

The Future of Component-Based Development is Generation, not Retrieval

Hans de Bruin Hans van Vliet
Vrije Universiteit, Amsterdam
Mathematics and Computer Science Department
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
e-mail: {hansdb,hans}@cs.vu.nl

Abstract

Component-Based Development (CBD) has not redeemed its promises of reuse and flexibility. Reuse is inhibited due to problems such as component retrieval, architectural mismatch, and application specificity. Component-based systems are flexible in the sense that components can be replaced and fine-tuned, but only under the assumption that the software architecture remains stable during the system's lifetime. In this paper, we argue that systems composed of components should be generated from functional and non-functional requirements rather than being composed out of existing or newly developed components. We propose a generation technique that is based on two pillars: Feature-Solution (FS) graphs and top-down component composition. A FS-graph captures architectural knowledge in which requirements are connected to solution fragments. This knowledge is used to compose component-based systems. The starting point is a reference architecture that addresses functionality concerns. This reference architecture is then stepwise refined to cater for non-functional requirements using the knowledge captured in a FS-graph. These refinements are the architecture-level counterpart of aspect weaving as found in Aspect-Oriented Programming (AOP).

1 Introduction

Component-Based Development (CBD) is concerned with the development of systems from reusable parts, that is, components. Unfortunately, CBD has not lived up to its expectations yet (see for instance [1, 3, 12]). To be successful with CBD as a reuse technology, components have to be retrieved from a kind of repository based on a set of criteria to judge

whether components are fit for the job or not. The criteria not only include functional requirements, but non-functional requirements, such as performance and footprint, as well. The current state of affairs is that not many components can actually be reused. The problem is partially caused by incomplete component specifications that do not tell the whole story. Yet another cause is architectural mismatch [10]; a component may function perfectly well in one setting, but may fail in a different setting due to making (possibly undocumented) assumptions on the environment. Despite these problems, we believe that there are more fundamental problems to be solved before components can be reused. True, we have no difficulty with reusing domain independent components such as user interface and database connectivity components. However, it is less obvious to reuse application specific components, not only across application domains, but in the same application domain as well. Typically, components are too rigid as far as its functional and non-functional properties are concerned to be adapted to (slightly) different settings.

For the aforementioned reasons, some researchers and practitioners do not longer view CBD as a means for reuse (see for instance [5]). They view CBD as a development method for constructing flexible software in which the main driver is change. A system should be designed in such a way that components exhibit cleanly defined interfaces and that they do not depend strongly on the support offered by surrounding components. If designed as such, components are eligible for change by means of component replacement or customization. Unfortunately, CBD is not the final answer for managing change. The underlying assumption is that a software architecture, composed of components, can be developed that remains relatively stable during the lifetime of a system. This assumption is not

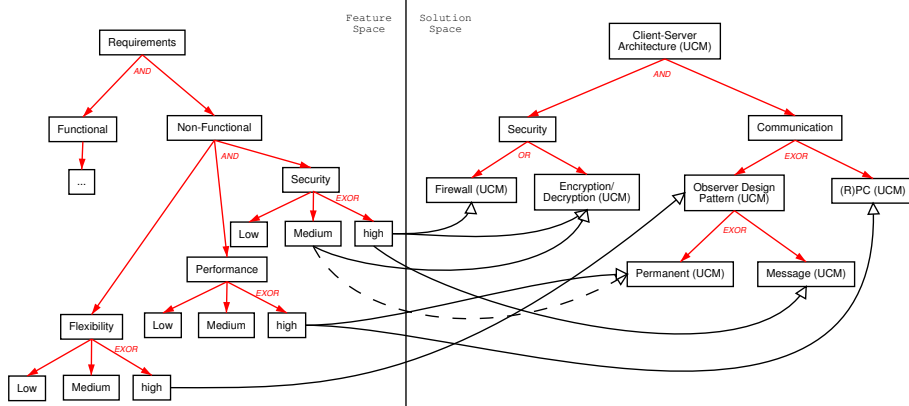


Figure 1. Feature-Solution graph for the Client-Server system.

valid in general. Systems evolve over time, and many times in rather unpredictable ways. At some point in time, we might find that a software architecture can no longer accommodate evolutionary changes, and hence the CBD approach falls in pieces. Also, some changes are not localized, but affect multiple components. Examples of such crosscutting effects are typically caused by imposing new or changed non-functional requirements on a system. Consider, for example, shifting the emphasis from performance to footprint, or vice versa. This will likely have an impact on a large number of components, and might even require a different software architecture.

We have reached the conclusion that CBD does not deliver its promises of reuse and flexibility. This leaves us with the question which alternative methods and techniques can be used instead. Our answer is that component-based systems should be generated. We are not alone in this view, generative programming techniques are gaining more and more interest these days [6]. We propose a generation technique that generates systems from functional as well as non-functional requirements. The generation technique is based on two pillars:

Feature-Solution (FS) graphs. A FS-graph captures architectural knowledge in the form of desired features (e.g., functional and non-functional requirements) and solutions that realize these features (e.g., architectural and design patterns).

Top-down component composition. We envisage a quality-driven approach to generating component-based systems in which the FS-graph

plays a crucial role. This process is akin to the process described in [2]. The first step in this process is the derivation of a reference architecture that meets the functional requirements set. Next, the attention focuses on non-functional requirements by iteratively applying known design solutions as codified in the FS-graph. Typically, this requires several iterations. These iterations might also involve backtracking steps because we usually have to deal with conflicting requirements.

The impact of quality aspects, such as performance and security, is typically not restricted to a single component only, but may affect a large number of components. In a FS-graph, we can capture all the knowledge that is required to refine multiple components simultaneously in a consistent manner. This type of refinement may be called Aspect-Oriented Programming (AOP) [11] at the architectural level.

2 Component Generation Techniques

In this section, we discuss the generation techniques in more detail. At the heart of the iterative, quality driven approach for generating component-based system is the FS-graph. Consider as an example a Client-Server (CS) system in which a client component requests a server component to perform one of its duties. A FS-graph for the CS system is shown in Figure 1. Two spaces are recognized in the FS-graph. The Feature (F) space contains the requirements, whereas the Solution (S) space contains solutions addressing these requirements. Features as well as solutions are decomposed in AND-(EX)OR decomposition trees. An AND

decomposition of a node in either the feature or the solution space means that all its constituents must be available, an OR requires an arbitrary (≥ 0) number of constituents, and an EXOR requires precisely one constituent. The key idea is that a feature in the F-space may select a solution in the S-space as defined by directed selection links between nodes (indicated by a solid line). It is also possible to explicitly rule out a particular solution. This is done by connecting a feature to a solution with a *negative* selection link (indicated by a dashed line).

In the example, we focus on non-functional requirements, in particular flexibility, security and performance requirements. If a high flexibility level is desired, the FS graph dictates that we should use the Observer design pattern, because of its properties of reducing the coupling between peers and supporting multiple observers. On the other hand, if we want high performance, the FS graph selects a direct invocation style in the form of (remote) procedure calls. It is interesting to observe that a high level of flexibility and a high level of performance cannot be obtained simultaneously since these requirements select solutions that rule out each other, as implied by the EXOR decomposition of the communication node. Thus, a FS graph contains trade-off information as well. Typically, several design process cycles are required to arrive at a design that satisfies all non-functional requirements. The process is shown in Figure 2 and is described in detail in [8].

The generator generates the components on the basis of requirements and architectural solutions captured in the FS-graph. The generator uses the FS-graph to refine the given CS architecture. Next, the generated system is evaluated against all functional and non-functional requirements set. Now suppose that a certain requirement has not been met in the current system. By consulting the FS-graph, we might come up with several solutions that can be applied to remedy the shortcoming. Thus, in principle, the outcome of the evaluation phase can be used to drive the architecture generation process in the next iteration. That is, the generator selects a solution and then generates a refined system, which is evaluated in its turn. This process is repeated until all requirements are met or we run out of potential solutions.

We use Use Case Maps (UCM) [4] for representing component-based systems. UCM is a diagrammatic modeling technique to describe behavioral and, to a lesser extent structural, aspects of a system at a high level of abstraction. UCM provides stubs (i.e., the hooks or variability points) where the behavior of a system can be varied statically at construction time as well as dynamically at run time. The advantage of UCM is that it focuses on the larger, architectural issues, and its support of plug-ins and stubs, both static and dynamic. Further details can be found in [9].

3 Component Generation in Practice

We are currently putting our approach in practice in the QUASAR (QUALity-driven Software Architecture) project. The goal is to generate real systems in real application domains using the techniques described in this paper. Although it is too early to draw definite conclusions, the approach looks promising.

As a starting point, we use a reference architecture that is composed of components that implement the required functionality. The components are identified through a domain engineering process capturing the commonalities and variabilities of a domain. What sets our approach apart from a standard CBD approach is that we do not treat a component as a black-box that can be adapted to a certain extent. Instead, we use a grey-box approach [7] in which the behavior of components is described in terms of UCMs with stubs. The stubs provide the means to refine components recursively. The FS-graph plays a central role in this process. It contains all the knowledge that is required to tailor components in order to meet non-functional requirements such as performance and security requirements. A refinement operation is not necessarily restricted to a single component only but may crosscut

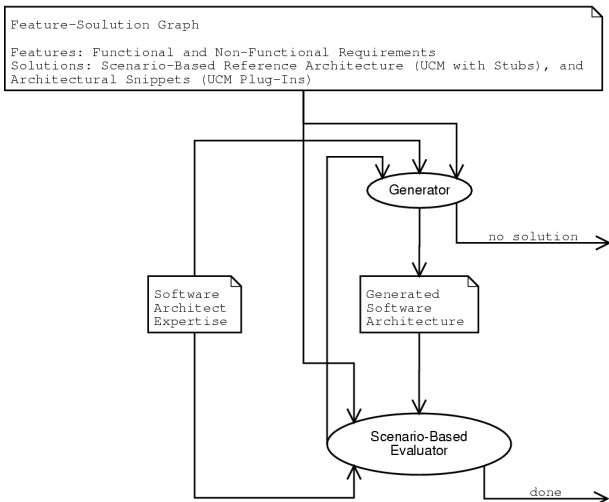


Figure 2. The process of generating and evaluating architectures.

several components. The knowledge of how and where the system has to be adapted can be captured in a FS-graph as is illustrated in Figure 1. Effectively, the FS-graph ensures that all refinements needed to satisfy particular requirements are effectuated.

Our current experiences show that the goal of generating component-based systems from functional and non-functional requirements is not (yet) feasible. The limitations of our approach are the following. A fairly developed reference architecture is needed before the iterative generation process can start. In order to arrive at a reference architecture, obviously some design decisions need to be made. In principle, the generation process should be a closed-loop process without requiring human (software architect) intervention. This is hard to achieve since the evaluation of certain quality attributes requires an expert eye.

4 Concluding Remarks

CBD as a black-box assembly technique is in our opinion a dead-end street, because of its inherent problems of lack of reuse and flexibility. Instead of composing a system out of existing or newly developed components, we propose to compose a component-based system from functional and non-functional requirements and design solution fragments captured in a FS-graph. In this approach, reuse assets stem from two sources. Firstly, from a domain engineering process we can identify a set of reusable (grey-box) components. Secondly, reusable solution fragments (e.g., architectural and design patterns) can be connected to domain and application specific functional and non-functional requirements in a FS-graph. These solution fragments can then be used for component adaptation.

In conclusion, the keywords for characterizing the CBD future are domain engineering and component generation/adaptation. The FS-graph and the top-down composition techniques described in this paper show how this future can be realized, although there is plenty of room for improvement.

References

- [1] Paul G. Basset. *Framing Software Reuse: Lessons from the Real World*. Prentice Hall, Upper Saddle River, New Jersey, 1996. Yourdon Press.
- [2] Jan Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison-Wesley, 2000.
- [3] Alan W. Brown and Kurt C. Wallnau. The current state of CBSE. *IEEE Software*, 15(5):37–46, September 1998.
- [4] R.J.A. Buhr. Use Case Maps as architecture entities for complex systems. *IEEE Transactions on Software Engineering*, 24(12):1131–1155, December 1998.
- [5] John Cheesman and John Daniels. *UML Components: A Simple Process for Specifying Component-Based Software*. Object Technology Series. Addison-Wesley, Reading, Massachusetts, 2000.
- [6] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Reading, Massachusetts, 2000.
- [7] Hans de Bruin. A grey-box approach to component composition. In Krzysztof Czarnecki and Ulrich W. Eisenecker, editors, *Proceedings of the First Symposium on Generative and Component-Based Software Engineering (GCSE'99), Erfurt, Germany*, volume 1799 of *Lecture Notes in Computer Science (LNCS)*, pages 195–209, Berlin, Germany, September 28–30, 1999. Springer-Verlag.
- [8] Hans de Bruin and Hans van Vliet. Scenario-based generation and evaluation of software architectures. In Jan Bosch, editor, *Proceedings of the Third Symposium on Generative and Component-Based Software Engineering (GCSE'2001), Erfurt, Germany*, volume 2186 of *Lecture Notes in Computer Science (LNCS)*, pages 128–139, Berlin, Germany, September 10–13, 2001. Springer-Verlag.
- [9] Hans de Bruin and Hans van Vliet. Top-down composition of software architectures. In Per Runeson, editor, *Proceedings of 9th International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'2002), Lund, Sweden*, pages 1–10, April 8–11, 2002.
- [10] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17–26, November 1995. Carnegie Mellon University.
- [11] Gregor Kiczales, John Lamping, Anurg Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In M. Askit and M. Matsuoka, editors, *Proceedings of 11th European Conference on Object-Oriented Programming (ECOOP'97), Finland*, volume 1241 of *Lecture Notes in Computer Science (LNCS)*, pages 220–242, Berlin, Germany, June 9–13, 1997. Springer-Verlag.
- [12] Hafedh Mili, Ali Mili, Sherif Yacoub, and Edward Addy. *Reuse-Based Software Engineering: Techniques, Organization, and Controls*. John Wiley and Sons, New York, 2002.