Towards a Catalogue of Software Evolution Services

Jan Jelschen, Andreas Winter
Carl von Ossietzky Universität, Oldenburg, Germany
{jelschen,winter}@se.uni-oldenburg.de

Abstract

Evolving large software systems comprises the application of many different techniques to analyse, reverse engineer, visualize, and transform software systems. Tools supporting these activities mostly lack interoperability support, and thus need to be wired manually to facilitate desired tasks. This paper proposes describing existing techniques as services, as a prerequisite to create an interoperability framework for software evolution tools.

1 Introduction

Software evolution encompasses all activities changing a software system after its deployment [2]. Software migration aims at transferring software systems from one platform or environment to another, without changing functionality [5]. Enhancing software quality is the goal of software reengineering or renovation. These fields overlap in techniques and tools used. Particular evolution projects, their objectives, and single evolution tasks, determine the techniques to be used and their composition in appropriate tool chains. Concrete processes, workflows, tasks and enabling tool support need to be defined individually for each evolution effort.

However, automating these processes often requires a great amount of manual effort, wiring various tools only providing single evolution techniques. *Data integration*, as defined by Wasserman [6], is available by many techniques trough a common exchange file format (e.g. GXL [1]), which is still limited, compared to sharing data via common (distributed) repositories. Sophisticated interoperability requires direct tool interaction (*control integration*) and flexible wiring of tools for given workflows (*process integration*).

This paper views software evolution techniques as *services* as a prerequisite to tool interoperability. The level of abstraction of services enables loose coupling, and thereby direct interoperability, flexible enough to wire (*orchestrate*) services into workflows as needed. Rather than developing an interoperability framework from scratch, a service catalogue can be implemented utilizing existing technologies from component-based or service-oriented software engineering.

Section 2 describes a migration scenario as an example of a tool-dependent, integrated process, as it might occur in software evolution projects. The example is used to identify services, and to deduce main challenges for future research in Section 3. Section 4 concludes with an overview.

2 Working Scenario

Consider a project aiming at migrating a legacy COBOL system to Java. It entails evaluating different code translation tools by measuring the quality of the resulting Java code and comparing it to the quality of the COBOL code. In subsequent steps, Java code might be analysed for code clones, and be restructured to remove them.

Each of these activities is supported by one or more services. Figure 1 depicts these services, and how they depend on each other. First, the legacy COBOL system needs to be *parsed*. Results of the parsing service are used to *evaluate metrics*, and as input for a *translation to Java*. The latter service is actually depicted twice in Figure 1 (highlighted in grey), as the scenario involves comparing two different implementations. On the service-level, this is transparent – it allows to exchange one implementation for the other without affecting depending services. The service to *evaluate Java metrics* is used twice, as well. It can rely on both the results of the translation service and those produced by applying *clone detection* and subsequent *clone removal* services. The same holds for *generating Java source code*.

This service view only incorporates *application* services, as opposed to *technical* services, which provide infrastructure. Both should be kept strictly separated for reusability (cf. [4]).

A technical view is shown in Figure 2, evoking ECMA's *Reference Model for Software Engineering Framworks* [3]. It makes some implementation as-

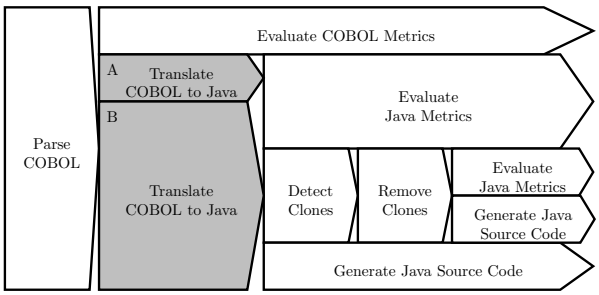

Figure 1: Services involved in the working scenario.

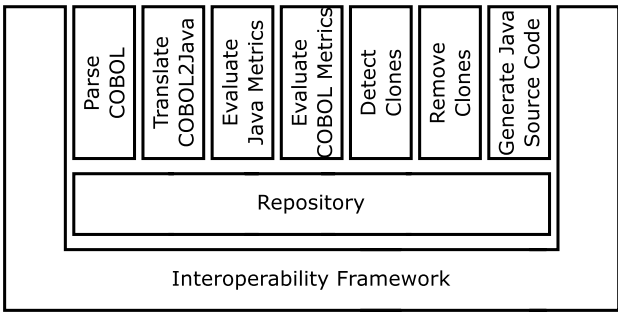

Figure 2: A technical view on the service architecture.

sumptions and reveals additional (technical) services: a *repository* providing central data storage and retrieval has been chosen for data integration. Interaction and communication between services is brokered by an *interoperability framework*. The actual service wiring is done by the framework at runtime.

The following section discusses observations from the scenario regarding key challenges and obstacles.

3 Challenges

Realizing a service catalogue and a corresponding interoperability framework requires the following steps: First, a rigorous *service description* template is required. The service descriptions are to be used as contracts, against which tools can be build. They also provide significant information needed for service composition. *Services have to be identified* to fill the catalogue, which will be used to guide the design of the interoperability framework, by determining what kind of services are available, how they relate, and what their requirements are. High-level interoperability can be implemented using existing frameworks. Since most software evolution services are data-centric, a *repository* is to be used for data integration. Its design has to take into account the size of real-world legacy systems, performance issues, and different data formats specific to individual services.

Service Description Drawing from component-based and service-oriented software engineering approaches, the following common concepts making up a service are identified: *Operations*, which can be invoked by other services, the operations' *data formats* for inputs and outputs, *references* to other services, and *protocols* governing invocation constraints. In addition, a (formal) description of the services' *behaviour* is required. Data schemata can be specified using *meta-models* (see "Repository Design").

Service Identification Techniques for inclusion in the service catalogue are identified by reviewing publications in the field of software evolution, and analysing real-world migration and renovation projects. Existing successful usage examples guide which services to include, to prevent the catalogue from being cluttered with services of marginal significance. Complex services are decomposed into subservices, to identify common parts in different techniques. A reasonable level of minimal granularity has to be defined, to avoid degradation into trivial "microservices".

Repository Design As software evolution services are used to analyse or modify legacy systems, they have to access it. Furthermore, services will often build on each others results. Due to the large amounts of data involved, passing it entirely from service to service is inadequate. Therefore, a common repository is required to handle and optimize data exchange, and store data in an appropriate base format. At the same time, many services require data in task-specific formats to deliver results with optimal performance. Such task-specific views on the data need to be provided by the repository or separate services, as well as the means to reflect possible changes back into the base repository. This will be approached by metamodelling (cf. [7]), requiring rigorous descriptions of the services' input and output formats, and corresponding transformations. Model-driven software engineering offers appropriate technologies to this end.

4 Outlook

The scenario presented in this paper is used as base for a first prototypical interoperability framework, to substantiate requirements and refine the approach. Once more groundwork has been established, and the service description template has been refined, more software evolution scenarios will be analysed, and used to fill the catalogue with services. The finished catalogue will serve both as a taxonomy of the field, and as specification for an interoperability framework for software evolution tools.

References

[1] R. C. Holt, A. Schürr, S. Elliot Sim, A. Winter. GXL: A graph-based standard exchange format for reengineering. *SoCP*, 60(2):149–170, 2006.

[2] IEEE ISO. *International Standard - ISO/IEC 14764 IEEE Std 14764-2006 - Software Engineering - Software Life Cycle Processes - Maintenance.* IEEE, 2nd edition, Sept. 2006.

[3] NIST/ECMA. Reference Model for Frameworks of Software Engineering Environments. NIST Special Publication 500-211, August 1993.

[4] J. Siedersleben. *Moderne Softwarearchitektur.* dpunkt-Verlag, 2005.

[5] H. M. Sneed. 20 Years of Software-Reengineering: A Résumé. In R. Gimnich, U. Kaiser, J. Quante, A. Winter, editors, *Workshop Software Reengineering*, *LNI* 126, pages 115–124. GI, 2008.

[6] A. Wasserman. Tool Integration in Software Engineering Environments. In *Software Engineering Environments*, pages 137–149. Springer, 1990.

[7] A. Winter and J. Ebert. Using Metamodels in Service Interoperability. In *Software Technology and Engineering Practice, 2005. 13th IEEE International Workshop on*, pages 147–158. IEEE, 2006.