A Strategy for Selecting Multiple Components

Ed Mancebo
School of EE & CS
Washington State University
P.O. Box 642752
Pullman, WA 99164-2752
emancebo@eecs.wsu.edu

Anneliese Andrews
School of EE & CS
Washington State University
P.O. Box 642752
Pullman, WA 99164-2752
aandrews@eecs.wsu.edu

ABSTRACT

This paper presents a systematic method for simultaneously defining a software architecture and selecting off-the-shelf components for reuse. The method builds upon existing techniques for component selection and architecture evaluation. We identify architectural decisions that have a large effect on the components used early in the process so that different ways of building the system can be investigated. The result of applying the method is a partial definition of a system's architecture along with a set of components that could be incorporated.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures; D.2.13 [**Software Engineering**]: Reusable Software

Keywords

Component-based software engineering, component selection, software architecture, multi-criteria decision making

1. INTRODUCTION

Maintaining consistency between the architecture of a system and the components available in the market is a persistent concern in component-based software engineering. When functionality is packaged as a component, architectural assumptions are made that affect the role it plays in a system [8]. By incorporating off-the-shelf components, one accepts their architectural restrictions. Implicitly, the selection of components and the definition a system's architecture are intertwined [7, 17]. When the architecture chosen does not accommodate a particular type of component, we have a choice: either change the architecture or use a different component. The presence of this choice is fundamental to the selection of multiple components. Allowing the flexibility to make changes in either area complicates the design process—instead of performing a unilateral comparison between components available and components needed, we must manage the simultaneous evolution of both.

This complication can be eliminated by reducing the multiple selection problem to several instances of the traditional selection problem. Instead of trying to match components with a changing architecture, we specify multiple alternatives for the architecture that remain constant. The fitness of each component in one architecture or another is evaluated by analyzing the architectural assumptions it makes (at least those that are discernible early in the process). Essentially, we are treating each possible change as a different architecture and then analyzing the components that can be used with that architecture (using traditional component selection techniques). Each architecture and its set of corresponding components is a solution. At this point we must decide which solution is the most favorable.

This paper presents a systematic method for defining components and architecture together. Applying the method produces a set of feasible approaches for building a system (an approach is an architecture and a set of corresponding components). Given these alternatives, we describe how to evaluate the tradeoffs involved with each approach to arrive at an outline of the architecture and components that best satisfy the needs of the system. Section 2 describes background information, and section 3 details the method. We discuss conclusions and areas for future work in section 4.

2. BACKGROUND

2.1 Definitions

Component A component is an existing piece of software written with reuse in mind that can be deployed with little or no modification. We assume that components are designed to be used in certain types of applications, which implies that there are constraints regarding the incorporation of a component into a system. Components can be obtained in-house or off-the-shelf. A more precise definition of a component can be found in [22, 9].

Architecture Software architecture deals with the definition of components, their external behavior, and how they interact [15, 2]. In this context an architecture contains a description of component needs or roles. The process of architectural definition can be viewed as a number of architectural decisions that need to be made. We are chiefly concerned with the way in which these decisions affect the components needed by the system.

Architecture can be expressed informally; or using modeling (as in UML 2.0 [3]) or an architecture description lan-

SAC'05 March 13-17, 2005, Santa Fe, New Mexico, USA

guage [15]. A more formal way to express an architecture is useful because it may uncover more subtle architectural assumptions; however, the method we present does not rely on any specific technique for specifying architecture.

Architectural Approach The process of defining an architecture involves decisions about components and their interactions. An architectural approach captures instances of these decisions, thereby partially specifying the components needed by a system and restrictions on how they interact.

Implementation Approach The specification of an architectural approach provides us with a list of components needed. When an architectural approach is combined with a set of actual off-the-shelf components that meet these needs, the result is an implementation approach.

2.2 Related Work

Research in the area of component selection is closely related to the method described in this paper. Our method builds upon existing component selection techniques to address a more complicated problem. Component selection methods are traditionally done in an architecture-centric manner, meaning they aim to answer the question: given a description of a component needed in a system, what is the best existing alternative available in the market? Existing methods include OTSO [12], STACE [13], and BAREMO [14]. These methods define criteria upon which to judge alternatives for a component role and the synthesis of multiple criteria to decide the most promising alternative.

Another type of component selection approaches is built around the relationship between requirements and components available for use. The goal here is to recognize the mutual influence between requirements and components in order to obtain a set of requirements that is consistent with what the market has to offer. This is done by merging requirements engineering and component selection techniques. Examples of these methods include PORE [16] and CRE [1].

In relation to existing component selection methods, our approach aims to achieve goals similar to those of PORE and CRE, except we are interested in the relationship between architecture and components instead of requirements and components. While the mutual influence between requirements, architecture, and components is well known [7, 17] we are unaware of any systematic methods that address the simultaneous definition of architecture and component choices.

In any component selection method, it is unrealistic to expect a perfect match between components needed and components available. A group of components that compose a system may have overlaps and gaps in required functionality. A gap represents a lack of functionality; an overlap can cause a confusion of responsibility and degrade non-functional properties like size and performance. Techniques to express and minimize gaps and overlaps are described in [18] and [10].

While existing component selection techniques guide the analysis of components, architecture evaluation analyzes the properties of architectures. Of the methods in this area, ATAM [11] is particularly relevant due to its treatment of quality attributes and scenarios. An architectural decision is analyzed by observing the effect it has on certain properties (quality attributes) of an architecture. Quality attributes are elicited through the use of scenarios. Alternatives for an architecture can be evaluated by applying scenarios and analyzing how well each alternative satisfies the relevant quality attributes. These techniques are used in our method to evaluate architectural approaches.

The interdependencies between architecture and components can be traced back to the phenomena of architectural mismatch. Influential papers in this area include [8] and [21]. Mismatch occurs when components in a system make conflicting architectural assumptions. In the context of this paper, we determine architecture and components by finding a satisfactory solution with a minimal amount of mismatch.

Finally, we rely on existing multi-criteria decision making techniques to balance tradeoffs. The Analytic Hierarchy Process (AHP) [19] provides a method to assign a relative importance to each criterion in a consistent manner. Using the notion of a utility function [4], values measured for each criterion can be mapped to a common scale that represents "overall usefulness". For a good description of the subtleties of multi-criteria decision making, we recommend [5].

2.3 Example System

An example system is referenced throughout the paper for clarification. The system under analysis was a senior design project at Milwaukee School of Engineering named *Reaction! Presentation Software. Reaction!* allows the user to create multimedia enhanced presentations that include synchronized 3D animations, audio, and video. A sophisticated component selection method is ideal for this type of application due to the large potential for reuse and the diversity of components available.

Figure 1 sketches a generic architecture for this system. A presentation is authored through a user interface. To execute the presentation, the file is parsed and interpreted to create a collection of Scene objects (for each slide) that get rendered. The rendering component makes use of various APIs.

3. A REDUCTION APPROACH

The difficulty of orchestrating the changes between architecture and components directly is that a decision made in one area often influences the other area in a way that is not well understood up front. The reduction approach is advantageous because we postpone making architectural decisions until we have analyzed the constraints that those decisions put on the components that can be used.

The process consists of five steps, which are shown in Figure 2. The numbers order the steps in the process, which are explained in the following subsections.

3.1 Choosing Key Architectural Decisions

Our goal is to model a flexible architecture with specifications for multiple architectures that will remain constant. A set of fixed architectures allows components to be selected using traditional techniques. However, it is not practical to anticipate and formulate every possible change that could be made. Instead, we must focus on identifying and modeling only the most important changes.

The process of defining an architecture can be viewed as a set of decisions. Decisions are made on various aspects of an architecture, for example: architectural style, decomposition of functionality into components, infrastructure, protocols, and data formats. To identify these decisions, one must understand what the system will do and have ideas for how it could be built.

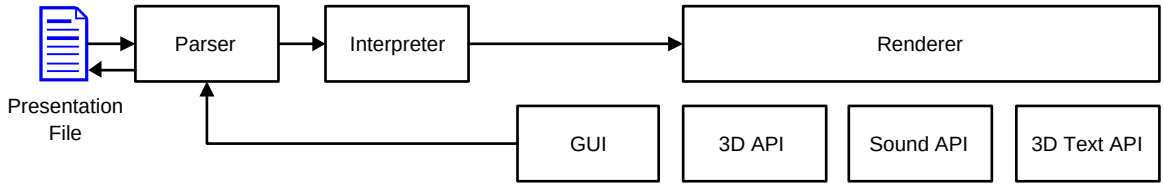

Figure 1: A sketch of the architecture for the *Reaction!* system.

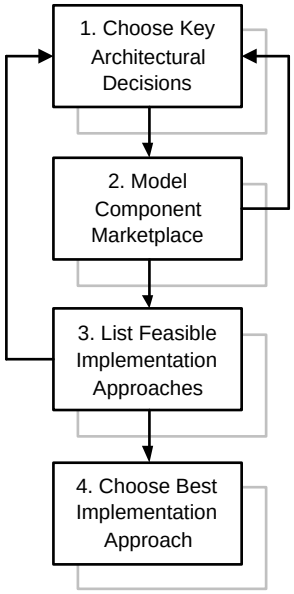

Figure 2: Steps in the process. The upstream arrows indicate iterative paths through the steps.

An architectural approach can be characterized by choosing an alternative for each aspect. Each choice places different limitations on the components that could feasibly be used. The architectural decisions identified are analogous to sensitivity points in constraint optimization problems. Each decision is an independent variable, and the alternatives for a given variable represent the values it can assume. The dependent variable is the set of components that could feasibly be used in the system. A decision is a sensitivity point if different alternatives for that decision yield widely varying sets of components to choose from.

The knowledge required to judge which decisions will have a large influence on component selection can be derived from domain knowledge. Additionally, it may be useful to do an initial exploration of the component marketplace to identify trends in the types of components available. The underlying architectural decisions at work can be inferred from the differences between predominant component types.

An architectural approach is the result of only a few key decisions to fully specify an architecture. Key decisions must affect component selection. Decisions that are independent of the components used can be made at any time based on the merit of the alternatives themselves and do not need to be considered when analyzing the relationship between architecture and components.

We consider three architectural decisions in the analysis of the *Reaction!* system. The first two decisions relate to the distribution of functionality across components, and the third decision concerns data format.

1. **Object Abstraction** A presentation will include a number of graphics objects. Should these objects know how to render themselves (KAD_{1a}), or should they be decoupled from rendering mechanisms (KAD_{1b})?

2. **Object Communication** How are the objects synchronized? Will they keep track of timing individually (KAD_{2a}), or will a master object coordinate them (KAD_{2b})?

3. **Presentation Format** What is the best file format to store presentations in? Choices could include XML (KAD_{3a}) or as serialized objects (KAD_{3b}).

3.2 Modeling the Component Marketplace

The next step identifies candidate components for reuse. This is by nature a loosely defined activity because a specific description of the components needed is not available yet. Thus, the goal is to identify components that may prove useful in the construction of the system. The functional requirements of the system should provide some guidance, in that one can search for components that provide some portion of the functionality described by the requirements.

Each candidate component needs to be analyzed with respect to the assumptions it makes about the type of architecture it will be used in. When this information is coupled with a description of each component's basic functionality (provided and required), we have a portrait of the components the marketplace has to offer and the types of applications they were designed for. The result is a Model of the Component Marketplace (MCM), that we will utilize later to decide which components satisfy certain architectural constraints.

We represent the MCM in three matrices. Candidate components are listed along the columns of each matrix. A *requirements fulfillment matrix* and *component dependency matrix* show the functionality provided and required by each component. An architectural assumptions matrix documents the relationship between each component and the key architectural decisions. Tables 1, 2, and 3 show examples of these matrices.

The relevancy of each architectural decision can be evaluated by observing the *architectural assumptions* matrix. Decisions that have no relationship with any candidate components (e.g. KAD_2 and KAD_3) can be removed—they don't help characterize the relationship between architecture and available components. The most significant key architectural decisions are found by iteratively selecting them and analyzing their relationship with existing components.

Table 1: Abbreviated Requirements Fulfillment Matrix: Each mark signifies that the component in the column provides the functionality listed in the row.

	Ogre	DirectX	OpenGL	OpenAL	OGLFT	Qt
Scene Mgmt.	x					
Graphics API		x	x			
Audio		x		x		
3D Text		x			x	
GUI Widgets						x

Table 2: Abbreviated Architectural Assumptions Matrix: The use of Ogre implies a commitment to KAD_{1a}, where objects render themselves.

	Ogre	DirectX
KAD_1	1a	
KAD_2		
KAD_3		

Table 3: Abbreviated Component Dependencies Matrix: Ogre requires an underlying graphics API, and DirectX can only be used with MFC. OGLFT and GLTT are used with OpenGL

	Graphics API	MFC	OpenGL
Ogre	x		
DirectX		x	
OGLFT			x
GLTT			x

With these matrices, it is possible to determine if a particular choice of architectural alternatives (an architectural approach) has an associated set of components that provides the necessary functionality. An architectural approach with such a set of components constitutes a feasible implementation approach.

3.3 Enumerating Feasible Implementation Approaches

Each implementation approach represents a different way to build the system. Before defining an implementation approach more formally, we must define an architectural approach. Step 1 of the process defined a set of key architectural decisions $KAD_1 ... KAD_n$. Each key architectural decision has an associated set of alternatives, represented by the sets $A_{KAD_1} ... A_{KAD_n}$. An architectural approach is a member of the set $A_{KAD_1} \times A_{KAD_2} \times ... \times A_{KAD_n}$. In the *Reaction!* analysis there are only two distinct architectural approaches (since we eliminated KAD_2 and KAD_3), for example $AA_{ex} = (KAD_{1a})$.

Let C_u be the set of all candidate components identified in step 2 of the process. An implementation approach IA is defined by the tuple (AA, C) where AA is an architectural approach, C is a set of components, and $C \subseteq C_u$. Using the example architectural approach from above, we can construct an implementation approach by adding a set of components: $IA_{ex} = (AA_{ex}, \{DirectX, Ogre, Qt\})$.

An implementation approach $IA = (AA, C)$ is feasible if it satisfies the following constraints:

1. The components in C satisfy the functionality required by the system. This condition is met if each row of the *requirements fulfillment matrix* is covered by at least one component in C.
2. C is self-sufficient. Every dependency documented in the component dependency matrix is provided by a component in C.
3. The alternatives chosen in the architectural approach AA are consistent with the assumptions in the *architectural assumptions matrix* for each component in C.

Our example implementation approach IA_{ex} satisfies conditions 1 and 3. The set of components for IA_{ex} is not self-sufficient because DirectX requires MFC, which is not a member of the set.

Using this definition, multiple feasible implementation approaches can be associated with the same architectural approach. For example, $(AA_{ex}, \{DirectX, Ogre, MFC\})$ and $(AA_{ex}, \{DirectX, Ogre, MFC, Qt\})$ are both feasible. To reduce the number of superfluous permutations, it is useful to define a greatest implementation approach. An implementation approach (AA, C) is the greatest implementation approach for the architectural approach AA if $\forall C'((AA, C')$ is feasible$) \Rightarrow C' \subseteq C$. According to our definition of feasibility, there can only be one greatest implementation approach.

Calculating the greatest feasible implementation approaches using the matrices from the MCM is straightforward. Up to this point, we have assumed that a combination of components is either feasible or not. In reality, a set of components may be a viable option even when some level of architectural mismatch is present. To accommodate this, we can make the stipulation that only strictly infeasible implementation approaches are to be eliminated. Implementation approaches that introduce an acceptable level of mismatch are included in the set of possibilities. Alternatively, a more restrictive policy would more selectively choose which implementation approaches are feasible, but run the risk of eliminating the best option.

To deal with the option of creating a custom component if no satisfactory off-the-shelf components exist, we introduce the concept of a generic custom component. If the components for an implementation approach do not satisfy the first feasibility constraint, a generic custom component (to be constructed later) is used as a place holder. This preserves the implementation approach as an option until the next step, where we must evaluate if it is worthwhile to build the custom component.

Table 4 shows the greatest feasible implementation approaches for the *Reaction!* system. An implementation approach may include many components that provide the same functionality. If these components are interchangeable, then the implementation approach is *consistent*. Inconsistent implementation approaches mean that some dependencies are remaining and need to be removed using another iteration

with adjusted key architectural decisions (see the upward arrow in Fig. 2).

Table 4: Greatest feasible implementation approaches.

Architectural Approach	Components
(KAD_{1a})	C_u
(KAD_{1b})	$C_u - \{Ogre\}$

The *Reaction!* analysis contains inconsistent implementation approaches because Qt and MFC are not interchangeable due to the dependency between MFC and DirectX (see the *component dependency matrix*). To remedy this, the choice of GUI toolkit should be made a key architectural decision (KAD_{2a}: MFC, and KAD_{2b}: Qt). The same problem exists regarding the dependency between OGLFT/GLTT and OpenGL. Thus, KAD_3 becomes the choice of graphics API, where KAD_{3a}: OpenGL and KAD_{3b}: DirectX. After repeating the analysis, we obtain the results shown in Table 5. Figures 3 and 4 each depict one of these implementation approaches.

Table 5: Greatest feasible consistent implementation approaches obtained after adjusting the key architectural decisions. The architectural approach tuples reference alternatives for each KAD.

Arch. Appr.	Components
(1a, 2a, 3a)	$C_u - \{Qt, DirectX\}$
(1a, 2a, 3b)	$C_u - \{Qt, OpenGL, OGLFT, GLTT\}$
(1a, 2b, 3a)	$C_u - \{MFC, DirectX\}$
(1a, 2b, 3b)	$C_u - \{MFC, OpenGL, OGLFT, GLTT\}$
(1b, 2a, 3a)	$C_u - \{Ogre, Qt, DirectX\}$
(1b, 2a, 3b)	$C_u - \{Ogre, Qt, OpenGL, OGLFT, GLTT\}$
(1b, 2b, 3a)	$C_u - \{Ogre, MFC, DirectX\}$
(1b, 2b, 3b)	$C_u - \{Ogre, MFC, OpenGL, OGLFT, GLTT\}$

3.4 Choosing an Implementation Approach

To compare implementation approaches, we need to balance tradeoffs between the suitability of an architecture and the quality of components available in that architecture. A natural solution would be to evaluate architecture and components separately, then compare the results between each pair of implementation approaches. However, we run into a problem: sets of components from different implementation approaches can't be compared using the same criteria since they are meant for different architectures.

We need to evaluate architecture and components as a whole; our main concern is the effect an implementation approach has on the overall quality of the system. We define quality by specifying system-wide properties. These properties can be functional requirements (features), run-time attributes (e.g. performance, reliability, usability), or non-run-time attributes (e.g. cost, modifiability, scalability). These properties are essentially criteria used in a multi-criteria decision-making problem. The definition of these criteria is the subject of many existing papers on component selection [12, 6]

For each implementation approach, we evaluate the consequences of the architectural decisions and components chosen for each of the system-wide properties. This can be done informally using subjective judgment, or more rigorously by defining metrics (as in [20]). After evaluating each implementation approach on each property, a ranking of alternatives can be produced using a method for synthesizing multiple criteria like AHP [19].

In the *Reaction!* system, properties of interest include performance, portability, and extensibility. It is most important to analyze properties that discriminate between solutions. Portability is a good example, since some components we've identified are more platform independent than others. Consider the implementation approaches (1a, 2b, 3a) and (1b, 2a, 3b), shown in figures 3 and 4. The latter approach relies on MFC and DirectX, which are designed for Windows. The former approach uses Qt and OpenGL instead, which have cross-platform implementations. With all other properties being equal, the first approach would be preferred based on its better portability.

After obtaining the most preferred implementation approach, we can rely on existing component selection processes to choose specific components for each role (in this case only one component exists for each role). The result is a partial definition of the system's architecture, and a list of specific existing components that the system will use.

3.5 Conclusion

We have presented a systematic method for analyzing the relationship between components and architecture. The goal of the method is to discover the effect of chosen architectural decisions on the usable component base. By enumerating the components that go with different architectural options, we are able to arrive at a partial definition of the system's architecture and a set of corresponding components.

Our approach has inherent limitations. We rely on the ability of the analyst to discern the architectural assumptions each component makes. This is may be difficult, since many assumptions are implicit and not detectable early in the process [7].

With a large number of components to choose from and many potential ways to build the system, the practicality of the method is diminished due to the exponential growth of alternatives and comparisons. One way to address this problem is to create tool support for the method. Once criteria is defined and evaluated for each implementation approach, the comparisons that follow can be done automatically.

One premise of component-based software engineering is that requirements, architecture, and the selection of components are interrelated. Existing methods address the relationship between requirements and components, while we focus on the relationship between architecture and components. The integration of these methods is needed to address the complete problem.

We have included a trivial example to illustrate the concepts of the method. The next step is to validate it on a mid-sized system.

4. REFERENCES

[1] C. Alves and J. Castro. Cre: A systematic method for cots component selection. In *Brazilian Symposium on Software Engineering*, Rio De Janeiro, Brazil, October 2001.

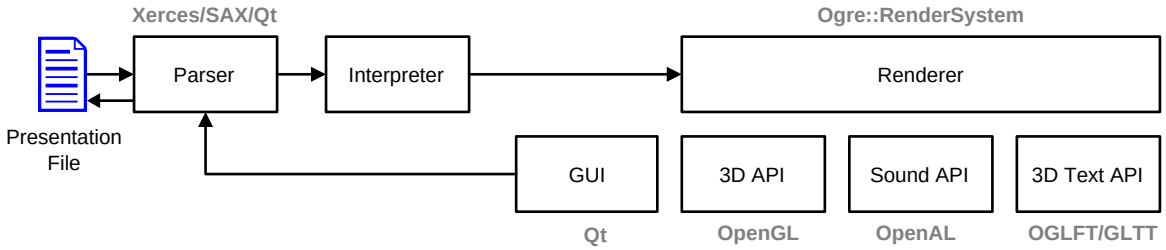

Figure 3: The implementation approach (1a, 2b, 3a). Off-the-shelf components that go with each role are shown in grey. In places where there is a choice between multiple components, we are free to interchange the given options.

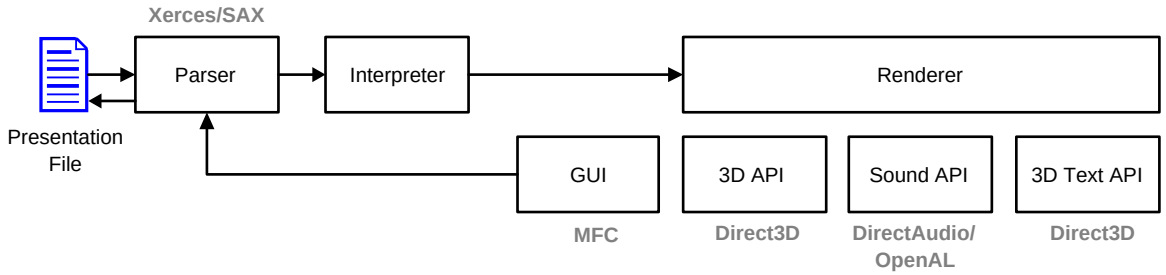

Figure 4: The implementation approach (1b, 2a, 3b). In this case, a generic custom component must be used to implement the renderer.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 2001.

[3] M. Bjorkander and C. Kobryn. Architecting systems with uml 2.0. *IEEE Software*, 20(4):57–61, July/August 2003.

[4] B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

[5] P. Bogetoft and P. Pruzan. *Planning with Multiple Criteria*. North-Holland, 1991.

[6] P. Botella, X. Burgues, J. P. Carvallo, et al. Towards a quality model for the selection of erp systems. *LNCS*, 2693:225–245, 2003.

[7] L. Brownsword, T. Oberndorf, and C. A. Sledge. Developing new processes for cots-based systems. *IEEE Software*, pages 48–55, July/August 2000.

[8] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Or why it's hard to build systems out of existing parts. In *Proc. Int'l Conf. Software Eng.*, pages 179–185, 1999.

[9] G. T. Heineman and W. T. Councill. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 2001.

[10] L. Iribarne, J. M. Troya, and A. Vallencillo. A trading service for cots components. *The Computer Journal*, 47(3):342–357, May 2004.

[11] R. Kazman, M. Klein, and P. Clements. Atam: Method for architecture evaluation. Technical Report CMU/SEI-2000-TR-004, CMU, August 2000.

[12] J. Kontio. Otso: A systematic process for reusable software component selection. Technical Report CS-TR-3478, December 1995.

[13] D. Kunda and L. Brooks. Applying social-technical approach for cots selection. In *UKAIS Conference*. University of York, April 1999.

[14] A. Lozano-Tello and A. Gomez-Perez. Baremo: How to choose the appropriate software component using the analytical hierarchy process. In *Proc. of the 14th Int'l Conf. on Software Eng. and Knowledge Eng.*, pages 781–788, 2002.

[15] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture desription languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000.

[16] C. Ncube and N. Maiden. Pore: Procurement-oriented requirements engineering method for the component based systems engineering development paradigm. In *Int'l Conf. Software Eng. CBSE Workshop*, 1999.

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

[18] A. O'Fallon, O. Pilskalkns, A. Knight, and A. Andrews. Defining and qualifying components in the design phase. In *Proc. of the 16th Int'l Conf. on Software Eng. and Knowledge Eng.*, June 2004.

[19] T. L. Saaty. *The Analytic Hierarchy Process*. McGraw-Hill, 1990.

[20] S. Sedigh-Ali, A. Ghafoor, and R. A. Paul. A metrics-guided framework for cost and quality management of component-based software. *LNCS*, 2693:374–402, 2003.

[21] M. Shaw. Architectural issues in software reuse: It's not just the functionality, it's the packaging. In *Proc. of the 1995 Symposium on Software Reusability*, pages 3–6, 1995.

[22] C. Syzperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.