Modeling Service Capabilities for Software Evolution Tool Integration

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

1 Introduction

Software evolution activities, like reengineering or migrating legacy systems, depend on tool support tailored towards project-specific needs. Most software evolution tools only implement a single technique, requiring the creation of toolchains by assembling individual tools. These tools come with little to no means of interoperability, e.g. two tools may implement the same functionality, but offer different interfaces. They can also be implemented using different technologies, and expecting input data in different formats. Manual integration is therefore tedious and error-prone, yielding brittle, non-reusable glue code.

SENSEI (*Software EvolutioN SErvices Integration* [1]) aims to ease toolchain building, using service-oriented principles to abstract from concrete implementations, describe software evolution techniques on a high level, and standardize them as *service catalog*. On the service level, processes are modeled as *orchestrations*. SENSEI aims to have actual toolchains automatically generated from orchestrations, by having tools wrapped as *components* according to an appropriate component-based framework providing uniform interfaces, and mapping services to implementations in a *component registry*. Model-driven techniques are used to realize a code generator able to create a toolchain as *composition* of components conforming to a given service orchestration. This avoids having to (re-)write integration code, and facilitates experimentation and more agile processes as changing the toolchain is eased and sped up.

Here, there are opposing requirements regarding the *granularity* of service description detail, as the higher service level demands more abstraction, while on the lower component level much more specificity and technical detail is needed.

The service catalog demands more general services to facilitate standardization, e.g. specify a *calculate metrics* service, not *calculate McCabe metric on Java code*. Such fine-grained descriptions would lead to a catalog cluttered with only marginally differing services, making it hard to identify the right services for a given task. In orchestrations, the high abstraction level hides interoperability issues.

In contrast, service orchestrations do need to be specific about the functionality required, e.g. here it is necessary after all to declare *McCabe* as the actual metric to be evaluated, and *Java* as the data to evaluate it on. Purely technical properties of particular implementations, e.g. that the input Java AST needs to be encoded in a specific XML format, should still remain hidden, though. In a component registry, both functional and technical properties need to be described rigorously, to enable automatic toolchain generation by matching up provided with required functionality, and be able to coordinate tools and accommodate for non-compatible data formats.

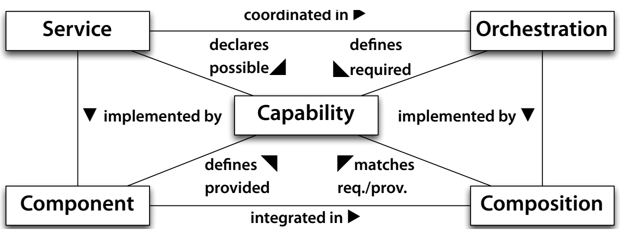

Figure 1: Capabilities in SENSEI.

To bridge between these different abstraction levels, a means for synchronization is needed to map from services to components, and orchestrations to component compositions, representing an executable toolchain.

This paper proposes a simple model to explicitly represent a service's *capabilities*. The model has to support *a*) concise, generic service descriptions, *b*) implementation-agnostic, yet functionally precise orchestrations, and *c*) functionally and technically rigorous component descriptions and service mappings. This allows SENSEI to provide an uncluttered software evolution service catalog, hide interoperability issues in orchestrations, and have sufficient technical detail on components to automatically compose them into toolchains. The following Sec. 2 sketches this model and gives examples. Sec. 3 points to ongoing work and concludes the paper.

2 Service Capability Model

Figure 1 depicts central concepts of SENSEI and their relationships with *capabilities*. An overview of the applications of capabilities in SENSEI is given here (numbered 1 through 4), before revisiting these use cases to describe them in more detail. A distinction is being made between *services*, describing encapsulated units of abstract functionality, and *components*, actually realizing services. On the level of services, capabilities are introduced as a simple mechanism to keep them generic and only *declare possible capabilities* (1) as variation points. Services are selected and coordinated in an *orchestration* to model processes in need of tool support. Here, capabilities are used to instantiate generic services by *declaring required capabilities* (2a) specifically. Components implement the functionality defined by services. Capabilities allow to precisely *define provided capabilities* (2b), which, in contrast to orchestrations, contain additional technical information regarding concrete data types. The latter two use cases are distinct, yet utilize capabilities in the same manner. To create an integrated toolchain, a *composition* of components needs to be derived from an orchestration. Capabilities are leveraged to *constrain component mapping* (3) to only match components to used services which can provide the required functionality. They are further used to *constrain component composition* (4) to only select compatible components or add data transformers for services requiring direct interoperability.

1 Declaring possible capabilities. Generic catalog services use capabilities to declare variation points, establishing a domain of capabilities to choose from when referring to services. A distinction can be made between *functional* and *technical* capabilities. While the former refer to *what* a service can do, the latter are used mostly only with respect to component implementations, to specify *how* a functionality is realized, especially with respect to used data formats. Capabilities can also *restrict* types of service parameters to specific sub-types. This allows to derive capabilities based on runtime data, and can be leveraged during toolchain generation (see Par. 3).

In the service catalog, capabilities serve to keep it uncluttered and clear-cut, which eases service selection for software evolution practitioners.

Example: A metric service may have two classes of capabilities: the *programming language* on which it can operate, and the actual *metrics* it supports. Capabilities of the former class would be *Java*, *COBOL*, *C*, etc., and capabilities of the latter would be *McCabe*, *Halstead*, *inheritance tree depth (ITD)*, etc. The programming language capabilities also define restrictions on the service's data types. E.g. the *Java* capability would restrict the input parameter from a generic source code data structure to Java source code.

2 Defining required and provided capabilities. These two use cases refer to the activities of selecting services for a service orchestration, and registering components for service implementation, respectively. In both cases, capabilities are chosen from the domain declared in the service catalog to narrow a service's functionality down to specific ranges. Here, the capability mechanism allows tool builders to accurately specify their tools provided functionality, and practitioners to define and delimit functionality as required for their projects' tasks. While the latter will mostly only be concerned with *functional capabilities*, the former also needs to specify *technical capabilities*, i.e. what concrete data types components expect as input or make available as output result. Technical capabilities are revisited in Par. 4.

Example: To use a metric service in an orchestration, capabilities are chosen according to project needs. Assuming the project entails evaluating metrics on both Java and COBOL code, and calculating McCabe on both languages, as well as calculating inheritance tree depth (ITD) on Java only, capabilities would look like this:

$$\begin{pmatrix}\text{Java}\\\text{McCabe}\end{pmatrix}, \begin{pmatrix}\text{COBOL}\\\text{McCabe}\end{pmatrix}, \begin{pmatrix}\text{Java}\\\text{ITD}\end{pmatrix}.$$

Capabilities are used likewise to register components implementing a metric service. This example assumes there are two implementing components, *JMetrics* and *CMetrics*, with the following capabilities declared:

$$\text{JMetrics}: \begin{pmatrix}\text{Java}\\\text{McCabe}\end{pmatrix}, \begin{pmatrix}\text{Java}\\\text{ITD}\end{pmatrix}; \text{CMetrics}: \begin{pmatrix}\text{COBOL}\\\text{McCabe}\end{pmatrix}.$$

For toolchain generation, synchronization of required capabilities from orchestrated services with provided capabilities of registered components is needed. This is covered by the following two use cases.

3 Constraining component mapping. The information provided via the previous use cases through capabilities can be exploited to automatically find components for services used in orchestrations, by matching required to provided capabilities. Since components may implement more than one service, each with the provision of multiple capabilities, the same component can be chosen for distinct services and their required capabilities. A single orchestrated service might also be mapped to different components, each providing at least one of its required capabilities.

Example: In the previous example, no single component satisfied all required capabilities. Instead, the *JMetrics* and *CMetrics* tools need to be combined to match all capabilities. Also, the orchestration does not (and need not) directly model when a specific capability should be chosen at runtime – this information is embodied in data type restrictions. Using restrictions, the required branching logic to check input data at runtime and select a component, can be generated automatically.

This allows to automate the creation of potentially complex integration logic using a *constraint solver*, thereby further easing the task of toolchain design.

4 Constraining component composition. When choosing components to compose in conformance with an orchestration, additional constraints have to be taken into consideration, chiefly regarding concrete input and output data types and representations. These additional constraints are expressed by *technical capabilities* in the component registry, and, if a scenario requires using a specific data type, also in orchestrations.

Example: The metrics service's inputs are the metric to be evaluated, and the data to evaluate it over. A component implementing this service has to take several design decisions to concretize these abstract data structures. If the component has the capability to evaluate metrics over Java ASTs, the actual Java meta-model understood, and the expected data representation (e.g. XML, JSON, or a binary format) has to be specified.

Toolchain generation can leverage these constraints to either select components with compatible data outputs and inputs for directly interoperating services, or place an appropriate transformer in between. This facilitates reuse of data transformers through the creation of a library.

3 Outlook

This paper introduced the notion of *capabilities* in the context of the SENSEI software evolution tool integration approach, to control the granularity level of service descriptions, and enable automatic toolchain generation. While SENSEI focuses on the field of software evolution, as toolchain building is of particular importance here, the approach to tool integration is a general one.

Current work is focused on implementing a toolchain generator based on TGraphs and the GReTL transformation language [2], solving constraints formed by capabilities, for automatic creation of versatile and complex toolchains based on clear and incisive orchestrations.

References

[1] J. Jelschen, "SENSEI: Software Evolution Service Integration," in *Softw. Evol. Week — Conf. Softw. Maintenance, Reengineering, Reverse Eng.* Antwerp: IEEE, Feb. 2014, pp. 469—-472.

[2] J. Ebert and T. Horn, "GReTL: an extensible, operational, graph-based transformation language," *Softw. Syst. Model.*, 13(1), pp. 301–321, May 2012.