SENSEI Applied: An Auto-Generated Toolchain for Q-MIG

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

1 Introduction

Large software evolution, migration, or reengineering endeavors require integrated tooling to support their specific goals [1]. While some functionality is project-specific, for many standard software evolution tasks, tools are readily available. Those tools usually provide little means for interoperability, making integration a tedious and error-prone struggle. Furthermore, software evolution projects must usually follow iterative processes – even with fully elicited requirements, subjected legacy systems, being large, complex, and undocumented, obscure the view to a clear path through a project. Rigid, ad-hoc tool integration impedes experimentation, encumbers adaption and extension, and overall slows down the project.

SENSEI (*Software EvolutioN SErvices Integration* [2]) is a conceptual framework developed to ease the toolchain-building process, by combining service-oriented, component-based, and model-driven techniques. SENSEI provides the means and structures to describe required functionality and their interplay as *services* and *orchestrations*, respectively, and enables automatic mapping to appropriate implementing *components* and auto-generation of integration code.

The utility of SENSEI has been put to the test by using it to (re-)build the toolchain for the Q-MIG [3] research project. This paper aims to demonstrate SENSEI's advantages by explaining its application, and comparing it to "manual" toolchain-building. To this end, Section 2 gives a brief overview over the goals of Q-MIG. Section 3 outlines the key principles of SENSEI, and how it is practically applied, using Q-MIG as example. Section 4 exemplifies the utility of SENSEI regarding *flexibility*, *reusability*, and *productivity* using tooling-related issues that arose during the project. The paper concludes with a summary in Section 5.

2 The Q-MIG Project

Q-MIG investigated quality dynamics of software under language migrations from COBOL to Java. Its objectives were to *measure*, *compare and visualize*, as well as *predict* software migration quality. Therefore, a *Quality Control Center* has been developed: a toolchain to complement an existing migration toolchain in support of *a*) researchers studying migration quality, *b*) experts rating the inner quality of systems, and *c*) software migration consultants projecting post-migration quality to improve migration tools, provide insights to clients, and choose migration strategies and tooling according to quality goals. The required tools and toolchains had been developed conventionally, first, as the SENSEI tooling had not been ready when the project commenced.

3 The SENSEI Approach Applied

The key aspects of SENSEI can be summarized along the utilized principles of *service-oriented*, *component-based*, and *model-driven* paradigms, consolidated by *capabilities* in an integrated meta-model. More detail is given in the following, explaining the steps of applying SENSEI (*defining services*, *designing orchestrations*, *adapting and registering components*, and *generating toolchains*), using Q-MIG as example.

Defining Services. First, the required functionality has to be identified and described as services. In SENSEI, a service consists of a name and description of its intended function, consumed inputs and produced outputs with associated (abstract) data structures, and *capability classes* (explained in the following).

Services can either be defined *top-down* or *bottom-up*. The former approach identifies them from relevant publications and diverse software evolution projects [5], to create a catalog of generic, standardized services. If available, services can be picked from the catalog instead of being created for the project. Otherwise, services can be created *bottom-up*, only for a project's required functionalities, giving full control over service design, but potentially leading to project-specific services with lower reuse value. This can be used, though, to fill a catalog incrementally, and refine and generalize its services in the process.

Lacking a comprehensive catalog, the bottom-up approach was chosen for Q-MIG. Services were identified for *parsing*, *calculating metrics*, *visualizing*, *learning and predicting*, as well as *extracting* and *consolidating* data. All services were fitted with input and output parameters, e.g. the parsing service's input is *source code*, and its output is a corresponding *abstract syntax tree*. To be able to refine what kind of source code can be parsed, the service also got a capability class named *programming language*, with COBOL and Java among the possible values. The parameter's types can be restricted according to a particular capability, e.g. Java requires Java source code as input.

Designing Orchestrations. Once all service descriptions are ready, they can be instantiated in an orchestration. A graphical editor [4] is available to support this task, so that service instances can be wired up to define control and data flows. Service instances can be nested in control structures to model concurrent execution, or loops, for example.

The orchestration for Q-MIG's quality measurement starts with parsing. Next, a service to calculate a metric is invoked once for each metric specified in the input, using a loop, while concurrently, a service extracts sub-system nesting information. Finally, the data is consolidated and returned. The orchestration supports COBOL and Java *both*, which is specified declaratively using *required capabilities*; no branching was modeled.

Adapting and Registering Components. To be usable with SENSEI, components have to conform to their services' interfaces. The service-to-component mapping is not one-to-one: for example, the parsing service is implemented by two different components, one for parsing Java, and one for COBOL. Data extraction and consolidation, as well as metric calculation are implemented in a single component.

Generating Toolchains. Lastly, the toolchain generator *SCAffolder* (targeting *SCA*: "Service Component Architecture" as component framework) is fed with the artifacts resulting from the previous steps. It will try to find matches for each service instance in the orchestration. It may select multiple components to realize a single service with different capabilities, and generate appropriate branching logic. SCAffolder leverages service capabilities and associated data type restrictions on them to invoke the right implementation at runtime based on concrete input data.

4 Evaluation

This section compares the conventional toolchain-building approach with SENSEI, again using Q-MIG as example. The comparison is structured according to the criteria *flexibility*, *reusability*, and *productivity*. Three exemplary issues that arose during Q-MIG have been picked to illustrate relevance and particular advantages of SENSEI: (**1**) To integrate a clone detector for the *number of cloned lines* metric within the toolchain, technical interoperability issues became a major selection criterion. A Java tool was selected, because it was the easiest to integrate. (**2**) Some project members had less experience in object-oriented programming, leading to architecture violations. (**3**) Due to legal restrictions, parts of the toolchain had to be run by, and on the premises of, the project's industry partner. The distributed part could not be automated, introducing manual steps, communication overhead between the partners, and a rigorous release process.

Flexibility. SENSEI enables a technology-independent choice of existing tools (1), as it abstracts from interoperability and implementation issues. The target platform SCA offers support for different implementation languages. This helps less experienced developers (2) to create components with familiar techniques, strictly isolated from other parts of the toolchain. The high abstraction level also enables non-programmers (e.g. data scientists, analysts) to partake in designing toolchains. And it abstracts from deployment concerns, easing toolchain distribution (3).

Reusability. Reuse is facilitated through SENSEI by building up a library of components, with interfaces standardized through a service catalog. With SENSEI, adapters will rest with the tools, whereas in Q-MIG (1), it was natural to keep them somewhat "buried" and mixed in within the metric calculation code.

Productivity. SENSEI decreases development effort partly through automation (code generation), and by avoiding redundant developments through added flexibility and easier reuse. E.g., while external tools (1) still have to be adapted to SENSEI's infrastructure, it only has to be done once (possibly even by the tool vendor). The application to Q-MIG has shown that small changes can sometimes be implemented more quickly without SENSEI's imposed structure, but their accumulation may lead to declining evolvability of the toolchain.

The inability to create a gapless, fully integrated toolchain in Q-MIG (3) highlights its importance, as the manual procedures lead to misunderstandings, and markedly slowed down turnarounds. While SENSEI does not currently support distributed toolchains, it can be extended towards it, providing full toolchain control without the need of human intervention. Here though, it remains unclear whether full integration would have been permissible from a legal point of view.

5 Summary

SENSEI structures and partly automates toolchain building to support (not only) software evolution project processes. It facilitates flexibility and reuse, and can thereby help save time and effort. Its application to a concrete project is a first proof of viability.

Achieving the same advantages building toolchains "conventionally", e.g. by adhering to principles like loose coupling and encapsulation, or by "only" using a particular component framework requires more foresight, very disciplined development, and additional implementation effort – something that is hard to keep up under the pressures of a time- or budget-constraint project and evolving conditions and requirements. SENSEI enforces the required structures, and reduces the overall effort through automated integration code generation. The overhead of defining services and adapting component interfaces required at the outset is set off by integration automation. In the long term, it is expected to pay off due to increased reusability.

References

[1] S. E. Sim, "Next generation data interchange: Tool-to-tool application program interfaces," in *WCRE*, 2000, pp. 278–280.

[2] J. Jelschen, "SENSEI: Software Evolution Service Integration," in *Software Evolution Week (CSMR-WCRE)*. Antwerp: IEEE, Feb. 2014, pp. 469—-472.

[3] G. Pandey, J. Jelschen, D. Kuryazov, and A. Winter, "Quality Measurement Scenarios in Software Migration," in *Softwaretechnik Trends*, vol. 34, no. 2. Bonn: GI, 2014, pp. 54–55.

[4] J. Meier, "Editoren für Service-Orchestrierungen," master's thesis, University of Oldenburg, 2014.

[5] J. Jelschen, "Discovery and Description of Software Evolution Services," *Softwaretechnik-Trends*, vol. 33, no. 2, pp. 59–60, May 2013.