

Feature and Feature Interaction Modeling with Feature-Solution Graphs

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

July 27, 2001

Abstract

This position paper discusses an approach for feature and feature interaction modeling. The key idea is to connect features (i.e., user requirements) with solution fragments in a so called feature-solution graph. This graph serves two purposes. Firstly, it can be used to pinpoint feature interactions. Secondly, it can guide an iterative architecture development and evaluation process.

1 Introduction

The architecture of a software system captures early design decisions. These early design decisions reflect major quality concerns, including functionality. We would obviously like to design our systems such that they fulfill the quality requirements set for them. Unfortunately, we in general do not succeed in doing so in a straightforward way. This is especially true for product lines that must evolve and/or must support variations with slightly different features. What we need is an approach that on the one hand can assess the impact of feature interactions and on the other hand can be used to generate different versions of a system dependent on the required features in such a way that all quality requirements are satisfied. This position paper¹ is concerned with techniques to support this approach. In particular, we propose to use a rich feature-solution graph to capture the evolving knowledge about quality requirements and solution fragments. This graph is next used to pinpoint feature interactions and to guide an iterative architecture development and evaluation process. The structure of this feature-solution graph resembles that of the goal-hierarchy in goal-oriented requirements engineering [2, 3]. The solution fragments included in this graph have much in common with Attribute-Based Architectural Styles (ABASs) [1]. In principle, any kind of solution description will do. The approach to generating and evaluating architectures from a feature-solution graph is depicted in Figure 1.

2 Feature-Solution Graph Example

A generic Resource Management (RM) system is used as an example to show how a typical feature-solution graph looks like. The basic idea is that a *customer* can *reserve a resource* that can be taken up later on. *Resources* are being described in *resource types*. The RM system can be seen as an abstraction for a collection of information systems that all share the concept of claiming resources. For instance, in a hotel reservation system, the resources are rooms, whereas in a car rental system, the resources are cars.

In this example, we use a 3-tier architecture as a starting point for devising architectures for classes of information systems, such as the RM system. As the name suggests, the 3-tier reference architecture is

¹A detailed discussion can be found in our paper presented at GCSE'2001: Scenario-Based Generation and Evaluation of Software Architectures.

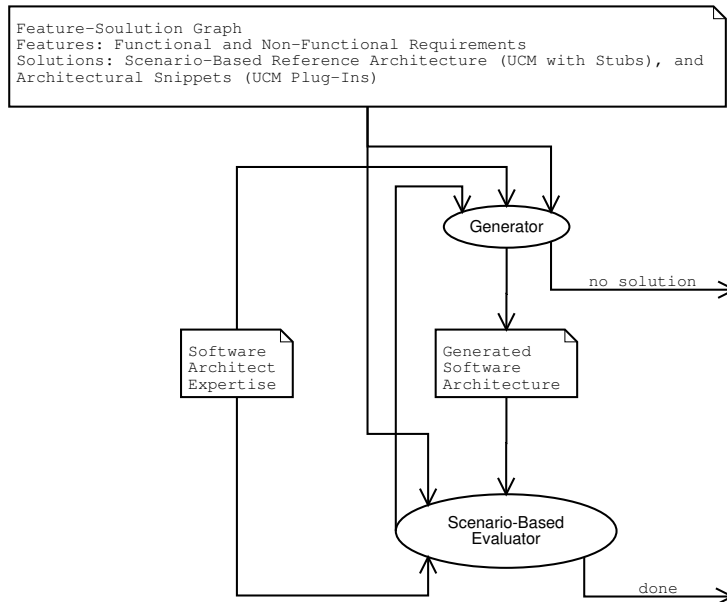


Figure 1: The process of generating and evaluating architectures.

composed of three layers: User Interface (UI), Business Logic (BL), and Data Management (DM) layer. The characteristics of the 3-tier architecture are flexibility, scalability, user independence, availability, and low upgrade costs. On the downside, the architecture can suffer from performance problems and may involve high initial costs [4].

The starting point for generating and evaluating architectures is first to derive feature dependencies, which in their turn spark off solutions in the form of architectural fragments. The feature-solution dependencies are captured in the feature-solution graph. We make a clear distinction between features (i.e., user requirements) and the design particles that provide solutions by defining the following two spaces:

Feature space describes the desired properties of the system as expressed by the user.

Solution space contains the internal system decomposition in the form of a reference architecture composed of components. In addition, the solution space may also contain general applicable solutions that can be selected to meet certain non-functional requirements.

A feature-solution graph for the RM system is given in Figure 2. It is composed of the usual AND-OR decompositions to denote combinations and alternatives of features and solutions. We use an AND decomposition to denote that all constituents are included, an OR to select an arbitrary number of constituents, and an EXOR to select exactly one constituent. Besides the AND-OR relationships, the graph contains directed *selection* edges (represented by a solid curve that ends with a hollow pointer) to establish the connection between features and solutions. Thus, a feature in the feature space selects a solution in the solution space. A solution may be connected by selection edges to more detailed, general applicable solutions (e.g., design patterns). That is, solutions are found by determining the transitive closure of selection edges originating from the feature space.

In some cases, it is useful to outrule a solution explicitly. This is done with *negative selection* edges (represented by a dashed curve that ends with a hollow pointer). For example, if we want high flexibility, then the BL layer should *not* be integrated in the DM layer, since merging both layers makes it more difficult to adapt the business logic.

It is interesting to observe that the feature-solution graph contains tradeoff knowledge. For example, the features “high flexibility” and “medium and high performance” give rise to a clash in the sense that for

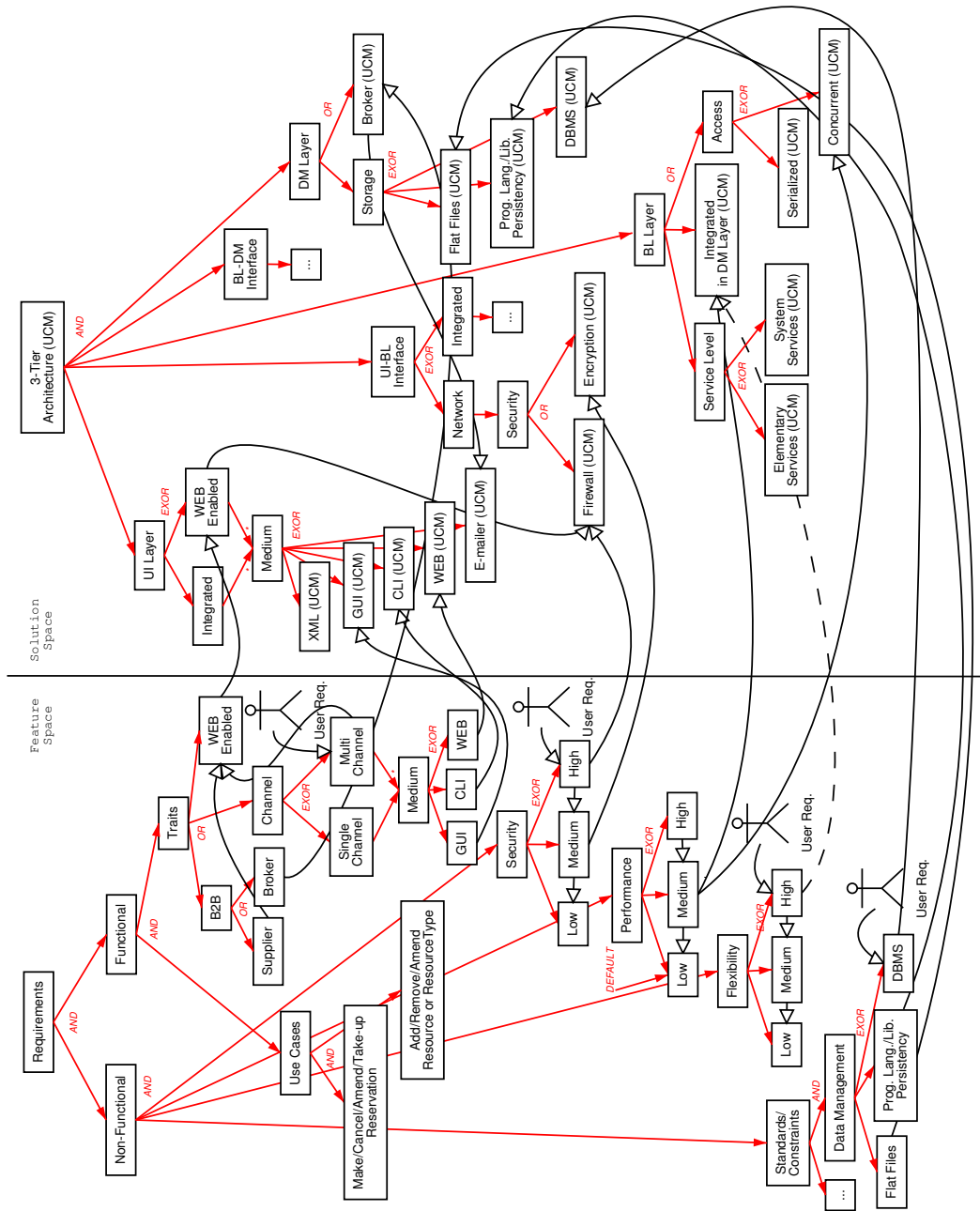


Figure 2: Feature-Solution graph.

flexibility reasons the BL and DM layer should be separated, whereas for performance reasons they should be integrated. Other indications of tradeoff points can be found when two or more solutions in an OR decomposition are selected simultaneously, that is, their selection stems from distinct feature choices.

For understanding the process of generating and evaluating architectures, recall the scheme given in Fig-

ure 1. The architecture generator is driven by the feature-solution graph. Some features can be selected directly on the basis of the requirements. For instance, we might require a high level of flexibility. As a result, the non-functional requirement “Flexibility” in the feature space is set to “High”. The implication of this decision is that the BL and DM layer may *not* be merged. For some requirements, the required level of support is harder to determine. Consider, for example, performance. A performance requirement might be that a “Make Reservation” takes less than a second. However, it is not clear from the outset which performance level in the feature-solution graph will satisfy the performance requirement. So as an initial guess, we set the performance to “Low”, since this results in less constraints on the design than higher performance levels. The software architecture then has to be evaluated in order to assess whether the performance requirement is satisfied or not. If not, the outcome of the evaluation process will result in setting the performance to a next level (i.e., “Medium”). The next step is to generate a new architecture, and so on, until all requirements are satisfied, or we reach the conclusion that the requirements are too strict.

To summarize the closed-loop process: The non-functional features can be set to a required level, in our example ranging over “Low”, “Medium”, and “High”. A level for a particular feature selects solutions, providing the basis for generating a candidate software architecture. The required level of some features (e.g., flexibility in our example) can be determined directly from the requirements, whereas others must be determined from evaluations (e.g., performance). To put it differently, the feature levels are the knobs with which the architecture can be fine-tuned. Notice that the feature-solution graph is typically underspecified in the sense that not all (EX)OR decompositions are necessarily connected by selection edges. In this way, we have created degrees of freedom for generating design alternatives.

3 Concluding Remarks and Future Work

We have discussed an approach to feature and feature interaction modeling. The central model is the feature-solution graph, which combines AND-OR feature and solution decompositions with (negative) selection relationships. The feature-solution graph connects quality requirements with design alternatives. In addition, it can be used to pinpoint tradeoffs between quality attributes such as flexibility and performance, as shown in this paper.

We envisage that the feature-solution graph can be further enriched with relationships and annotations to accurately capture domain and architectural knowledge and the connection between them. In this way, we build a body of knowledge that can be applied to similar problems.

References

- [1] M.H. Klein, R. Kazman, L. Bass, J. Carriere, M. Barbacci, and H. Lipson. Attribute-based architectural styles. In P. Donohue, editor, *Software Architecture*, pages 225–244. Kluwer Academic Publishers, 1999.
- [2] Axel van Lamsweerde. Requirements engineering in the year 00: A research perspective. In *Conference Proceedings ICSE'00*, pages 5–19, Limerick, Ireland, 2000. ACM.
- [3] John Mylopoulos, Lawrence Chung, Stephen Liao, Huaqing Wang, and Eric Yu. Exploring alternatives during requirements analysis. *IEEE Software*, 18(1):92–96, January 2001.
- [4] A. Umar. *Object-Oriented Client/Server Internet Environments*. Prentice Hall, Englewood Cliffs, New Jersey, 1997.