Software Architecture Evolution

Olivier Barais\(^1\), Anne Françoise Le Meur\(^2\), Laurence Duchien\(^2\), and Julia Lawall\(^3\)

\(^1\) Université de Rennes 1/IRISA/INRIA, Triskell project - France
\(^2\) Université de Lille 1, LIFL/INRIA ADAM project - France
\(^3\) DIKU, University of Copenhagen, Denmark

Summary. Software architectures must frequently evolve to cope with changing requirements, and this evolution often implies integrating new concerns. Unfortunately, when the new concerns are crosscutting, existing architecture description languages provide little or no support for this kind of evolution. The software architect must modify multiple elements of the architecture manually, which risks introducing inconsistencies.

This chapter provides an overview, comparison and detailed treatment of the various state-of-the-art approaches to describing and evolving software architectures. Furthermore, we discuss one particular framework named TranSAT, which addresses the above problems of software architecture evolution. TranSAT provides a new element in the software architecture descriptions language, called an architectural aspect, for describing new concerns and their integration into an existing architecture. Following the early aspect paradigm, TranSAT allows the software architect to design a software architecture stepwise in terms of aspects at the design stage. It realises the evolution as the weaving of new architectural aspects into an existing software architecture.

10.1 Introduction

The role of software architecture in the engineering of software-intensive systems is becoming increasingly important and widespread. A software architecture models the structure and behavior of a system, including the software elements and the relationships between them. It is the basis of the design process, a guide for the software development process and one of the main inputs to drive the development of integration tests. There are currently a number of Architecture Description Languages (ADLs) [358], which enable an architect to specify a software architecture. During the design process, the architect uses an ADL to create the software architecture of a system by constructing and combining increasingly complex components and connectors.

An ADL makes it easy to construct an initial description of the architecture of a system. In practice, however, so that an architecture can remain useful over time, it must be able to evolve in response to the changing and often conflicting requirements of the many diverse stakeholders. An architecture can thus not be viewed as simply
a description of a static software structure, but as a description of the space in which this software structure can dynamically evolve. Most ADLs, however, do not provide support for describing the evolution of a software system.

Software systems undergo two main kinds of evolution: \textit{internal evolution} and \textit{external evolution}. Internal evolution models the changes in the topology of the components and interactions as they are created or destroyed during execution. As such, it captures the dynamics of the system. External evolution models the changes in the specification of the components and interactions that are required to cope with new stakeholder requirements. It entails adaptation of the software architecture. In the first part of this chapter, we study a number of approaches that address these issues of evolution in a software architecture. We furthermore classify the approaches according to the kind of evolution that is supported.

In the second part of this chapter, we focus on the issue of separation of concerns in the context of the external evolution of a software architecture. Software architectures are designed around the concepts of components and their interactions, and thus suffer from the "tyranny of the dominant decomposition" [490], in which some concerns cannot be adequately modularised because they crosscut the chosen dimension of decomposition. Evolutions in such concerns require pervasively modifying the ADL specification, at all points affected by the concerns, which can be tedious and difficult. In the context of implementations, Aspect-Oriented Software Development (AOSD) [276], has been proposed to improve the separation of concerns [153]. At the architecture level, several approaches have proposed to follow the spirit of AOSD, by putting the description of each concern in a separate architecture construct, that can automatically be integrated into an existing software architecture by a \textit{weaver}. However, because architectures are complex and aspects are invasive, many transformations may be needed to integrate or modify a concern, making the specification of the transformation highly error prone. We present in detail the system TranSAT [36, 38], which detects inconsistencies that may be introduced by such an architectural aspect as early as possible.

The rest of this chapter is organised as follows. Section 10.2 presents several software architecture languages in order to identify the key concepts of these languages and their advantages and shortcomings. Sections 10.3 and 10.4 present several initial solutions to cope with internal and external evolution. Section 10.5 presents the TranSAT approach, showing how an explicit specification of weaving can help guarantee the consistency of the resulting architecture. Section 10.6 describes some related work and finally Section 10.7 concludes and presents some remaining critical issues.

### 10.2 Component-Based Software Architecture: Concepts and Open Issues

The software architecture of a software system describes its high level structure and behavior. In a software architecture specification, a system is represented as a set of software components, their connections, and their behavioral interactions. Creating