectures, service-oriented architectures and Web Services, the notion of component has been further stretched, in the sense that it can be any physical module of an architecture, any business abstraction implemented with software or any component deployed as a web-enabled service. This adds to the initial confusion with terms like object, building block and module. We do believe the concept of service to be linked to different approaches and technologies that may indicate a service as a specific type of component, but the difference with objects is clearer, as the next section will argue.

From Objects to Components

The notions of instance, identity and encapsulation are associated with the "object" notion, whose properties are: being an instantiated unit with unique identity; having an externally observable state; and encapsulating its state and behavior. A component, on the other hand, is an independent deployment unit, built by a third party, which has no externally observable state [25].

Historically in software development the idea behind libraries, subroutines, and abstract data types, was modularization. In the 80's and 90's, the arrival of the Object Oriented (OO) paradigm brought about the possibility of creating highly encapsulated and easily maintainable systems. In OO, a component was normally seen as a collection of related classes which provided a consistent and logical set of services [22]. For some authors, the fact that CBD is a natural evolution of OO [7][24], gives origin to Object-Oriented Components, which includes benefits such as: interoperability, extensibility, reusability, easy assembly, flexibility in run-time, integrated standards for design, flexible development, speed, quality and reliability when using components-off-the-shelf (COTS) [17].

However, it seems that the OO paradigm has not proven to be useful enough for reuse, because most OO applications are developed relatively from scratch or require complex efforts to exploit reuse, including adaptations and inheritance mechanisms that effectively make the system, although built with reused classes, hardly reusable. Furthermore, while the OO paradigm is based on programming techniques and models, CBD extends such ideas to other areas, which complement the programming field. For example, to be able to achieve an effective interaction among components, it is necessary to adopt rigorous design and documentation disciplines, and modeling standards. This is the basic notion of Design-by-Contract, a discipline conceived within the OO paradigm, which fits more naturally inside the CBD approach [5]. In other words, component-based development is not just about developing with components, but developing with component-based models, formalism, project design and tools.

CBD Implications on Reuse

The most important advantage of CBD is the possibility of reuse it brings into the software development field. The main idea is to be able to build systems based on reliable and already tested components as it is done in other engineering disciplines [3]. To achieve this, it is important to extend our comprehension from a software understanding (where models describe software technically and code is the focus) to a component understanding (where the functionality, application and adaptation requirements of components gain importance) [2]. This means that besides a model which technically describes the software, the component must be viewed and understood in terms of its reusability (functionality, requirements and restrictions). When this is fully achieved, it will be possible to build systems by means of component integration and by directing programming efforts to the integration and not to coding functionality.

Effective software reuse offers robustness, reliability and interoperability, but these benefits must outweigh the cost of achieving reuse and it seems that CBD is not there yet. Some argue that building a system from scratch may still be cheaper than building a mostly reused fully component-based system [23]. Aside from the adaptation cost, components usually have an undesired performance cost also (performance is usually proportional to customization), resulting in software that, in most cases, performs less efficiently than one fully customized from scratch [5]. Another difficulty in achieving successful reuse is that the original developer of a component usually makes implicit assumptions with regards to the possible applications of his components, making these uses implicit and hard to find by other developers attempting to reuse the component [12]. A further concern might be that depending on reusable components goes against the rapid change in software technology and the need for innovation, but as we can see from electronics or mechanical engineering, working with components does not need to imply that.

It can be concluded that reuse is not a direct benefit of CBD; for it to be effective, it must be coupled with adequate component's description that aims toward reducing adaptation costs and that will allow performance improvement, without affecting significantly its interoperability or increasing its development cost. One way to achieve this is by using a component's formal specification; such specification should include the component's static restrictions (consistency) and dynamic restrictions (allowed sequences of execution, allowed redefinition). It might be naive to believe that there already exists a component in a repository satisfying in its entirety the requirements of a given problem; thus, the component must be adapted to the problem and to the system's architecture, so it can be used in that context. The other necessary issue, as we have already mentioned, is to focus on components through a component-based approach; this will de discussed in the next section.

3. COMPONENT-BASED DEVELOPMENT

Component-based development can be regarded as a new paradigm of software construction. Some of its success factors are: the existence of an adequate component repository and the placement of components within the context of architectures, patterns and structures.

CBD as a new Software Development Paradigm

Traditional information systems development is based on large work groups and long periods of time, which has resulted in undesirable economical and chronological consequences, disrupting an organization's competitiveness. The biggest challenge for developing successful software systems will be to build systems in a shorter lapse of time, with lower cost and inside a changing and complex environment [8]. This also suggests the need for a more permanent solution, which will bring the possibility of information systems development that may be flexible and adaptable to an organization's conditions and existing technology. Research and practice point towards the CBD paradigm as one feasible way to achieve this goal

[1][3][8][15][28]. This approach includes improvements in: quality, throughput, performance, reliability and interoperability; it also reduces development, documentation, maintenance and staff training time and cost [18]. Most recent trends in software engineering show that future developments will follow in the CBD path. This argument is partially confirmed by the large amount of component development technologies that exist today (CORBA, EJB, DCOM, and .NET among others), and also given the amount of components (COTS) available in the market [2].

Although CBD promises to improve the software development processes, quality, productivity and reuse in particular [27], these achievements are not new in other areas of industry [3]. The biggest contribution of the Industrial Revolution was precisely this. From that point forward this paradigm has been used in the electronics, automotive, and construction industries. Automobile parts, hardware components (such as video and networking cards), and construction design patterns are reused in every new project, making use of rigorous and reliable standards, without losing innovation in the process (as the dynamics of these particular sectors has shown).

In the last few years Component-Based Software Engineering (CBSE) has emerged as a new paradigm for plug-and-play software, where components are provided and stored in component repositories which provide the interface information for each component. This approach supports complex and usually distributed applications, while reducing maintenance costs and increasing reliability [21].

This new form of applying Software Engineering requires particular aspects for its success, such as: component and architecture selection; adaptation and integration of components inside the chosen architecture; and maintenance of the components along with the evolution of requirements [2][3][16]. This new approach might not be easy to implement, because of the fact that it must guarantee coexistence and compatibility among different component versions, and from different sources. So in order to be able to maintain independence, which facilitates maintenance of third party components, it is recommended to divide CBSE into two levels: the component level and the application level [5]. This means that in one level, the engineering of the components itself is the focus, while in the second level, the application engineering is the focus (without, off course, disconnecting these two aspects).

CBSE remains as an immature discipline [3] that can learn from other engineering areas, and from the experience of the object-oriented paradigm and traditional software engineering, but for it to be successful, it most be used along with an adequate support by a component repository.

Component Repositories

In order to make CBD more cost-effective and allow components to be more easily found, good repositories must be available. The aim is to obtain effective, easy to use (navigate), complete and efficient repositories. However, a problem arises from using open (on-line) repositories: it must be guaranteed that components comply with the minimum security expectations [20]. An integrator or architect will expect components developed by third-parties to have the properties (behavior) which the creators claim, along with maintaining the basic requirements of quality, legitimacy, abstraction, encapsulation, low coupling and high cohesion.

This concern supports the idea that a certifying authority is required; one which should aid in the following: 1) outsourcing (managing the outsourcing contract for the development of components and auditing the performance of the developing institution); 2) component selection (selection of the most adequate components according to user requirements, such as functionality, security, trustworthiness and performance); and 3) testing of components (verifying that components satisfy the requirements with acceptable quality and reliability) [9].

Aside from certifying conformance, it is useful that the certifying authority guard the relevant security aspects, studying components in the following manner [20]: 1) characterizing atomic components independently; 2) checking the compatibility of security features among components; and 3) validating property visibility with respect to external entities. Some additional efforts would indeed be desirable, such as the implementation or adoption of security standards, such as the Department of Defense's Orange Book, the NIST Common Criteria or ISO/IEC's 15408 standard. An example of a repository that both validates a component's properties and compliance to standards is the CLARiFi project [6][10].

Architecture and Patterns

Another critical issue is the place of components within an architecture with the aid of design patterns. In this context, architectures are a set of decisions regarding the platform, component frameworks and interoperability design among such frameworks [25]. In recent years, software development has oriented its focus upwards, in the direction of an abstract level of architecture specification [5]. The transition towards CBD has introduced new distributed technological infrastructures based on Microsoft's COM (Component Object Model), OMG's CORBA (Common Object Request Broker Architecture) or Sun Microsystem's EJB (Enterprise Java Beans). When these models are coupled to a well-defined layered architecture, they become the best technological support to develop the infrastructure necessary for component-based systems [17].

Besides software architectures, design patterns exist to optimize object-oriented systems design, based on a catalogue of generic structures guided towards solving recurring problems. Some actually see design patterns as micro-architectures [25]. Yau and Dong [27] propose the use of design patterns for integrating components in CBD. After designers have selected a design pattern to describe relationships among components (within a particular system), the pattern must be instantiated; this instantiation consists in transforming the participating relationships into design interactions. After this, the structure and interactions are verified, to guarantee that the pattern's restrictions are complied with, and wrappers can then be employed as decorators of the components. In this way, architectures and patterns are formally integrated into the system's design.

With component-based software development, the engineering process centers its attention on high-level design through architecture and design patterns, in which components are assembled and adapted to this design. Such assembly and composition, requires semantic clarity and detailed specification, accompanied by static and dynamic models of the components in isolation or within determined design or architecture patterns. This is why the rest of this paper focuses on the issue of component description and modeling.

4. COMPONENT DESCRIPTION AND MODELING

A component is usually specified (described) in terms of its interface (as Interface Definition Languages do), but this offers no information regarding the component's performance, security or synchronization [12]. More detailed descriptions are useful or necessary, and they can vary in terms of formality, dynamics and modeling language.

Formal vs. Informal Specification
There are three different options for specification: it may be informal, formal or semiformal. Informal specification concentrates on describing the component through natural language; formal specification presents more detailed aspects such as semantic information or requirements in formal logic; semiformal approaches describe components without being either totally formal or totally unstructured [12][21][23].

Detailed and formal descriptions of a "white box" type could provide enough information for the use and understanding of a component, but the effort required to use and comprehend this formal models dissuades developers from using it, inclining them towards "black box" type descriptions [2][26]. Because of this, a more appropriate model would be one that offers not only understanding of the domain (requirements vs. capabilities), but also of the program itself (interfaces, data types, syntax, parameters, and acceptable ranges) and the situation (structure, connections, and flows) [2].

One semiformal proposal is LIPS [12] which, besides use policies, shows instance and thread support information. Another semiformal approach is CDM (Component Description Manager) [21] that presents a classification framework for the effective management of components, based on problem domain, semantic information and the component's ontology. The resulting classification tree concentrates on characteristics, grammar, components and standards. Such classifications are useful in different contexts: for instance, in component brokers or repositories, they help in finding and identifying components; in a development environment, they help in verifying compatibility and performance of the components.

Static vs. Dynamic Specification
Aside from the level of formalism, a component may also be specified statically or dynamically. The static structure of a component is described in terms of the services that the component provides and which remain valid for any instance of the component; such specification, however, makes it difficult to describe certain restrictions [5]. Dynamic structure may describe legitimate call sequences and behavior of the services or operations of the component in a given moment of the component's execution (i.e. in run-time) [22]. An example of this type of dynamic description is provided by the Abstract State Machine Language (AsmL) based on Microsoft's .NET technology, which incorporates non-determinism ad transactions to generate IL (.NET's intermediate language) and verify a component in run-time [4]. Another approach consists in defining the component in a mathematical and hierarchical fashion, based on finite state-machine modeling of classes and the components which contain them. [22].

Fortunately, it is possible to describe a component both formally and informally and both statically and dynamically, as a developer might need a first informal description to get acquainted with the component and a formal one for adapting or integrating it. He or she might also want to know the static composition and properties of the component as well as the dynamic behavior and run-time restrictions. An effective modeling language for components should account for all these possibilities or views.

Some description and modeling languages
Along with the type of specification available for a component, there is the language chosen to specify it. Such languages may be formal, semiformal or formal to answer to the required type of specification, but they must also be standard languages in order to stimulate adoption, comprehension and interoperability.

The UML is an obvious natural candidate for describing components given its widespread adoption and generic nature. Its strengths are (among others) that it integrates in a natural way, various modeling approaches (disconnected in the past) and thus becomes a stronger, more mature and integrated alternative; it also offers enough popularity and available tools to make it attractive in practice. However, some argue that it is still ambiguous for describing components and that its understanding of components is limited [14]. For instance, valid redefinition of operations is a restriction that is not formally modeled with UML [22]. Nonetheless, the use of UML within a set of object-oriented practices and languages, make it one of the best modeling languages for designing (visually, in particular) systems and describing components [19]. Versions of UML after 2.0 also have an increased understanding of components, with a higher level abstraction, but a remaining focus on components as deployable software units.

One particular effort which extends the UML for components is the Catalysis approach [13]. CBD and Catalysis are clear examples of modeling techniques which support analysis and design of components [21]. Catalysis offers a non-ambiguous definition of component and subsystem interfaces to ensure adherence to a given model and set of business rules. Indeed, this means that Catalysis allows the description of a component according to the component and application levels presented in section 3.1 [2]. Its disadvantage is that any representation of a component with Catalysis means that the component is built with the Catalysis approach, making it difficult to describe an already made component or adopting a different development method. It should also be noted that even though it treats components, Catalysis is mainly an object-oriented approach.

Another language worth pointing out is Microsoft Research's AsmL (already mentioned), an executable specification language that provides the possibility of verifying components statically and monitoring behavior dynamically.

One could also think of using architecture description languages (ADLs) as possible languages for specifying components. This possibility will be discussed in the last section.

5. ADLs FOR COMPONENT SPECIFICATION

Although ADLs are aimed at architectures, these are made up (basically) of components and connectors. Although their high level of abstraction and limited commercial adoption might make them less attractive, we have already mentioned the tendency of CBD to focus in architectures and patterns, making ADLs desirable for representing components within high-level structures.

ADLs describe component interfaces without getting into their internal details [31]. So the first question that arises is whether or not ADLs are enough to describe components. We argue that it is enough when structuring component-based systems from an architectural point of view in which integrating components means doing so through their interfaces, regardless of how the component is implemented. Of course, in reality, often the architect or developer dives into the component's code, but this shouldn't be necessary if, and this is precisely what we are striving for, the component's interface is well defined along with the non-functional aspects of the component.

Another advantage of ADLs within the CBD approach is that it helps in assigning tasks to the development team, as well as helping guarantee that the different constraints imposed by the interfaces and the architectural structure are satisfied [30]. This means that an ADL departs from object-oriented modeling languages and helps in focusing on higher-level issues and contributes to the project planning and management, which we have already argued is critical to the success of CBD.

Even if each ADL has a different goal, most share a common ontology [29] and support the following elements:

1) Components: the primary computational elements and data stores of the system.
2) Connectors: interactions between components.
3) Systems: configurations of components and connectors as a topology.
4) Properties: semantic information of a system and its components.
5) Constraints: facts of the architectural design that must remain true.
6) Styles: families of related systems.

This ontology suggests two additional benefits. The first is that regardless of which specific ADL to choose from, there is already a common vocabulary and agreed representational interest. The other is that it offers the possibility of describing properties, constraints and styles which exceed the components in isolation and both clarify and restricts component integration within well-defined architecture decisions.

6. CONCLUSIONS AND FUTURE WORK

In this paper we have established some of the advantages and challenges associated with component-based development. We consider component technologies as an excellent tool for building robust software in a changing environment with many restrictions. At first, we clarified the differences between components and objects and how the first are more suitable for software reuse, through flexibility and reliability. We have also pointed out some critical issues for effective CBD: the existence of component repositories is one of the most important requirements; the existence of conditions and guarantees of security is also fundamental; having components supported by and integrated within structures and high-level architectures is also seen as a very desirable approach.

But the issue that is of highest relevance in this paper is component specification. Without adequate specification, components will not be found, understood or used effectively. It has been underlined that it is necessary to define the level and characteristics of specification (formal vs. informal and static vs. dynamic) and also to select an adequate specification language (among many available).

We argue that by using ADLs to describe components in context, it will be possible to offer a description that is better suited to the CBD approach, by considering higher-level issues. In this same sense, we believe that by integrating research in CBD with research in software architecture we can finally take mature steps towards reliable, effective, dynamic and reusable component-based software development, by means of integration, storage, distribution and adequate description of components.

7. REFERENCES

[1]. Allen, P.; Frost, S., **Component-Based Development for Enterprise Systems: Applying the Select Perspective**, Cambridge University Press, 1997.

[2]. Andrews, A.; Ghosh, S., Choi, E. "A Model for Understanding Software Components," **Proceedings of the International Conference on Software Maintenance**, Oct. 2002 (ICSM' 02), pp. 359–368, IEEE Computer Society, 2002.

[3]. Apperly, H. "The Component Industry Metaphor," Heineman, G. & Councill, B. (Eds.) **Component-Based Software Engineering: putting the pieces together**, Addison-Wesley, Boston, 2001.

[4]. Barnett, M. *et al.* "Serious Specification for Composing Components" in Crnkovik, I. *et al.* (Eds.) **Proceedings of the 6th ICSE Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction**, Portland, Oregon, USA, May 2003, Carnegie Mellon University, 2003.

[5]. Bertolino, A.; Mirandola, R., "Towards Component-Based Software Performance Engineering" in Crnkovik, I. *et al.* (Eds.) **Proceedings of the 6th ICSE Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction**, Portland, Oregon, USA, May 2003, Carnegie Mellon University, 2003.

[6]. Brereton, P. *et al.*., "Software components - enabling a mass market," **Proceedings of the 10th International Workshop on Software Technology and Engineering Practice**, Oct. 2002, pp. 169–176.

[7]. Brown, A.; Wallnau, K., "The Current State of Component-Based Software Engineering", **IEEE Software**, September/October 1998. IEEE, 1998

[8]. Brown, A., **Large-Scale Component-Based Development**, Prentice-Hall PTR, 2000.

[9]. Cai, X. *et al.*, "Component-based software engineering: technologies, development frameworks, and quality assurance schemes," **Proceedings of the Seventh Asia-Pacific Software Engineering Conference**, 2000. APSEC 2000, Dec. 2000, pp. 372–379, IEEE.

[10]. Ci, J.; Tsai, W., "Distributed component hub for reusable software components management," **Proceedings of the 24th Annual International Computer Software and Applications Conference**. COMPSAC 2000., Oct. 2000, pp. 429–435, IEEE.

[11]. Councill, B.; Heineman, G., "Definition of a Software Component and its Elements," Heineman, G. & Councill, B. (Eds.), **Component-Based Software Engineering: putting the pieces together**, Addison-Wesley, Boston, 2001.

[12]. DePrince, W.; Hofmeister, C., "Usage Policies for Components," Crnkovik, I. *et al.* (Eds.), **Proceedings of the 6th ICSE Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction**, Portland, Oregon, USA, May 2003, Carnegie Mellon University, 2003.

[13]. D'Souza, D.; Wills, A., **Object, Components and Frameworks with UML: the Catalysis approach**, Addison-Wesley, Reading, 1999.

[14]. Eden, A., "A Theory of Object-Oriented Design," **Information Systems Frontiers**, vol. 4, No. 4, pp. 379-391, Kulwer Academic Publishers, 2002.

[15]. Eeles, P.; Sims O., **Building Business Objects**, John Wiley & Sons, New York, 1998.

[16]. Griss, M.; Pour, G. "Accelerating Development with Agent Components", **IEEE Computer**, vol. 34, No. 5, May 2001, pp. 37–43, IEEE, 2001.

[17]. Hall, L. *et al.*, "COTS-based OO-component approach for software inter-operability and reuse (software systems engineering methodology)," **Proceedings of the Aerospace Conference**, vol. 6 , March 2001, pp. 2871–2878, IEEE, 2001.

[18]. Herzum, P.; Sims, O., **Business Component Factory**, John Wiley & Sons Inc., 2000.

[19]. Houston, K.; Norris, D., "Software Components and the UML," Heineman, G. & Councill, B. (Eds.), **Component-Based Software Engineering: putting the pieces together**, Addison-Wesley, Boston, 2001.

[20]. Khan, K.; Han, J., "A security characterization framework for trustworthy component based software systems," **Proceedings of the 27th Annual International Computer Software and Applications Conference**, 2003. COMPSAC 2003., Nov. 2003, pp. 164–169, IEEE, 2003.

[21]. Meling, R. *et al.*, "Storing and Retrieving Software Components: a component description manager," **Proceedings of the Australian Conference on Software Engineering** , April 2000, pp. 107–117, IEEE, 2000

[22]. Moisan, S. *et al.*., "Behavioral Substitutability in Component Frameworks: a Formal Approach," **Proceedings of the SAVCBS'03 Specification and Verification of Component-Based Systems 2003 Workshop at ESEC/FSE'03**, Helsinki, Finland, September 2003, pp. 22-28, ACM, 2003.

[23]. Morel, B.; Alexander, P., "Automating Component Adaptation for Reuse," **Proceedings of the 18th IEEE International Conference on Automated Software Engineering**, pp. 142-151, Oct. 2003. IEEE, 2003.

[24]. Orfali, R., Harkey D., y Edwards J., **The Essential Distributed Objects Survival Guide**, John Wiley & Sons, 1996.

[25]. Szyperski, C., **Component Software: Beyond Object-Oriented Programming**, Second Edition, ACM Press, Addison-Wesley, New York, 2002.

[26]. Weinreich, R.; Sametinger, J., "Component Models and Component Services: concepts and principles," Heineman, G. & Councill, B. (Eds.), **Component-Based Software Engineering: putting the pieces together**, Addison-Wesley, Boston, 2001.

[27]. Yau, S.; Dong, N., "Integration in component-based software development using design patterns," **Proceedings of the the 24th Annual International Computer Software and Applications Conference**, 2000. COMPSAC 2000, Oct. 2000, pp. 369–374, IEEE, 2000.

[28]. Zahavi, R., **Enterprise Application Integration with CORBA: Component and Web-Based Solutions**, OMG Press, John Wiley & Sons, Inc., 2000.

[29]. Graham, D.; Monroe, R. & Wile, D. "Acme: Architectural Description of Component-Based Systems" in Leavens, G. & Sitaraman, M. (eds.), **Foundations of Component-Based Systems**, pp. 47-68, Cambridge University Press, 2000.

[30]. Grau, A.; Shihada, B. & Soliman, M. **Architectural Description Languages and their Role in Component Based Design**, Project Report, Department of Computer Science, University of Waterloo, Canada, Available: http://www.cs.uwaterloo.ca/~bshihada/adl.pdf, 2002.

[31]. Riemenschneider, R. & Satvridou, V. "The Role of Architecture Description Languages in Component-Based Development: The SRI Perspective" in **Proceedings of the International Workshop on Component-based Software Engineering held in conjunction with the ICSE1999**, LA, May 1999, pp. 203-206.