



Simple Component Interface Language (SCIL) - A Lightweight Formal Approach for Software Component Retrieval

Vishnu Sharma¹, Vijay Singh Rathore²

Research Scholar, Mewar University, Chittorgarh, Rajasthan, INDIA.
Director, Karni College, Jaipur, Rajasthan, INDIA

Abstract: In order to write an unambiguous component specification, one should include semantics of the interface model and the interaction rules with its environment in the specification. Using heavyweight formal methods can serve the purpose, but it is not practical because there will not be many people who have good mathematical skills. The main focus of this paper is to suggest an efficient Simple Component Interface Language (SCIL), a lightweight formal approach to achieving both precision and practicality. That is to say, SCIL has easy to understand syntax, but can precisely describe components. It is unrealistic for a specification language to capture everything, including signatures, behaviours and other details. Thus SCIL only focuses on the high-level behaviors of components.

Keywords: Component Based Development (CBD), Simple Component Interface Language (SCIL), Component Off The Shelf (COTS), Unified Modeling Language (UML), Object Constraint Language (OCL)

I. Introduction

Today, component-based software reuse is considered as one of the hallmarks that will bring software engineering more likely to the standard of a full-fledged engineering discipline [1]. In spite of this, the component based market and the software reuse are not meeting with their expected breakthrough as required. Drastic growth of software systems in size and complexity, the software components have been proposed as the key technology to address this challenge. By enabling reuse, components permit one to rely on the subsystems developed by other developers to modify system design and implementation. This reduces the effort and time needed to develop the overall system. In more general term, by using components the building of complex systems can be simplified, so that outcome can be increased. But component selection is not that much easy, it is very critical because the other development steps will depend on it. A wrong selection of the components can cause the failure of the entire system [2]. Of the current search engines for software components, whether a general search engine such as Google, or a particular search engine for a component repository such as the one for Component Source [3], only a free text-based search is supported. We consider a free text-based search as the current practice of searching components because these search engines are the most widely used. For example, Component Source [3] is the most popular component repository in the world, providing a large variety of components from different developers.

II. Problem Statement

The results obtained by this text-based search approach have exposed its weaknesses. First, except for the free text-based searching, most selection work is manual. For example, one often receives a long list of candidate components to it takes a lot of time to decide their relevance. Testing components is also time-consuming. Secondly, even with so much of time spent, the component retrieved still cannot meet the requirement of user. It happens because of manual work that easily causes mistakes. If happens, adapting the component at a later stage would be more difficult. The idealized CBD process assumes that the components obtained from other developers are sufficiently close to the units identified when decomposing the system that is being developed, so that the component adaptation requires less effort than the unit implementations. However, this is rarely the case in reality, especially when the components are not in-house developed, for example, Commercial Off-The-Shelf (COTS) components. People often cannot get the required components. Thus adapting the components into the system becomes an extra development and maintenance cost [4]. This additional cost may trade off the benefits of CBD. This is the problem of component selection, especially when selecting from COTS components. Therefore it is very difficult to get the user-required components with imprecise specifications. Furthermore, this problem is difficult to solve due to the nature of CBD, which involves two separate development processes: develop components and use components. These two development processes do not have an obvious connection to each other. That is to say, even if the misunderstandings on components or requirements exist, it is hard to get them clarified. Meanwhile, there are a large variety of components, but these components are not tailor made to the particular requirements. Thus in reality it is very rare that component users can get an exactly wanted component without any adaptation when integrating it into the system. However, if component developers can

get involved in component selection by customising their products based on user requirements, the possibility of finding and retrieving the required components can be increased.

III. Literature Survey

Some work has been done to extend the syntactic approaches by adding invariants and pre/post condition pairs to constrain component behaviours, such as JML [5], Spec# [6]. Unified Modeling Language (UML) [7] is a semi-formal modeling language, widely used in design and documentation. Integrated with Object Constraint Language (OCL) [8], UML can also describe component behaviours using the existing notations. Combining such techniques with existing programming languages often results in a fairly complicated and very detailed specification. One is then unable to use such a specification to retrieve components for a particular situation.

Another problem of these approaches is that it is difficult to specify when an operation should be invoked. When describing the interactions among components, the order by which the operations of the components should be invoked is important. Therefore, specification of a complete component needs to include temporal properties of component behavior.

To specify component behaviors, one can use formal description including semantics and interaction rules of component interfaces at an abstract, but sufficiently precise and complete level. Formal descriptions allow the testing of consistency and accuracy of the interface models. Whereas formal methods successfully used to specify behavioral properties, such as those described in [4, 9, 10, 11, 12, 13, 14, 15], and tools have been developed to verify these properties, they are still not well-known with professionals. Only reason is that formal methods need a strong background in mathematics which is currently unrealistic to expect normal component users and developers to have such a background. That is why we cannot find existing component repositories providing formal specifications of components.

Several classification schemes such as keywords [16], enumerated [17], faceted [18] and hypertext [19] can facilitate the user's search of components. However, component users may find it difficult to take advantages of such classification schemes if they do not know the vocabularies that are used to build the schemes [20]. One way to overcome this limitation is to define a natural language user interface [21], by which the users input queries in natural languages, such as English. The input queries will be examined and decomposed into the previously defined classification vocabularies. Using retrieval techniques through classification-based component can help to filter irrelevant components, but cannot assure that the components retrieved will have the expected behaviours. This is because classification schemes do not specify the components behavioural properties. A further examination of the retrieved components is required.

With specifications, component matching is performed upon signatures [22] and behaviours [23]. Signature matching is also mostly syntactic and therefore it is tough to receive expected results. Behavior matching relies on semantics and interaction protocol of the interface model, so it can enhance the possibility of the component retrieved being integrated and working as expected in the target system.

Most research work in component evaluation focuses on the evaluation processes [24, 25, 26] that are driven by models, which include the product descriptions and evaluation criteria, and the particular methods such as multiple attribute utility [27] or component ranking [28]. The lessons learned from [29] have told us that component evaluation is not easy.

To minimize the additional cost of component adaption, like as writing glue code, components need to be tailor made to user requirements. The research performed in [4] has suggested that writing glue code typically takes longer and requires more effort to complete than tailoring. This may be because the intellectual effort required to simply configure (or tailor) a given COTS product is usually less than that required to create code around it that is not only new, but also highly constrained – the situation that exists with glue code.

IV. Collaboration Process

The in-house developed components only can be tailor made to user requirements. Since the components are built by other internal developers, the component user can get access to all the component documents, including the source code. The process of selecting components becomes teamwork within the same organisation. These components can be tailor made to the requirements. Even if the components cannot meet user requirements, the developers can easily modify the component implementations to suit user needs. This is because the communication within the same organization is easy. If the components are built from external developers, the communication between component users and developers becomes difficult. However, the teamwork still can be done if two conditions are met: first, it should be in the interests of the external developers in assisting component users to find and retrieve components; second, there should be technologies and tools supporting the collaboration. These two conditions are not necessary when components are tailor made to the user requirements. For communication purposes, a common language is needed by both component users and developers to exchange information. The information can be the specifications of user requirements and components. When determining whether components satisfy user requirements, one should be able to use model checking tools to check the specifications. These tools should be available for both sides of the collaboration. By allowing the exchange of requirements and component descriptions, the collaboration can help component users

to make their requirements clearer, and can ensure component developers deliver suitable components to users. The collaboration can also solve the problem of being unable to find suitable components, because either component users can adjust their requirements, or component developers can provide additional choices by customizing their products.

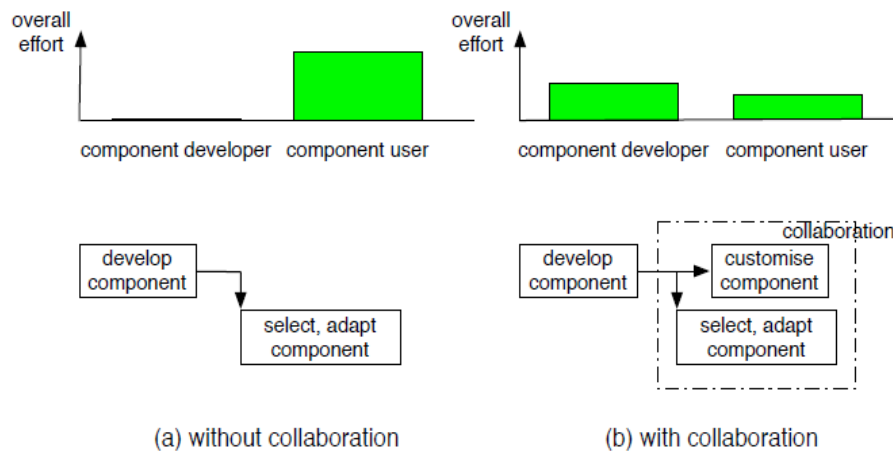


Figure 1: Collaboration process

In order to write an unambiguous component specification, one should include semantics of the interface model and the interaction rules with its environment in the specification. Using heavyweight formal methods can serve the purpose, but it is not practical because there will not be many people who have good mathematical skills. Few traditional approaches to get best qualified component are:

1. **Keyword search** requires assigning to each object a number of relevant keywords or indices [30].
2. **Full-text Retrieval** : when a person wants information from that stored collection, the computer is instructed to search for all documents containing certain specified words and word combinations, which the user has specified [30].
3. **Hypertext Search**: The basic building blocks in hypertext are nodes and links. Each node is associated with a unit of information, and nodes can be of different types[30].
4. **Enumerated classification** : Enumerated classification uses a set of mutually exclusive classes, which are all within a hierarchy of a single dimension [31].
5. **Attribute value** : The attribute value classification scheme uses a set of attributes to classify a component [31].
6. **Faceted** : Faceted classification schemes are attracting the most attention within the software reuse community [31, 32].
7. **Signature matching** : Consider the signatures for a stack of integers and a queue of integers, respectively [33]

Deduction-based software component retrieval uses pre and post conditions as indexes and search keys and an automated theorem prover (ATP) to check whether a component matches. This idea is very simple but the vast number of arising proof tasks makes a practical implementation very hard. We thus pass the components through a chain of filters of increasing deductive power. In this chain, rejection filters based on signature matching and model checking techniques are used to rule out non-matches as early as possible and to prevent the subsequent ATP from “drowning.” Hence, intermediate results of reasonable precision are available at (almost) any time of the retrieval process. The final ATP step then works as a confirmation filter to lift the precision of the answer set. We implemented a chain which runs fully automatically and uses MACE for model checking and the automated prover SETHEO as confirmation filter. We evaluated the system over a medium-sized collection of components. Whenever different ways for achieving an objective keep their pros and cons than a perfect solution is, which keeps pros of all of these. It is mixed approach.

V. Simple Component Interface Language (SCIL)

As a solution, we define Simple Component Interface Language (SCIL), a lightweight formal approach to achieving both precision and practicality. That is to say, SCIL has easy to understand syntax, but can precisely describe components. It is unrealistic for a specification language to capture everything, including signatures, behaviours and other details. Thus SCIL only focuses on the high-level behaviours of components. There exists a variety of modeling languages that rely on different formalism to support specifying different behavioural properties. If these languages with their tools are used as complements to each other, more behavioural properties can be checked.

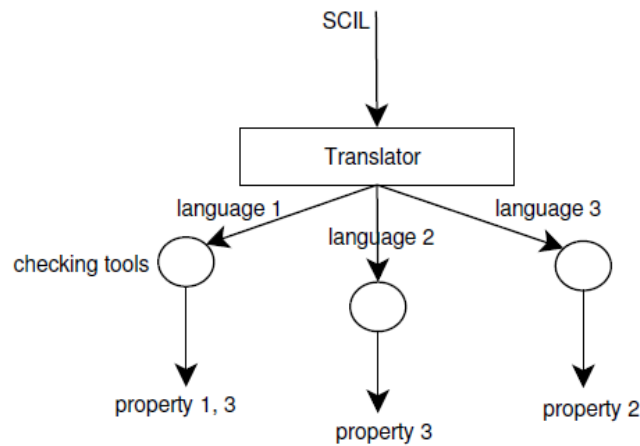


Figure 2: Using SCIL as the Bridge to Checking Various Properties

SCIL acts as a bridge to access these previously developed tools. For example, in figure 2 different languages can be used to check different properties. By translating SCIL to a variety of modeling languages it is possible for SCIL users to access all the properties that cannot be checked by a single tool. Furthermore, it allows users who do not have a mathematical background to use such tools indirectly. In the collaboration framework, SCIL is also used as the communication language. If component users and component developers use different languages to describe requirements and component products, communication would become difficult. A software component contains a set of interfaces, and the encapsulated implementations of these interfaces, which cannot be straightly retrieved from its environment. The separation of interfaces and their implementations makes it possible to either add new interfaces (with new implementations) without changing existing implementations, or replace old implementations with new ones without affecting their interfaces. A software component is produced to be composed with other components and deployed in a container. This is done by any person other than its developer through the component interfaces and the contracts attached to these interfaces without the knowledge of the component's internal workings.

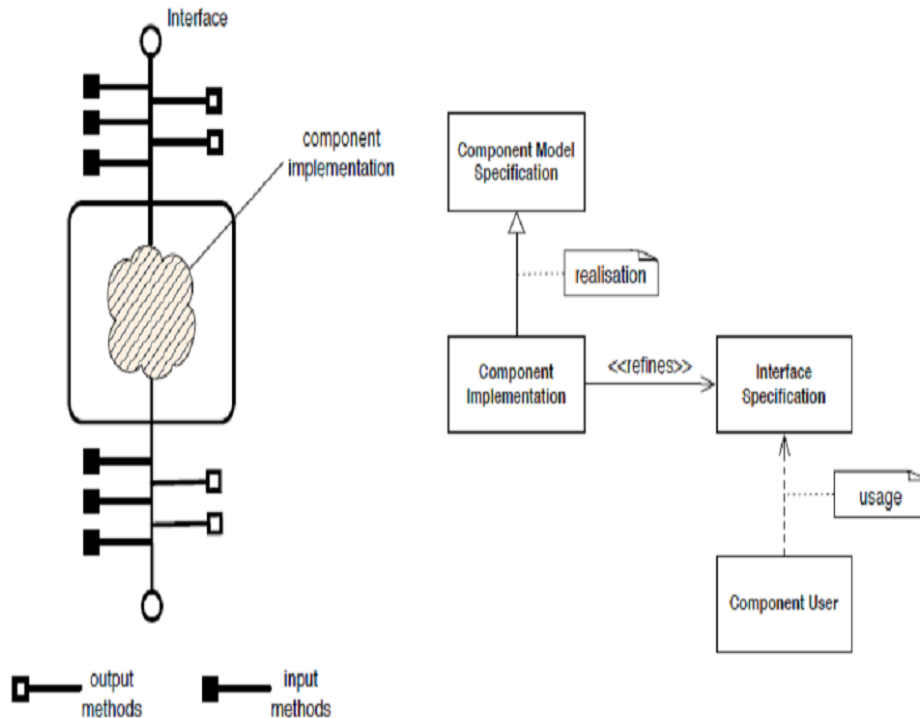


Figure 3: Interfaces, Methods, Contracts and Specifications

Figure 3 shows the correlation between Interfaces, Methods, Contracts and Specifications. Specifications of interface are simpler, but consists enough information to determining how the underlying components are composed. The specifications of component do not make assumptions on their working environment. For more discussion on the differences can be easily seen in [34]. The specification of component is also surprisingly used

in some literature to mean component model specification, which contains of component model, support infrastructure, implementation framework, packaging and deployment models. The classical examples of component model specification include Sun's JavaBean [35], Microsoft's Component Object Model (COM) [36], OMG's CORBA Component Model (CCM) [37], and Enterprise JavaBean (EJB) [38]. Following concept model may be presented for a efficient component retrieval process. This problem has been explained in different viewpoints. Therefore lots of concepts linked to component selection have been presented. Figure 3 drew a concept map that displays how these concepts can support the components selection. Among these concepts, we focus more on the existing techniques of specifying component interfaces, storing and retrieving components, as well as assessing components.

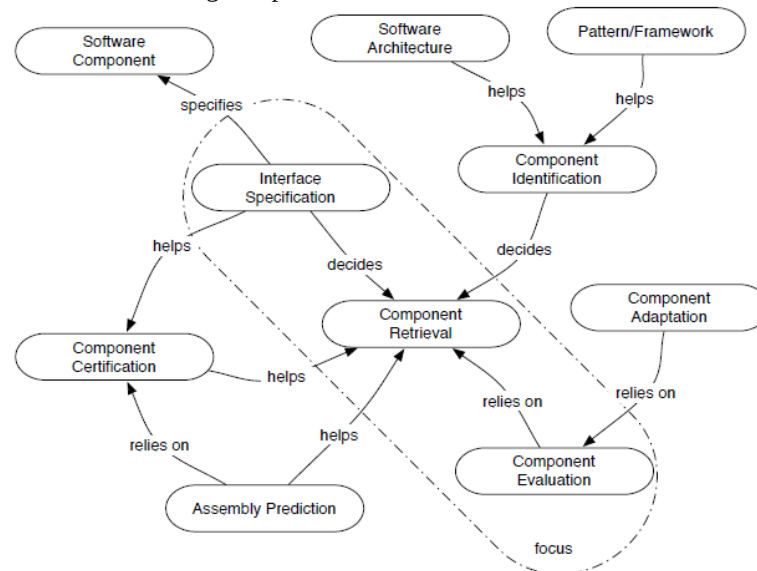


Figure 4: Concept Map of Component Selection
VI. SCIL PROPOSED FRAMEWORK

We have stated that the selection of component selection is still tough. While many dissimilar developers providing a huge variety of different kinds of components, it is difficult to recognize the essential one simply based on the information the developers deliver. This is because the components explanations are generally imprecise, whereas the internal mechanisms of the components are invisible to the users. On the other side, the requirements of user are also stated changeably and ambiguously. The gap between the components provided and the components required almost always exists when the components from external Sources are targeted. This gap is hard to decrease because of the nature of CBD, which has two separate processes of developing components and systems. Though, there is little connection between these two processes. This increases the misunderstanding between component users and component developers.

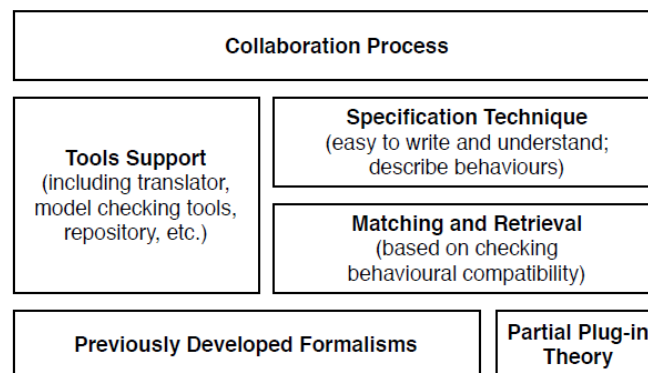


Figure 4: Simple Component Interface Language Proposed Framework Structure

Increasing the communication between component users and developers can decrease the gap and eliminate the misunderstanding; therefore the essential components can be found effectively. This is the objective of building our collaboration framework. In the proposed framework, for collaboration purposes, a common language is used to specify both components and the user requirements for the components. Therefore, the component developers and users can communicate with each other by exchanging component specifications and requirement. Anyone can use formal techniques to state components and user requirements at an abstract, but complete level and sufficiently precise. Still, it is presently impractical to imagine common component users to write formal specifications directly, especially when components are becoming larger and more difficult.

Therefore, it is required to retain the specifications easy to understand and write. It is very clear that the component behaviour is one of the most important aspects while considering to use the component in the target system. Specification language should be competent to explain component behaviours rather than interface syntax. Time being, there is underlying formalism to make the specifications precise, and this formalism is hidden from normal component users. In order for solution, reuse all those earlier developed formalisms from other research projects to check component behavioral compatibility. The major benefits of performing this are their accompanying tools can be reused as well in implementation. It is only required to form a bridge from the specification language to those formalisms.

VII. Conclusion & Future Work

In conclusion, SCIL is not proposed to explain interface signatures but explain a transition system along with requirements specified using a mix of temporal logic and the other transition systems, viz., user specified components and scenarios. A combined SCIL description of components and requirements are translated to some existing modelling languages. Then these models are checked by various tools. In future this simple Component Specification Language (SCIL) will be implemented and its efficiency over traditional methods will be measured.

References

- [1] Mili, A., S. Yacoub, E. Addy and H. Mili: "Toward an Engineering Discipline of Software Reuse", IEEE Software, Vol. 16, No. 5, 1999
- [2] P. Popic, D. Desovski, W. Abdelmoez, and B. Cukic. Error Propagation in the Reliability Analysis of Component Based Systems. In the 16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05), pages 53–62, Chicago, Illinois, USA, 2005. IEEE Computer Society.
- [3] ComponentSource. <http://www.componentsource.com>. 1996-2006.
- [4] C. Abst, B. Boehm, and E.B. Clark. Empirical Observations on COTS Software Integration Effort Based on the Initial COCOTS Calibration Database. In the 22nd International Conference on Software Engineering (ICSE 2000) COTS Workshop, Limerick, Ireland, 2000.
- [5] G. Leavens and Y. Cheon. Design by Contract with JML, 2003.
- [6] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# Programming System: An Overview. In Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS 2004), volume 3362 of LNCS, Marseille, France, 2004. Springer.
- [7] G. Booch, J. Rumbaugh, and I. Jacobson. The Unified Modeling Language User Guide. Addison-Wesley, 1999.
- [8] Object Management Group. UML 2.0 OCL Specification, 2003.
- [9] N. Aguirre and T. Maibaum. A Temporal Logic Approach to Component-Based System Specification and Reasoning. In the 5th ICSE Workshop on Component-Based Software Engineering: Benchmarks for Predictable Assembly, Orlando, Florida, 2002.
- [10] C. Canal, E. Pimentel, and J. M. Troya. On the Composition and Extension of Software Components. In Foundations of Component-Based Systems Workshop, Zurich, Switzerland, 1997.
- [11] L. de Alfaro and T. A. Henzinger. Interface Automata. In the 9th Annual Symposium on Foundations of Software Engineering, pages 109–120. ACM Press, 2001.
- [12] B. Finkbeiner and I. Kruger. Using Message Sequence Charts for Component-Based Formal Verification. In OOPSLA 2001 Workshop on Specification and Verification of ComponentBased Systems, Tampa, FL, USA, 2001.
- [13] Y. Jin, C. Lakos, and R. Esser. Component-based Design and Analysis: A Case Study. In the International Conference on Software Engineering and Formal Methods, Brisbane, Australia, 2003. IEEE Computer Society Press.
- [14] K. K. Lau and M. Omaghi. Logic for Component-based Software Development, pages 347–373. Computational Logic: Logic Programming and Beyond, Lecture Notes in Artificial Intelligence 2407. Springer-Verlag, 2002.
- [15] M. Lumpe, J. G. Schneider, O. Nierstrasz, and F. Achermann. Towards A Formal Composition Language. In Foundations of Component-Based Systems Workshop, Zurich, Switzerland, 1997.
- [16] T. Isakowitz and R. J. Kauffman. Supporting Search for Reusable Software Objects. IEEE Transactions on Software Engineering, 22:407–423, 1996.
- [17] William B Frakes and Thomas P Pole. An Empirical Study of Representation Methods for Reusable Software Components. IEEE Transactions on Software Engineering, 20:617–631, 1994.
- [18] Ruben Prieto-Diaz. Implementing Faceted Classification for Software Reuse. Communications of the ACM, 34(5):89–97, 1991.
- [19] Michael L. Creech, Dennis F. Freeze, and Martin L. Griss. Using Hypertext in Selecting Reusable Software Components. In the 3rd Annual ACM Conference on Hypertext, San Antonio, TX USA, 1991.
- [20] Af. Mili, F. Mili, and Al. Mili. Reusing Software: Issues and Research Directions. IEEE Transactions on Software Engineering, 21(6):528–562, 1995.
- [21] V. Sugumaran and V. C. Storey. A Semantic-Based Approach to Component Retrieval. SIGMIS Database, 34(3):8–24, 2003.
- [22] A. M. Zaremski and J. M. Wing. Signature Matching: a Tool for Using Software Libraries. ACM Transactions on Software Engineering and Methodology, 4:146–170, 1995
- [23] A. M. Zaremski and J. M. Wing. Specification Matching of Software Components. ACM Transactions on Software Engineering and Methodology, 6(4):333–369, 1997.
- [24] J. Kontio, G. Galdiera, and V. R. Basili. Defining Factors, Goals and Criteria for Reusable Component Evaluation. In the 1996 conference of the Centre for Advanced Studies on Collaborative research (CASCON '96), Toronto, Canada, 1996.
- [25] N. A. Maiden and C. Ncube. Acquiring COTS Software Selection Requirements. IEEE Software, 15(2):46–56, 1998.
- [26] M. Ochs, D. Pfahl, and G. Chrobok-Diening. A COTS Acquisition Process: Definition and Application Experience. In the 11th ESCOM Conference, pages 335–343, Shaker, Maastricht, 2000.
- [27] I. Crnkovic and M. Larsson, editors. Building Reliable Component-based Software Systems. Artech House Publishers, 2002.
- [28] M. Matsushita. Ranking Significance of Software Components Based on Use Relations. IEEE Transactions on Software Engineering, 31(3):213–225, 2005. Member-Katsuro Inoue and Member-Reishi Yokomori and Member-Tetsuo Yamamoto and Member-Shinji Kusumoto.
- [29] P. Oberndorf, L. Brownsword, E. Morris, and C. Sledge. Workshop on COTSBased Systems. Technical report, Software Engineering Institute, Carnegie Mellon University, Nov. 1997.
- [30] C. Albert and L. Brownsword. Evolutionary Process for Integrating COTSBased Systems (epic): An Overview. Technical Report CMU/SEI-2002-TR-009, Software Engineering Institute, Carnegie Mellon University, 2002.

- [31] P. Allen. Component-Based Development for Enterprise Systems : Applying the Select Perspective. Cambridge University Press, 1998.
- [32] R. Allen, R. Douence, and D. Garlan. Specifying Dynamism in Software Architectures. In Foundations of Component-Based Systems Workshop, Zurich, Switzerland, 1997.
- [33] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. ACM Transactions on Software Engineering and Methodology (TOSEM), 6(3):213–249, 1997.
- [34] L. de Alfaro and T. A. Henzinger. Interface Theories for Component-Based Design. Lecture Notes in Computer Science, 2211:148–165, 2001.
- [35] Sun Microsystems. JavaBeans API Specification, Version 1.01, 1997.
- [36] Microsoft Corporation. The Component Object Model Specification, 1995.
- [37] Object Management Group. CORBA Components, Version 3.0, 2002.
- [38] Sun Microsystems. Enterprise JavaBeans Specification, Version 2.1, 2002.