Science of Computer Programming 98 (2015) 764–784

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

A perspective on architectural re-engineering

Alejandro Sanchez [a,b], Nuno Oliveira [b], Luis S. Barbosa [b,*], Pedro Henriques [c]

[a] Universidad Nacional de San Luis, San Luis, Argentina
[b] HASLab – INESC TEC & Universidade do Minho, Braga, Portugal
[c] CCTC, Universidade do Minho, Braga, Portugal

ARTICLE INFO

Article history:
Received 9 March 2012
Received in revised form 16 February 2014
Accepted 17 February 2014
Available online 18 March 2014

Keywords:
Software architecture
Coordination patterns
Re-engineering

ABSTRACT

Continuous evolution towards very large, heterogeneous, highly dynamic computing systems entails the need for sound and flexible approaches to deal with system modification and re-engineering. The approach proposed in this paper combines an analysis stage, to identify concrete patterns of interaction in legacy code, with an iterative re-engineering process at a higher level of abstraction. Both stages are supported by the tools CoordPat and Archery, respectively. Bi-directional model transformations connecting code level and design level architectural models are defined. The approach is demonstrated in a (fragment of a) case study.

1. Introduction

Legacy software has to be maintained, improved, replaced, adapted and regularly assessed for quality, which brings their re-engineering to the top of concerns of the working software architect. This is not, however, an easy task. On the one hand a systems' architecture relies more and more on non-trivial coordination logic for combining autonomous services and components, often running on different platforms. On the other hand, often such a coordination layer is strongly weaved within the application at the source code level.

The CoordInspector tool [1,2] was a first attempt to address this problem by systematically inspecting code in order to isolate the coordination threads from the computational layer. This is done in a semi-automatic way through the combination of generalised slicing techniques and graph manipulation.

Such a stage of *architectural discovery* constitutes a necessary, but not sufficient step in a re-engineering process. Actually, experience shows that

- recovering an architectural model from code would be much more effective if driven by some notion of *pattern* encoding typical interactions;
- in any case, the low level model produced through slicing and code analysis, needs to be mapped to a more structural one, to precisely abstract and identify components and connectors and enable their re-engineering.

This paper, combining previous research on both *program understanding* and *software architecture*, addresses the challenge as follows:

- First of all it introduces a notion of a *coordination pattern* directly extracted from the program dependency graph of the legacy system, as well as a language, CoordL, to describe such patterns. A collection of coordination patterns constitutes

* Corresponding author.

http://dx.doi.org/10.1016/j.scico.2014.02.026
0167-6423/

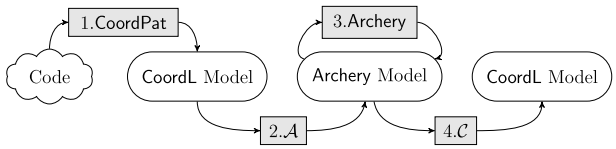

Fig. 1. An approach to architectural re-engineering.

a low level architectural description in terms of execution threads and interaction points. Its main purpose is to act as a template to inspect code and represent its coordination layer. CoordPat, a pattern search facility based on this idea, was combined with CoordInspector to enhance the tool support for the technique.

- Then a systematic method is proposed to translate such patterns into a high-level architectural model in Archery [3] which provides a proper setting for studying and simulating architectural changes. This iterative process is illustrated in Fig. 1 through the loop arrow from the Archery model.
- Finally, a reverse translation method is proposed to transform the new architectural model back to a collection of coordination patterns which guides the re-implementation process.

Fig. 1 sums up the proposed approach for architectural re-engineering. The combination of CoordPat and Archery equips the architect with suitable tool-support for recovering architectural decisions, reconstructing an architectural model, and analysing the impact of different possible modifications. Since the two frameworks work at different abstraction levels, (the first providing abstractions over dependency graphs; the second entailing a *components-and-connectors* view of architectural organisation), 'translations' $\mathcal{A}$ and $\mathcal{C}$ in Fig. 1 are central to the proposed method. Their application is illustrated in detail through an example, extracted from a real case study.

A main motivation for this work is the problem of quality assessment and re-engineering of Open Source Software (OSS) as discussed in [4]. Availability of code makes OSS particularly suited to application of backward analysis and program understanding techniques [5]. Often architectural decisions are only partially documented in OSS due to the *pay-as-you-go* documentation style and the distributed and heterogeneous nature of its development. Architectural re-engineering plays nevertheless a main role in OSS maintenance and evolution: it is particularly critical to endow OSS communities with techniques and tools to identify and to control architectural drift, i.e., the accumulation of architectural inconsistencies resulting from successive code modifications, that may affect different quality attributes of the system.

The paper is organised as follows: Section 2 describes the approach and the example we use to illustrate it; Sections 3 and 4 introduce, respectively, CoordPat and Archery, the two main methods/tools in this process; Section 5 describes the systematic translations of CoordL to Archery models and back; Section 6 illustrates the approach through a detailed example; finally, Section 7 reports on related work and concludes.

2. An approach to architectural re-engineering

The approach proposed in this paper for architectural re-engineering of legacy code is depicted in Fig. 1. As explained above, it resorts to the combination of a tool for extracting coordination patterns from source code (CoordPat) and a high level architectural description language (Archery) plus a guide to map patterns back and forth between these two levels.

The example chosen to illustrate the approach is part of a real case study on software integration described in [2]. It concerns a service to control the updating of user profiles and information common to a number of components of a company's information system. In its original formulation the context is that of a company offering professional training through e-learning courses. The information system comprises the following three main components: an Enterprise Resource Planner (ERP) for controlling expenses and profits; a Customer Relationship Management (CRM) for managing both general and customer-focused course campaigns; and a Training Server (TS) for managing the courses. These components worked almost independently, all information being shared by a set of scripts executed manually, which gave rise to frequent synchronisation problems. The decision to perform a global architectural analysis and reconstruction was pushed by a sudden growth in the company market share and the need for introducing a web portal for on-line sales.

CoordPat is first applied in the re-engineering process. The tool aims at uncovering, registering and classifying architectural decisions often left undocumented and hardwired in the source code. It implements a rigorous method [6] to extract the architectural layer which captures the system behaviour with respect to its network of interactions. This is often referred to as the *coordination layer*, a term borrowed from research on *coordination* models and languages [7] which emerged in the nineties to exploit the full potential of parallel systems, concurrency and cooperation of heterogeneous, loosely-coupled components.

The extraction stage combines suitable slicing techniques to build a family of *dependency graphs* by pruning a *system dependency graph* [8] first derived from source code. After the extraction stage, the tool exploits such graphs to identify and combine instances of *coordination patterns* and then reconstruct the original specification of the system coordination layer. CoordPat maintains an incrementally-built repository of patterns to guide the analysis process.