Quantifying COTS Component Functional Adaptation

Alejandra Cechich[1] and Mario Piattini[2]

[1] Departamento de Ciencias de la Computación
Universidad Nacional del Comahue, Buenos Aires 1400
Neuquén, Argentina
acechich@uncoma.edu.ar
[2] Grupo Alarcos, Escuela Superior de Informática
Universidad de Castilla-La Mancha, Paseo de la Universidad 4
Ciudad Real, España
Mario.Piattini@uclm.es

Abstract. For successful COTS component selection and integration, composers increasingly look at software measurement techniques. However, determining the complexity of a component's adapter is still an ongoing concern. Here, a suite of measures is presented to address this problem within a COTS-based software measurement activity. Our measures are based on a formally defined component-based model, aiming at expressing and measuring some aspects of component adaptations.

Keywords: Component-based system assessment. COTS components. Software Quality. Metrics.

1 Introduction

The rigorous measurement of reuse will help developers determine current levels of reuse and help provide insight into the problem of assessing software that is easily reused. Some reuse measures are based on comparisons between length or size of reused code and the size of newly written code of software components [12]. Particularly, measurement programs can facilitate incorporating an engineering approach to component-based software development (CBSD), and specifically to COTS component selection and integration, giving composers a competitive advantage over those who use more traditional approaches.

Measurements let developers identify and quantify quality attributes in such a way that risks encountered during COTS selection are reduced. Then, measurement information might be structured as the proposal in [11], in which a methodology facilitates the evaluation and improvement of reuse and experience repository systems by iteratively conducting goal-oriented measurement programs. However, most cost estimates for CBS developments are based on rules of thumb involving some size measure, like adapted lines of code, number of function points added/updated, or more recently, functional density [1,9].

J. Bosch and C. Krueger (Eds.): ICSR 2004, LNCS 3107, pp. 195–204, 2004.
© Springer-Verlag Berlin Heidelberg 2004

To address this problem, in a previous work [4] we have adapted the model introduced in [2], which explores the evaluation of components using a specification-based testing strategy, and proposes a semantics distance measure that might be used as the basis for selecting a component from a set of candidates.

By adapting this model, we have set a preliminary suite of measures for determining the functional suitability of a component-based solution. However, our measures are based on functional direct connections, i.e. there is no semantic adaptation between the outputs provided by a concrete component and its required functionality. The importance of defining functional adaptability measures comes from the importance of calculating the tailoring effort during COTS component integration. When analysing components, it is usually the case that the functionality required by the system does not semantically match with the functionality provided by the candidate components. Detecting additional or missed functionality is a more common case instead.

In this paper, we are extending our previous suite of measures to quantify the components' functional adaptability, in such a way that our measures may be combined to or used by some other approaches.

In section 2 of the paper, we introduce the component-based model for measurement (from [2]) along with a motivating example. Then, section 3 presents a suite of measures for determining the degree in which a component solution needs adaptation. Section 4 presents two possible applications of our proposal by combining some related works. Finally, section 5 addresses conclusions and topics for further research.

2 An Adaptation Model for Measurement

At the core of all definitions of software architecture is the notion that the architecture of a system describes its gross structure, including things such as how the system is composed of interacting parts, where are the main pathways of interaction, and what are the key properties of the parts [13].

Components are plugged into a software architecture that connects participating components and enforces interaction rules. For instance, the model in [2] supposes that there is an architectural definition of a system, whose behaviour has been depicted by scenarios or using an architecture description language (ADL), which usually provides both a conceptual framework and a concrete syntax for characterising software architectures.

The system can be extended or instantiated through the use of some component type. Due several instantiations might occur, an assumption is made about what characteristics the actual components must possess from the architecture's perspective. Thus, the specification of the architecture A (S_A) defines a specification S_C for the abstract component type C (i.e. $S_A \Rightarrow S_c$). Any component K_i, that is a concrete instance of C, must conform to the interface and behaviour specified by S_C. The process of composing a component

K with A is an act of interface and semantic mapping. In this work, only the latter will be addressed.

During the mapping, it can be the case that the semantics of K are not sufficient for S_C (i.e., $\neg(S_K \Rightarrow S_C)$. In this situation, K is somehow lacking with respect to the behavioural semantics of C. The possibility is that K has partial behavioural compatibility with C. In this case, K either has incompatible or missing behaviour with respect to some portion of S_C. To overcome this, a semantic adapter $A_{S_C}^{S_K}$ must be specified (and built) such that, when composed with S_K, the adapter yields a component that is compatible with C [2].
The composition of this specification, $A_{S_C}^{S_K}$, and S_K must satisfy the $(A_{S_C}^{S_K} \circ S_K) \Rightarrow S_C$, as shown in Figure 1 (from [2]). The dashed line indicates that the adapter may provide some of the behavioural semantics if the component K is somehow deficient.

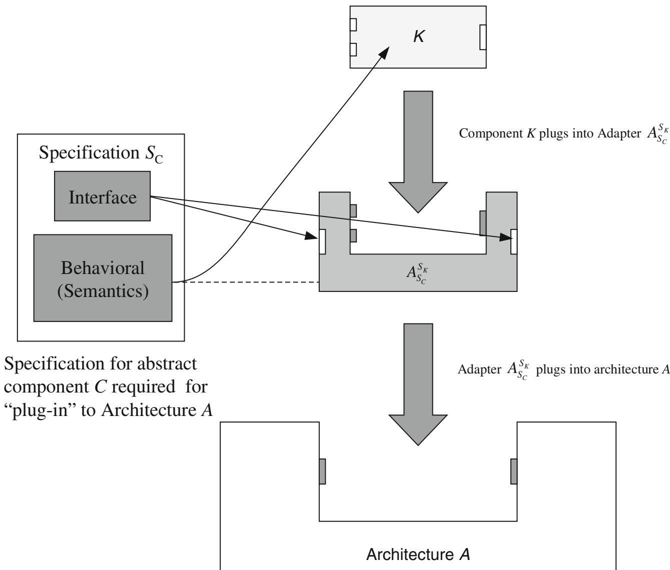

Fig. 1. Component K adapted for use in architecture A by adapter $A_{S_C}^{S_K}$ (from [2])

According to the work in [2], a number of issues arise when considering what behaviour $A_{S_C}^{S_K}$ must have. Firstly, all inputs in the domain of S_C that are not included in the domain of S_K must be accounted for by $A_{S_C}^{S_K}$ and likewise for the outputs in the range of S_C, i.e. the domain and range of the aggregate must at least include that of S_C. Given that the domain and range of $A_{S_C}^{S_K} \circ S_K$ is consistent with S_C, the adapter $A_{S_C}^{S_K}$ must include those mappings from S_C

that are not supported by S_K. Essentially, $A_{S_C}^{S_K}$ must provide those mappings whose domain is in S_C but not S_K, and those mappings whose domain in both S_C and S_K but where the element mapped to in the range of S_C is not the same as the element mapped to in the range of S_K. These mappings are described formally in [2] as [1]:

$$mapping(A_{S_C}^{S_K}) = \{(i,j) \mid (i \in (dom(S_C) \setminus D_{S_C}^{S_K}) \wedge S_C(i) = j) \vee (i \in D_{S_C}^{S_K} \wedge S_K(i) \neq S_C(i) \wedge S_K(i) = j)\}$$

Secondly, all mappings not included in S_C and additionally provided by S_K should be hidden by $A_{S_C}^{S_K}$ to simplify the integration, i.e. $A_{S_C}^{S_K}$ must hide those mappings whose domain is in S_K and not in S_C and the element mapped to in the range of S_K is not in the range of S_C . These mappings can be described formally as:

$$added(A_{S_C}^{S_K}) = \{(i,j) \mid i \in (dom(S_K) \setminus D_{S_C}^{S_K}) \wedge S_K(i) = j \wedge j \notin rng(S_C)\}$$

Finally, all mappings included in S_C and provided by S_K constitute the functionality provided by the component $\mathcal{K}$. These mappings can be described formally as:

$$funct(F_{S_C}^{S_K}) = \{(i,j) \mid i \in D_{S_C}^{S_K} \wedge S_K(i) = j \wedge S_K(i) = S_C(i)\}$$

2.1 A Motivating Example: Credit Card E-payment

Authorisation and *Capture* are the two main stages in the processing of a card payment over the Internet. Authorisation is the process of checking the customer's credit card. Capture is when the card is actually debited.

We suppose the existence of some scenarios describing the two main stages, which represent here a credit card (CCard) payment system. The scenarios will provide an abstract specification of the input and output domains of S_C that might be composed of:

- Input domain: (AID) Auth_IData{#Card, Cardholder_Name, Exp_Date, Bank_Acc, Amount}; (CH) Cardholder_ID;
  (CID) Capture_IData {Bank_Acc, Amount}.
- Output domain: (AOD) Auth_OData{ok_Auth}; (CHC) Cardholder_Credit; (COD) Capture_OData{ok_capture, DB_update}.
- Getting Authorization: $\{AID \mapsto AOD\}$.
- Calculating Credit: $\{CH \mapsto CHC\}$.
- Capture: $\{CID \mapsto COD\}$.

Suppose we pre-select two components to be evaluated, namely K_1 and K_2 respectively. However, the specification mapping, shown in Figure 2, reveals some inconsistencies that should be analysed.

[1] Comparison between ranges has been simplified by considering equality. A more complex treatment of ranges might be similarly specified, for example, by defining a set of data flows related by set inclusion.

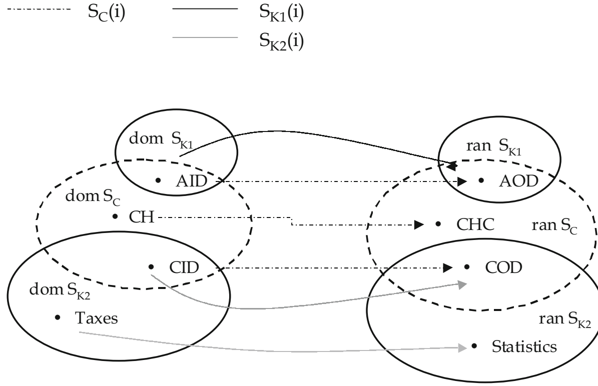

Fig. 2. Functional mappings of S_C and S_{K1}/S_{K2} (from [2])

3 A Measurement Suite for Functional Adaptability

For the measure definitions, we assume a conceptual model with universe of scenarios $\mathcal{S}$; an abstract specification of a component $\mathcal{C}$; a set of components $\mathcal{K}$ relevant to $\mathcal{C}$, and a mapping component diagram. In the following definitions, we use the symbol # for the cardinality of a set. To simplify the analysis, we also assume input/output data as data flows, i.e. data that may aggregate some elemental data. For the credit card example, input/output data are represented by {AID, CH, CID}, {AOD, CHC, COD} respectively.

3.1 Implemented Functional Adaptability Measures

Table 1 lists the proposed measures for measuring functional adaptability cases. The measures have been grouped into two main groups: component-level measures and solution-level measures. The first group of metrics aims at detecting incompatibilities on a particular component $\mathcal{K}$, which is a candidate to be analysed. For example, $EF_{A_{S_C}^{S_K}}$ aims at measuring the number of functions that are added when implementing the adapter of the component $\mathcal{K}$ (extended functionality).

However, we could need to incorporate more than one component to satisfy the functionality required by the abstract specification $\mathcal{C}$. In this case, the second group of metrics evaluates the functional adaptability of all components that constitute the candidate solution $\mathcal{SN}$. For example, $HF_{A_{S_C}^{S_{SN}}}$ aims at measuring the number of functions hidden when implementing the adapter/s of the solution $\mathcal{SN}$.

It is important to note that the amount of functionality implemented by the adapter depends on a design decision, that is, $EF_{A_{S_C}^{S_K}}$ does not represent the

Measure Id.	Description	Measure Definition
Component-level		
$EF_{A_{S_C}^{S_K}}$ Extended Functionality	The number of functional mappings provided by $A_{S_C}^{S_K}$ NOT in S_K and required by S_C in the scenario $\mathcal{S}$.	$\#\{AM(A_{S_C}^{S_K})\}$
$HF_{A_{S_C}^{S_K}}$ Hidden Functionality	The number of functional mappings in S_K, NOT required by S_C in the scenario $\mathcal{S}$, and hidden by $A_{S_C}^{S_K}$.	$\#\{HM(A_{S_C}^{S_K})\}$
$AC_{A_{S_C}^{S_K}}$ Adapter Contribution	Percentage in which a component adapter contributes to get the functionality required by S_C in the scenario $\mathcal{S}$.	$\frac{EF_{A_{S_C}^{S_K}}}{\#(mapping(A_{S_C}^{S_K}))}$
$HC_{A_{S_C}^{S_K}}$ Hidden Contribution	Percentage in which a component adapter contributes to hide the functionality provided by S_K and NOT required by S_C.	$\frac{HF_{A_{S_C}^{S_K}}}{\#(added(A_{S_C}^{S_K}))}$
Solution-level		
$EF_{A_{S_C}^{S_N}}$ Extended Functionality	The number of functional mappings not in $\mathcal{SN}$, provided by the adapters and required by S_C in the scenario $\mathcal{S}$.	$\#\{(\dagger AM(A_{S_C}^{S_{K_i}}))_{\forall K_i \in SN} \setminus (\dagger funct(F_{S_C}^{S_{K_i}}))_{\forall K_i \in SN}\}$
$HF_{A_{S_C}^{S_N}}$ Hidden Functionality	The number of functional mappings NOT required by S_C in the scenario $\mathcal{S}$, provided by $\mathcal{SN}$ and hidden by the adapters.	$\#\{(\dagger HM(A_{S_C}^{S_{K_i}}))_{\forall K_i \in SN}\}$
$AC_{A_{S_C}^{S_N}}$ Adaptation Contribution	Percentage in which the adapters of a solution contribute to get the functionality required by S_C in the scenario $\mathcal{S}$.	$\frac{EF_{A_{S_C}^{S_N}}}{\#\{(\dagger mapping(A_{S_C}^{S_{K_i}}))_{\forall K_i \in SN} \setminus (\dagger funct(F_{S_C}^{S_{K_i}}))_{\forall K_i \in SN}\}}$
$SC_{A_{S_C}^{S_N}}$ Simplicity Contribution	Percentage in which the adapters of a solution contribute to hide the functionality NOT required by S_C in the scenario $\mathcal{S}$.	$\frac{HF_{A_{S_C}^{S_N}}}{\#\{(\dagger added(A_{S_C}^{S_{K_i}}))_{\forall K_i \in SN}\}}$

Table 1. Description of the Functional Adaptability measures

number of tuples of the mapping $A_{S_C}^{S_K}$, but the number of tuples (functions) actually added by the adapter. Similarly, $HF_{A_{S_C}^{S_{SN}}}$ represents the number of tuples (functions) actually hidden by the adapter. Therefore, we define the notions of *Added Mapping* and *Hidden Mapping* as follows:

$$AM(A_{S_C}^{S_K}) = \{(i,j) \mid (i,j) \in mapping(A_{S_C}^{S_K}) \wedge (A_{S_C}^{S_K})(i) = j \wedge S_C(i) = j\}$$

$$HM(A_{S_C}^{S_K}) = \{(i,j) \mid (i,j) \in added(A_{S_C}^{S_K}) \wedge (A_{S_C}^{S_K})(i) = \{\}\}$$

Then, contribution measures are defined to measure the completeness of the adapters' functionality. For example, $AC_{A_{S_C}^{S_K}}$ aims at measuring the percentage in which a component adapter contributes to get the extended functionality required by S_C in the scenario $\mathcal{S}$. A more formal definition of the measures is shown in Table 1.[2]

Let's compute the measures for our credit card example, where:

$\text{mapping}(A_{S_C}^{S_{K_1}}) = \{CH \mapsto CHC; CID \mapsto COD\}$,
$\text{added}(A_{S_C}^{S_{K_1}}) = \{\}$,

$\text{mapping}(A_{S_C}^{S_{K_2}}) = \{CH \mapsto CHC; AID \mapsto AOD\}$, and
$\text{added}(A_{S_C}^{S_{K_2}}) = \{Taxes \mapsto Statistics\}$.

In a first scenario, suppose that all functionality is implemented by the adapters; i.e.
$AM(A_{S_C}^{S_K}) = mapping(AM(A_{S_C}^{S_K}))$ and $HM(A_{S_C}^{S_K}) = added(AM(A_{S_C}^{S_K}))$.

Hence, component-level measures for $\mathcal{K}1$ and $\mathcal{K}2$ are as follows:

$$EF_{A_{S_C}^{S_{K_1}}} = 2; HF_{A_{S_C}^{S_{K_1}}} = 0;\ EF_{A_{S_C}^{S_{K_2}}} = 2; HF_{A_{S_C}^{S_{K_2}}} = 1$$

The values of the measures show that selecting the component $\mathcal{K}1$ implies developing adaptation for adding two functions and no adaptation for hiding side functionality. On the other hand, selecting the component $\mathcal{K}2$ might lead in implementing the same number of functions (not necessarily the same amount of functionality), but it also implies hiding one function (the one represented by the map (*Taxes, Statistics*)). So, a balance is struck to decide on selecting a set of COTS candidates as a solution, selecting only one component, or developing the solution from scratch.

Let's suppose that we decide to select a set of COTS components suggesting the solution $\mathcal{SN}$ as the set $\{\mathcal{K}1, \mathcal{K}2\}$. Then, to calculate $EF_{A_{S_C}^{S_{SN}}}$ the override operation, $\dagger AM_{A_{S_C}^{S_{K_i}}} 1 \leq i \leq n$, is explicitly expressed as the calculation of

[2] † represents traditional map overlapping. Example:$\{3 \mapsto true, 5 \mapsto false\} \dagger \{5 \mapsto true\} = \{3 \mapsto true, 5 \mapsto true\}$.

$\{CH \mapsto CHC; CID \mapsto COD\} \dagger \{CH \mapsto CHC; AID \mapsto AOD\}$. It results in $\{CH \mapsto CHC; CID \mapsto COD; AID \mapsto AOD\}$, which implicitly states a selection when the same functionality might be provided by more than one adapter in the solution $\mathcal{SN}$. [3]

Then, we continue computing the solution-level measures for our example as follows:

$$\{(\dagger AM(A_{S_C}^{S_{K_i}}))_{\forall K_i \in SN} \setminus (\dagger funct(F_{S_C}^{S_{K_i}}))_{\forall K_i \in SN}\} =$$
$$\{CH \mapsto CHC; CID \mapsto COD; AID \mapsto AOD\} \setminus \{CID \mapsto COD; AID \mapsto AOD\} = \{CH \mapsto CHC\}$$
$$EF_{A_{S_C}^{S_{SN}}} = 1; HF_{A_{S_C}^{S_{SN}}} = 1$$

In this case, the adapter/s implement all the requirements, i.e. adding the missed mapping $\{CH \mapsto CHC\}$, and hiding the mapping $\{Taxes \mapsto Statistics\}$; hence $AC_{A_{S_C}^{S_{SN}}}$ is equal to 1 (its highest possible value). But in a more complex case, we could have decided not to add some missed functionality. Therefore, in this case the value of $AC_{A_{S_C}^{S_{SN}}}$ would be less than 1, indicating incompleteness for the adapter/s at the implemented-level. In this case, calculating $SC_{A_{S_C}^{S_{SN}}}$ would be also meaningful.

4 Related Works: Possible Uses of the Measures

Complexity of adaptability. Focusing on adapters, each extended function implies interaction with target components, which must be identified to determine all potential mismatches. For example, a component may try to access data that are considered private by the target component. This mismatch detection can be performed on every interface connection using the procedure defined by Gacek [8]. Once the connection type is known, such as call, spawn, or trigger, then all of the mismatches associated with that connection type are potential mismatches for the connection. Then, for each mismatch, potential resolution techniques might be considered from the proposal by Deline [7], where a weighting factor is assigned to each connection to describe the difficulty with which the solution can be implemented.

Now, we may associate a resolution complexity factor to each extended function of our model providing additional information, so that an appropriate choice can be made. For the E-payment example, there is one function added by the adapter at the solution-level: Calculating Credit. We suppose that there is a mismatch associated with this connection and the mismatch resolution technique is *wrapper* for this case; hence the relative complexity (RelCplx) will be 6 (from [7]). When more than one mismatch is associated to the same connection, or when there is more than one connection analysed for the same adapter, we

[3] Note that calculating $\dagger funct(F_{S_C}^{S_{K_i}})$ also implies a selection when more than one component is able to provide the same required functionality.

suggest the Functional Adaptability Complexity (FAC) – related to the adapter $A_{S_C}^{S_{SN}}$ – as the sum of all individual connection complexities on the table.

We should note here that the PIC Software Productivity Consortium's project [3], has recently determined an estimate for the effort required to integrate each potential component into the existing system architecture. The estimate of integration complexity considers several factors and the mismatch resolution involves semantic as well as syntactic adaptation. Therefore, our work might be considered as a more specific proposal that could be used along with other current research efforts on measuring COTS component integration, such as the BASIS approach.

Measuring architectural adaptability. Adaptability of an architecture can be traced back to the requirements of the software system for which the architecture was developed. The POMSAA (Process-Oriented Metrics for Software Architecture Adaptability) framework [6], achieves the need of tracing by adopting the NFR framework that is a process-oriented qualitative framework for representing and reasoning about NFRs (non-functional requirements) [4].

In the NFR Framework, the three tasks for adaptation become softgoals to be achieved by a design for the software system. An adaptable component of a software system should satisfice these softgoals for adaptation. Another point to be observed is that design softgoals are decomposed in a manner similar to the decomposition of the NFR softgoals. One of the softgoals to be decomposed is *adaptability*, which can be further described in terms of semantic adaptability, syntactic adaptability, contextual adaptability and quality adaptation.

Our proposal suggests analysing each branch of the hierarchy of *semantic adaptability* of the NFR softgoal graph in terms of complexity and size, as we have previously defined. In this way, qualitative judgments on architectural adaptability would be based on more precise and objective values. After that, the architectural adaptability will characterise system's stability at the higher level, conceptualised in terms of its functionalities for system's users.

5 Conclusions and Future Work

We have presented a preliminary suite of measures for determining the functional adaptability of a component-based solution. The suite of measures might be integrated to other approaches – such as BASIS – and the final calculation might be applied to ponderate architectural decisions when calculating measures for architectural adaptability – such as the work in [6] suggests.

Of course, there are some points for further research. On one hand, we should note that our measures are based on counting functional mappings, and their domains – specified by different levels of abstraction – could distort the final measure. Then, a more formal specification of input/output values as well as the relationships between them would reduce ambiguity when calculating the measures. At this point some related works might be helpful, such as the proposal in

[4] For more detailed description of NFRs, we refer the reader to [5]

[10], which measures a refinement distance by measuring both the requirements of $\mathcal{SC}$ that are left unfulfilled by $\mathcal{K}$ as well as the functional features of $\mathcal{K}$ that are irrelevant to $\mathcal{SC}$. On the other hand, functional mappings come from scenarios that also need further discussion on their generation and documentation.

Finally, our measures and the procedure need further validation. In order to demonstrate the applicability of our proposal, some empirical studies are currently carried out on the domain of E-payment systems.

Acknowledgments. This work is partially supported by the CyTED project VII-J-RITOS2, by the UNComa project 04/E048, and by the MAS project(TIC 2003-02737-C02-02).

References

1. C. Abts. COTS-Based Systems (CBS) Functional density - A Heuristic for Better CBS Design. In *Proceedings of the First International Conference on COTS-Based Software Systems*, volume 2255 of *LNCS*, pages 1–9. Springer, 2002.
2. R. Alexander and M. Blackburn. Component Assessment Using Specification-Based Analysis and Testing. Technical Report SPC-98095-CMC, Software Productivity Consortium, 1999.
3. K. Ballurio, B. Scalzo, and L. Rose. Risk Reduction in COTS Software Selection with BASIS. In *Proceedings of the First International Conference on COTS-Based Software Systems*, volume 2255 of *LNCS*, pages 31–43. Springer, 2002.
4. A. Cechich and M. Piattini. On the Measurement of COTS Functional Suitability. In *Proceedings of the Third International Conference on COTS-Based Software Systems*, volume 2959 of *LNCS*. Springer, 2004.
5. L. Chung, B. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publisher, 2000.
6. L. Chung and N. Subramanian. Process-Oriented Metrics for Software Architecture Adaptability. In *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering (RE '01)*, pages 310–312, 2001.
7. R. Deline. A Catalog of Techniques for Resolving Packaging Mismatch. In *Proceedings of the Symposium on Software Reusability, Los Angeles, CA*, 1999.
8. C. Gacek. Detecting Architectural Mismatches During System Composition. Technical Report USC/CSE-97-TR-506, University of Southern California, 1997.
9. L. Holmes. Evaluating COTS Using Function Fit Analysis. Q/P Management Group, INC - http://www.qpmg.com.
10. L. Jilani and J. Desharnais. Defining and Applying Measures of Distance Between Specifications. *IEEE Transactions on Software Engineering*, 27(8):673–703, 2001.
11. M. Nick and R. Feldmann. Guidelines for Evaluation and Improvement of Reuse and Experience Repository Systems Through Measurement Programs. In *Proceedings of the Third European Software Measurement Conference*, 2000.
12. R. Selby. *Quantitative Studies of Software Reuse.* In Software Reusability Vol II Applications and Experiences, T. Biggerstaff and A. Perlis (Eds.). Addison Wesley, 1989.
13. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.