A Description Model for Software Evolution Services

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

Abstract—*Software evolution encompasses a wide variety of activities to analyze, reverse engineer, transform, and visualize software systems, requiring properly integrated tool support. Many tools are available, however, most implement only a single technique, and offer little to no interoperability. Towards a service-oriented integration approach, this paper presents a service description model, developed along two case studies, as a basis for cataloging and standardizing activities and techniques as services. As proof-of-concept, a prototype has been implemented, realizing one of the case studies.*

Keywords—Software Evolution, Tool Interoperability, Service-Orientation

I. Introduction

Software systems continue to *evolve* after their initial deployment. Over time, these systems "erode", and become harder to maintain, yet often embody a substantial business value for their owning companies. Therefore, effort has to be spent on keeping software systems evolvable.

To this end, software evolution offers activities and associated techniques to analyze, reverse engineer, transform, and visualize software systems. Due to the large size of industry-scale software systems, these activities have to be tool-supported to be feasible. There are many tools available, however, most implement only a single technique, and offer little to no means for interoperability. Most tool suites, which combine several techniques (e.g. Bauhaus [1]), focus on activities for program comprehension only, and offer no means for project-specific tailoring.

Borchers [2] observes that a large amount of the tool infrastructure is project-independent, yet goal-specific tailoring is always necessary, as well. Building such toolchains requires hand-written "glue logic" and ad-hoc functionality (e.g. to transform data formats, or make control flow decisions), which yields tightly-coupled tools, little reusability, and inflexible projects. Ideally, software evolution practitioners should be enabled to focus on their project-specific goals and activities as much as possible, and not having to consider tool integration issues.

To facilitate this, software evolution activities have to be described on a high level, to abstract from concrete implementations, focus on specifying required functionalities, and standardize them. The approach pursued by the authors is based on taking a *service-oriented view* to model and decompose software evolution activities on a conceptual, implementation-agnostic level, and subsequently fill a *catalog of software evolution services*, which can serve as the basis for a component-based framework to largely automate integration. Envisioned is a solution to allow practitioners to model only their project's workflows in terms of required services and their coordination, and generate an integrated solution by mapping services to concrete implementations which abide by the services' specifications.

In this paper, a *description model for software evolution services* is developed along two case studies, modeling the activities of *architecture reconstruction* and *software measurement* as services, to identify the information required to comprehensively specify and catalog services. A service model of software measurement was created, and a prototypical realization using existing components was implemented for validation.

The remainder of this paper is outlined as follows: Section II presents the general idea of using a service-oriented approach to software evolution tool interoperability, and Section III discusses related work. Section IV introduces the case studies, and requirements derived from them. The service description model based on them is introduced in Section V. Section VI describes the proof-of-concept implementation of the software measurement case study. The paper closes with a summary and an outlook in Section VII.

II. Service-Oriented Tool Interoperability

In this paper, a service-oriented view towards tool interoperability is taken. The core idea is to identify software evolution activities and the techniques and tools used to carry them out, and to lift them onto the abstract, generic, and implementation-agnostic, yet rigid functional description level of *services*. This leads to a *software evolution service catalog*, serving as taxonomy of the field, as reference for standardized interfaces, and by extension, as basis for an automated tool integration framework.

The approach is referred to as SENSEI (*Software EvolutioN SErvices Integration*), and embodies the following vision: A *service catalog*, containing abstract, generic, yet rigorous descriptions of software evolution services, is used to pick the services required to support the project's activities and achieve its goals. A *service orchestration* is modeled, defining how the selected services are coordinated to interact with each other. A *component registry* provides a mapping between abstract services and concrete implementations by tools. These are encapsulated in, or developed as, *components*, conforming to the component model of an underlying framework. A set of model *transformations* embodies the mapping to a specific implementation technology. With all these artifacts, a *model-driven code generator* produces the required platform-specific code which realizes tool integration.

The final result is a set of tools integrated into an application framework able to execute the specified software evolution processes.

SENSEI is based on concepts from *service-oriented*, *component-based*, and *model-driven* software engineering paradigms. In particular, there is a clear and important distinction to be made between abstract services, which only contain a specification of a functionality with standardized interfaces, and components, which provide actual implementations and conform to the realized services' interfaces.

III. Related Work

A project which shares several ideas with SENSEI is *SOFAS* [3]. It also introduces services, although here, the term is closely related to the implementation technology, namely RESTful web services. Similar to SENSEI, it aims at integrating tools by orchestrating services into workflows. It further shares the concepts of a composer to enact orchestrations, and a service catalog with SENSEI. The approach is restricted to software analysis activities, which is explicitly exploited by making assumptions about their uniformity. It is focused on the actual framework and the services it can provide, and less on providing a generic, technology-independent concept as SENSEI is.

SENSORIA [4] was a large research project running for five years until 2010, aimed at creating a comprehensive software engineering approach for software systems based on service-oriented architecture. It developed many concepts for service-oriented engineering, one being the model-driven generation of integrated software systems from higher-level descriptions like BPEL-based orchestrations [3]. The project was neither focused on software evolution, nor was tool interoperability its central concern, though.

Similar ideas lie at the core of work by Kraemer et al. [5], [6], who also aim to automate the mapping of orchestrations to executable code, and closing the gap between high-level, process-oriented modeling and the implementation level, using model-driven techniques. Their approach involves transforming an orchestration as a UML activity diagram into a UML state machine, as an intermediate step to generate code.

Another approach for model-driven tool integration, aimed at software development toolchains, is ModelBus [7], and the related *reference technology platform* defined in the *CESAR* project [8]. Its use cases are the integration of tools like requirements management software, integrated development environments, software modeling tools, etc.

Also in the domain of integrating software development tools are the works by Biehl et al. [9]. It is similar to SENSEI in

several regards, e.g. it is based on services and their orchestration (using their domain-specific tool integration language *TIL*), and model-driven techniques to derive executable toolchains, for which they also chose *SCA* as target platform (compare Section VI). They address a different domain, with use cases from embedded software development. Discovery and description of relevant services, their cataloging and their standardization is not addressed.

In summary, approaches for tool integration in domains other than software evolution do exist, and are in part built around similar ideas as SENSEI. These ideas, e.g. using services and their orchestration as high-level descriptions, and model-driven techniques to derive (component-based) toolchains have therefore been proven practical means for tool interoperability. However, these approaches have been tailored for different application domains. The approaches known to the authors explicitly addressing software evolution are less generic than SENSEI, or do not cover the whole field. Moreover, no standardization of services is offered, such that implementations of conceptually equal services will still be incompatible, impeding reuse and the flexibility, e.g. to exchange one implementation for another without any changes to the integration logic. To achieve this level of interoperability, SENSEI aims to establish a catalog of software evolution services, including a seamless approach to the orchestration of appropriate implementations.

IV. Case Studies

To develop and refine the service description model needed to capture all relevant information of a catalog of software evolution services, and an integration framework based thereon, two case studies have been conducted, aimed at identifying services and the kind of information needed to fully describe them.

Each case study analyses a well-established software evolution activity: architecture reconstruction, and software measurement, respectively. A literature survey was performed to discover different variants, derive a generic process describing the activity, and to identify sub-activities. The software evolution activity was then modeled in terms of services, starting out with a basic description model focused on services' inputs, outputs, and possible decomposition.

As guidance, two different modeling perspectives were considered:

Software Evolution Activity as a Service. This is an outside view of a service. It is concerned with the service's interface (its expected inputs, provided outputs, and utilized data structures) and its semantics. It contains the information expected to be found in the service catalog.

Services for Software Evolution Activities. This is an inside view of service. While it does not make any implementation-specific assumptions, it does describe a service on a process-oriented level in terms of used sub-services. Its consideration therefore helps to identify the services relevant to the activity. It contains the information expected in a generic service orchestration.

New requirements for the service description model were derived from issues which arose while modeling the case studies' activities. Primarily these features deal with providing more flexibility in service orchestration. These requirements, in turn, have been the basis for the design of the description model. In the following, the case studies on modeling *architecture reconstruction* (Section IV-A) and *software measurement* (Section IV-B) as services are briefly presented. Next, the resulting requirements are described (Section IV-C).

A. Architecture Reconstruction

Evolving software systems are subject to continuous change. Modifications to be made to account for changing requirements or system environment often clash with the existing structure, and erode it over time. Documentation often becomes outdated or gets lost, so that the system's architecture has to be reconstructed out of the underlying source code. This aspect can be realized with *clustering algorithms*, on which this case study is focused. An architecture reconstruction taxonomy by Ducasse and Pollet [10]

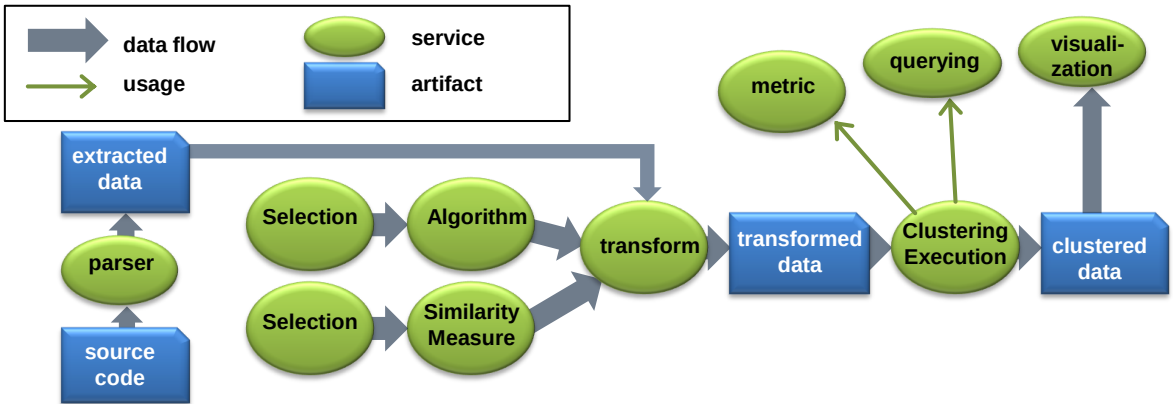

Figure 1: Services for architecture reconstruction.

	In	Out
Algorithm	input data (to be clustered), cut-point, similarity measure	clustered data
Similarity Measure	input data	calculated similarity values
Clustering Execution Service	algorithm, similarity measure	clustered data

Figure 2: Inputs and outputs of services for architecture reconstruction (Wiggerts [16]).

names several alternative techniques. Clustering algorithms group similar entities such as classes or methods into clusters. A *similarity measure* determines the closeness of entities.

There are many different clustering algorithms and tools in literature: Well-known algorithms and tools are *MoJo* presented by Tzerpos et. al [11], *Bunch* by Mitchell and Mancoridis [12], *Craft* introduced by Mitchell et al. [13] to evaluate clustering algorithms, ACDC by Tzerpos et al. [14], and one of the most recent tools is SCuV published by Xu et al. [15]. For maximum flexibility, it is desirable to have a clustering service able to work with different clustering algorithms and input data formats.

A possible way to realize such a service reusing pre-existing services is shown in Figure 1. In an initial step, a *parser* extracts information from source code and stores entities and their relationships in a data format suitable for further calculations. The first step in the clustering service itself is to select a proper clustering *algorithm* and *similarity measure*, since both affect the quality of the clustering result [17]. Due to the fact that this is not a trivial task, *selection* services offer a meaningful support: They embody a selection approach for clustering algorithms as, for example, presented by Shtern and Tzerpos [18]. After choosing the algorithm and a similarity measure, a transformation of the extracted data into the input data format of the chosen algorithm has to be done. Providing these selection services allows for more scalability of service realization regarding efficiency and precision. A possible *transformation* can be chosen at runtime. The transformed data is transferred to the *clustering execution* service, where it can be decomposed into clusters by executing the algorithm and similarity measure chosen previously. This service may utilize further sub-services, like *metrics* and *querying* to evaluate properties of clustering entities, or the chosen similarity measure. In the end, the clustered data is visualized using an existing *visualization* service.

Based on a classification by Wiggerts [16], the input and output parameters depicted in Figure 2 have to be specified in the service catalog for the new services. Modeled this way, the clustering service is able to realize all existing clustering algorithms and to execute them with different input data formats.

B. Software Measurement

During the software development and evolution process, it is difficult to evaluate the

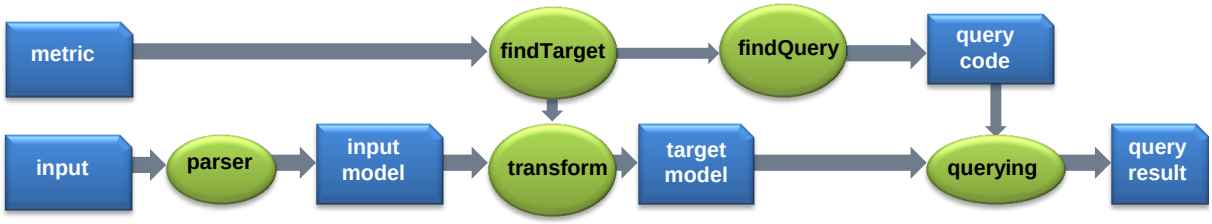

Figure 3: Services for software measurement.

quality and quantities of software artifacts to guarantee the quality of the whole software product, including the development process itself. *Software metrics* provide considerable help to measure the quality and quantity of software artifacts, and aid in program comprehension and directing further evolution of the software. The IEEE defines a metric as *"A function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality"* [19].

The input depends on the current software development phase. During analysis it is important to measure the quality of a given requirement specification or other documentation, whereas during implementation it is more significant to determine the quality of the source code. Possible input data for metrics are therefore documentation as well as source code. The wide range of possible input data formats can be seen in the classification of Fenton and Pfleeger [20].

This aspect means that a metrics service has to be flexible concerning the input data structure. Nevertheless, it is significant to have a uniform service for the calculation of software metrics, due to the fact that software measurement affects all phases of software development, and thereby affects various developers and employees with different skills. Having a uniform service eases the skill adaption to new metrics.

An approach to realize software measurement using services is depicted in Figure 3. The metric service needs two inputs: The metric, and the input to measure it on. As a first step, a *parser* extracts the input data into a format that can be used for further calculations. Due to the fact that the input format can be documentation as well as source code, or other resources of the software project, an appropriate parser has to be chosen. Therefore, the input has to provide information about the grammar and the format of the input – for example in form of a meta-model. The parser provides the input model that can be used for further calculations.

Metrics cannot always work directly on the data provided by a parser, e.g. McCabe's *cyclomatic complexity* [21] is defined over control-flow graphs. In these cases, a *transformation* has to be executed to get the target model as input for the metric calculation. A way to resolve to an appropriate transformation service from the required input data structure of the chosen metric has to be found. Here, this is represented by the *findTarget* service.

The *querying* service queries the target model to get the value of the chosen metric. Again, a means to resolve metric names to suitable query expressions has to be provided. This lookup is represented by the *findQuery* service. After this step, the chosen query code can be executed with the help of the querying service on the target model to deliver the query result as result of the software measurement.

C. Requirements

The case studies have been used to discover requirements for a service description model. In the following, observations regarding needed structures to properly model software evolution services, and recurring modeling patterns are described. Each paragraph is summarized by a requirement, which in turn forms the basis of the service description model subsequently designed (Section V).

1) Service Hierarchy: Besides the development of the service description model, which determines what kind of information

goes into the service catalog, and how it is structured, another major goal of the case studies was the discovery of relevant (sub)-services (to actually fill the catalog). This has been done by breaking down the main software evolution activities under study into the steps to be taken to perform them, and describe them as services, as well. Decomposing services into sub-services revealed, for example, services central to software measurement, namely *metrics* and *querying*, can be reused for architecture reconstruction.

Furthermore, including this hierarchical information in the service catalog is useful to organize it, and help catalog users find the right services for their tasks. Modeling services hierarchically also facilitates information hiding to handle complex services.

→ *The service description model must provide a means to arrange services in a decomposition hierarchy based on service usage relations.* (1)

2) Service Variability: In both case studies, services have been modeled in a rather abstract fashion. For example, both case studies require a parser service, which is what one would be looking for in a service catalog. In a given usage scenario, however, a specific parser, capable of parsing source code in the programming language(s) of the system under study is needed. As another example, consider the metric service from the software measurement case study, which has two such "degrees of freedom": one determines which metrics it should be able to calculate (as it cannot be expected that all implementations support every possible metric there is), the other what data structures it is able to process. There is also an interdependency between the two, since metrics are defined over certain data structures.

→ *The service description model must provide a means to model abstract services, and to describe all possible instances.* (2)

3) Data Abstraction Level: As a consequence of the different service abstraction levels required, it must also be possible to model input and output data accordingly. For example, for the generic parser service, the input data structure is source code, and the output data structure an abstract syntax tree (AST). On this level, both data structures cannot be described in much more detail than this. If a usage scenario is given, though, i.e. it is known that the parser must support the Java programming language as input, then the input data structure is known to be conforming to the Java language specification, and an appropriate Java meta model can be defined for the resulting AST.

→ *The service description model must provide a means to model data structures, arrange them in specialization / generalization hierarchies, and establish relationships between service instances and corresponding, specialized data structures.* (3)

4) Higher-Order Services: In the architecture reconstruction case study, it proved useful to be able to delay service resolution until runtime: Here, the actual clustering algorithm is selected based on requirements of the specific task at hand, and passed as an argument to the clustering execution service. Assuming the algorithm itself is also represented as a service, this would make the clustering execution service a *higher-order service* (in analogy to higher-order functions, which take functions as arguments). The algorithm cannot be chosen completely automatically, requiring a corresponding input to the clustering execution service. The selection of an appropriate algorithm can be tool-supported, though (in the case study e.g. by implementing the selection process proposed by Shtern and Tzerpos [18]), leading to the identification of selection services.

In a similar fashion, the *findTarget* service identified in the software measurement case study has to instantiate the generic transformation service with a concrete one at runtime, based on inputs. The *findQuery* service is modeled more simply, only looking up data to parameterize the *querying* service.

→ *The service description model must provide a means to delay service resolution until runtime.* (4)

5) Service Types: Both case studies showed that, besides services stemming from the problem domain, e.g. clustering for architecture reconstruction, a need for additional

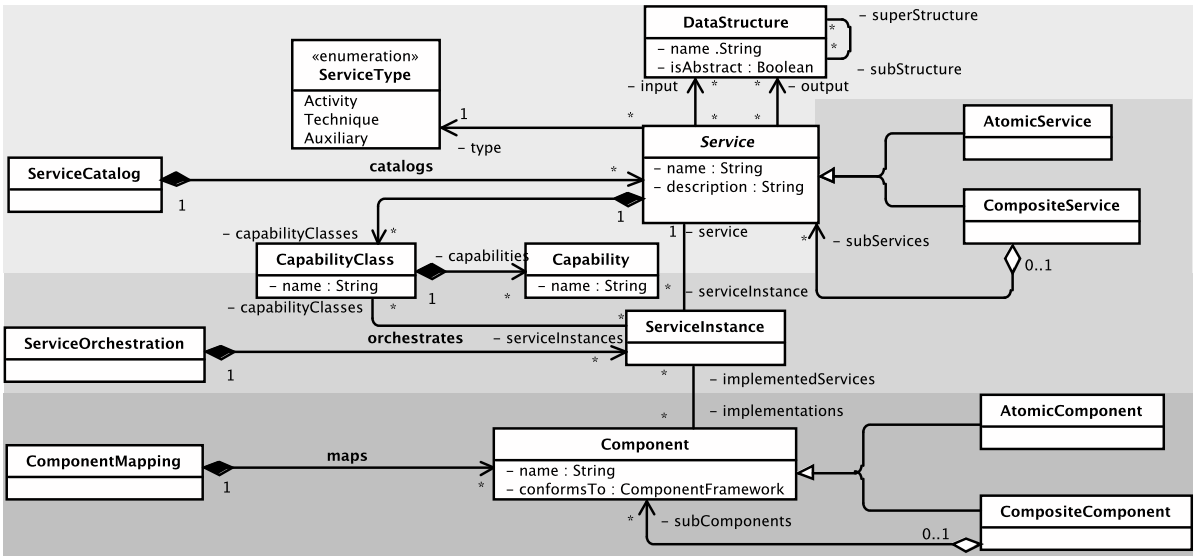

Figure 4: The service description model.

services arose for purely technical reasons, e.g. data transformation services. This is considered an important distinction: technical issues should be hidden from the orchestration designer, who is tasked with designing the workflows of his software evolution project to have them tool-supported, as much as possible. A tool builder, who is concerned with implementing a service in a component, might be more interested in technical subservices he might be able to utilize. In this regard, a distinction between these different types of services helps to organize the service catalog. Another reason to differentiate along these lines is reusability: mixing in technical aspects into a domain-specific service makes it hard to reuse, while the technical part cannot be reused independently, at all (cmp. software "blood types" by Siedersleben [22]).

Among the more auxiliary services encountered, some recurring patterns could be observed: the most obvious helper services are data transformations. Further helper services are those to resolve to concrete service instances at runtime, like *findQuery*. It might be desirable to keep such auxiliary functionality hidden away from practitioners, and have the integration framework take care of these aspects automatically, if possible.

→ *The service description model must provide a means to classify different types of services.* (5)

V. Service Description Model

Based on the requirements from the case studies, and in continuation of previous work [23], a service description model was designed, to capture all interoperability-relevant properties of software evolution services. The model is depicted in Figure 4. It can be explained from three viewpoints, each explained in the following: the *service catalog* viewpoint (Section V-A), the *service orchestration* viewpoint (Section V-B), and the *component mapping* viewpoint (Section V-C). An approach taking artifacts from all three viewpoints and automatically deriving a *toolchain integration* solution is briefly sketched, as well (Section V-D). The software measurement case study has been modeled according to the newly developed model, and serves as ongoing example. Based on this model, a prototypical integration solution has been realized (Section VI).

A. Service Catalog

This viewpoint describes services as listed in a service catalog, and therefore roughly corresponds to the "*as-a-service*" perspective. Each artifact of this viewpoint is an entry in the service catalog. The table in Figure 5 depicts a simplified example, listing a brief description of the service's semantics, type, input and output data structures, capability classes, and constraints (see below).

To keep the catalog clear and uncluttered (Requirement 2), these services ought to be

generic, e.g. by having a single service *CalculateMetric*, instead of one service for each possible combination of supported metrics and programming languages. To this end, each *Service* may define several *Capability-Classes*, describing variation points along which concrete manifestations may differ. In the aforementioned example, this would be the sets of *supported metrics* and *programming languages*, respectively. A capability class aggregates a set of capabilities, and is defined with all possible elements in the service catalog. The capability class *programming language*, for example, would contain *Java*, *C++*, *COBOL*, and so on.

The capability approach implies a requirement for a corresponding mechanism on data structures (Requirement 3). Data structures can therefore form generalization hierarchies, and be marked abstract (requiring a non-abstract subtype at runtime). This way, an abstract *AST* data structure can be specialized to a *JavaAST* or a *COBOL_AST*, essentially introducing polymorphism. Notice that there are usually direct interdependencies between the actual data structures which are handled by a service instance, and its required capabilities. Services therefore have to specify how a service instance's capabilities determine its concrete input and output data structures, using *constraints*. In the example in Figure 5, they express the correspondence between *ProgrammingLanguage* capabilities and input data structures, and that instances of the *CalculateMetric* service can only calculate those metrics for which they possess the corresponding capability. A capability class with a prime ($'$) indicates a reference to a specific subset chosen at design time, as opposed to the whole capability class containing all possible capabilities.

To fulfill Requirement 5 and distinguish between different kinds of services, they have a *ServiceType*, based on the way they are used (cmp. Siedersleben [22], or Cohen [24] for similar classification schemes): *Activity services* embody a software evolution activity, e.g. architecture reconstruction or software measurement, and are always aimed at a specific goal. *Technical services* realize and

CalculateMetric	
Description	Calculates a metric (provided by its name) over source code and returns the result as a single real number.
Type	Activity
Input	code : SourceCode
	metric : MetricEnum
Output	result : Real
Capability Classes	ProgrammingLanguage
	AvailableMetrics
Constraints	
$\forall x$: x $\in$ AvailableMetrics$'$ $\Leftarrow$ metric = x Java $\in$ ProgrammingLanguage$'$ $\Leftarrow$ typeOf(code) = JavaCode COBOL $\in$ ProgrammingLanguage$'$ $\Leftarrow$ typeOf(code) = COBOLCode	

Figure 5: *CalculateMetric* as a service.

support activities, and are possibly applicable to different ends, but are not directly useful on their own. For example, formal concept analysis is an alternative technique to be used for architecture reconstruction in place of clustering [10]. However, neither technique would usually be performed as an end in itself, but rather in the context of a concrete, goal-oriented activity. An even more primitive form are *auxiliary services*, which are only necessary for integration purposes, e.g. resolving concrete services (and corresponding components) based on input data at runtime, like the *findQuery* service of the software measurement case study does. Auxiliary services are simple by nature, and might not require dedicated components implementing them, but rather have the necessary logic be created as required by an automated integration framework. The tasks of auxiliary services can be modeled declaratively as *constraints*, keeping orchestrations clean from integration-related concerns.

B. Service Orchestration

This viewpoint captures the artifacts which are created when orchestrating services, i.e. coordinating existing services to work together to support a specific software evolution activity ("services-for" perspec-

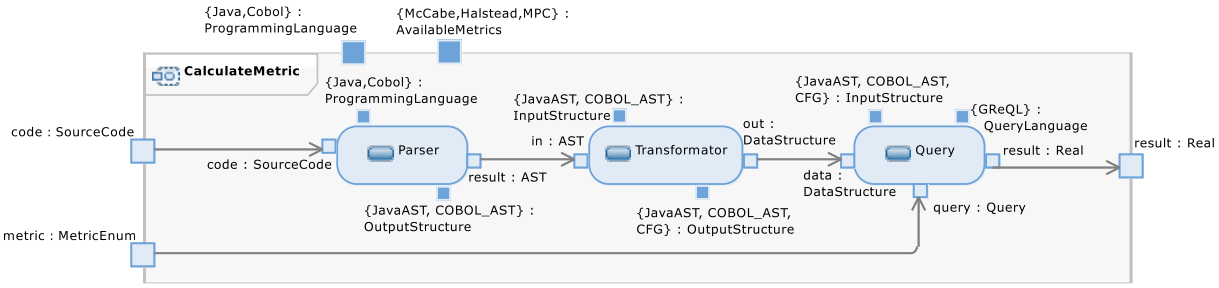

Figure 6: Orchestration of service instances to calculate several COBOL and Java metrics.

tive), and thereby forming a more complex service. The central concept of this viewpoint is *ServiceInstance*: it represents a service in a concrete usage scenario, with only a subset of all possible capabilities of the generic catalog service. Service instances are orchestrated using an appropriate orchestration (process, workflow) description language, or a subset thereof, e.g. UML Activities, BPMN [25], BPEL [26], or YAWL [27]. These languages provide concepts like *data flows*, *decision nodes*, etc. which have not been incorporated in the model in Figure 4.

Each artifact of this viewpoint is an orchestration of service instances, a high-level description of a software evolution activity to be tool-supported. Figure 6 shows an example of an orchestration for the *CalculateMetric* service using an UML activity diagram, extended by small dark squares to depict capability classes. Here, the capability classes have been subset to have this service instance support calculating McCabe, Halstead, and a "methods-per-class" (MPC) metric on Java and COBOL code. There are more capabilities to be set on the sub-services, however, except for the *QueryLanguage*, which has been fixed to *GReQL* [30], they can be derived from the outer service's capabilities. Constraints can be used to express their relationships. Also, this orchestration contains no auxiliary services, as opposed to the initial model of the software measurement case study (*findTarget*, *findQuery*). Constraints can be used in their stead, providing a solution to Requirement 4. The following constraint is an example excerpt of a constraint on the service orchestration level to derive a concrete query expression from capabilities (the chosen *QueryLanguage*: GReQL) and runtime data (the chosen *metric*: McCabe):

```
1 CalculateMetric.metric = "McCabe"
2 ∧ GReQL ∈ Query.QueryLanguage ⇒
3 Query.query =
4     "count(E{ControlFlow}) −
5      count(V{BasicBlock}) +
6      count(V{Procedure}) ∗ 2"
```

Decomposition of services into sub-services (Requirement 1) is also viewed at the orchestration level. To support this in the description model, the composite pattern [28] has been applied over class *Service* (in, by analogy, over class *Component*). The composition is based on service *usage*: a complex service is decomposed into those sub-services it can use to realize its functionality. On the service catalog level, this is a purely hypothetical relationship, serving e.g. as a hint for tool developers which services might useful when implementing a complex service instance. A tool developer might still choose to ignore this, and realize a service instance monolithically. Looking at the orchestration viewpoint, the relationship is visible, as an orchestration shows all service instances it is using. However, whether a used service instance is realized by a single, monolithic component, or can be decomposed itself into more basic sub-services, can be handled completely transparent. The other way around, orchestrations can also be seen as establishing more complex, composite service instances from more basic ones. By generalizing from service instances back to services, such services can then be placed in the service catalog again.

C. Component Mapping

This viewpoint associates service instances with actual implementations in the

form of *Components*. Like services, components can be hierarchically decomposed into *AtomicComponents* and more complex *CompositeComponents*. As components cannot be expected to realize the generic reference services of the catalog directly, i.e. supporting all possible capabilities, the mapping is done by having a component reference one or more service instances, which represent provided services, and the extend to which they are realized. Notice that this viewpoint does not model actual components, but references them to establish the mapping to services.

Each artifact of this viewpoint is an entry in a registry, recording a mapping between a concrete component (containing a tool), conforming to a specific component framework, and a service instance, representing the implemented service and supported capabilities.

D. Toolchain Integration

Using an artifact from each viewpoint – a service orchestration modeling the desired workflow to be tool-supported, a service catalog containing the definitions of the services used, and a component registry, providing mappings to implementations for those services – the creation of an appropriate toolchain can be automated. This can be done using model transformations to realize a *code generator*. It requires two sets of transformations: one to map to a specific integration platform, e.g. a component framework like *Service Component Architecture (SCA)* [29], and another one to derive data types from data structures by selecting an appropriate data representation technology, e.g. TGraphs [30], SDO [31], or EMF [32].

A component registry is assumed to only contain components conforming to the same framework for this to work. The data representation may vary, though. Requiring all components to use the same data representation technology might be too constraining, especially for integrating already existing tools. Encapsulating such tools in a simple component wrapper is comparatively easy, while changing the data representation would either require to have tools adapted internally, or to use data format transformations. Such transformations are best kept separate from components for reusability.

The output of the code generator is an application conforming to the selected integration platform, containing all the required components implementing service instances used in the orchestration, plus one additional component, called a *composer*, which is responsible for enacting the orchestration.

VI. Validation

As a proof-of-concept, the *CalculateMetric* service, as orchestrated in Figure 6, has been realized based on *Apache Tuscany's Service Component Architecture* (SCA) implementation [33]. SCA-conforming components were implemented for all the utilized sub-services, and mapped, appropriately: *ParseComponent* implements the *Parser* service, *TransformComponent* the *Transformator* service, and *QueryComponent* the *Query* service. Parsers by *pro et con* for COBOL and Java [34] were used, JGraLab's [30] GReQL facilities provided querying, as well as data representation in the form of TGraphs. Rudimentary transformations implemented previously [35] were used to create control flow graphs from abstract syntax trees of COBOL and Java programs. For this prototype, no automatic code generation was used, yet. Instead the job of the model-driven code generator has been carried out manually, i.e. a composer component (*MetricComponent*) to enact the orchestration as designed, as well as all the necessary SCA configuration artifacts, were written by hand. There is also a *FindQueryComponent*, which implements the lookup of queries for metrics, derived from corresponding constraints.

Figure 7 depicts a screenshot of the prototype project, opened in the development environment. To the left, the package explorer lists the service interfaces and classes implementing the components. In the center, a visualization of an SCA composite file is shown, which wires up components with their dependencies – here, the *MetricComponent* in its role as composer is dependent on all other services, to be able to invoke them and pass data along. The right-hand side shows a code snippet of the

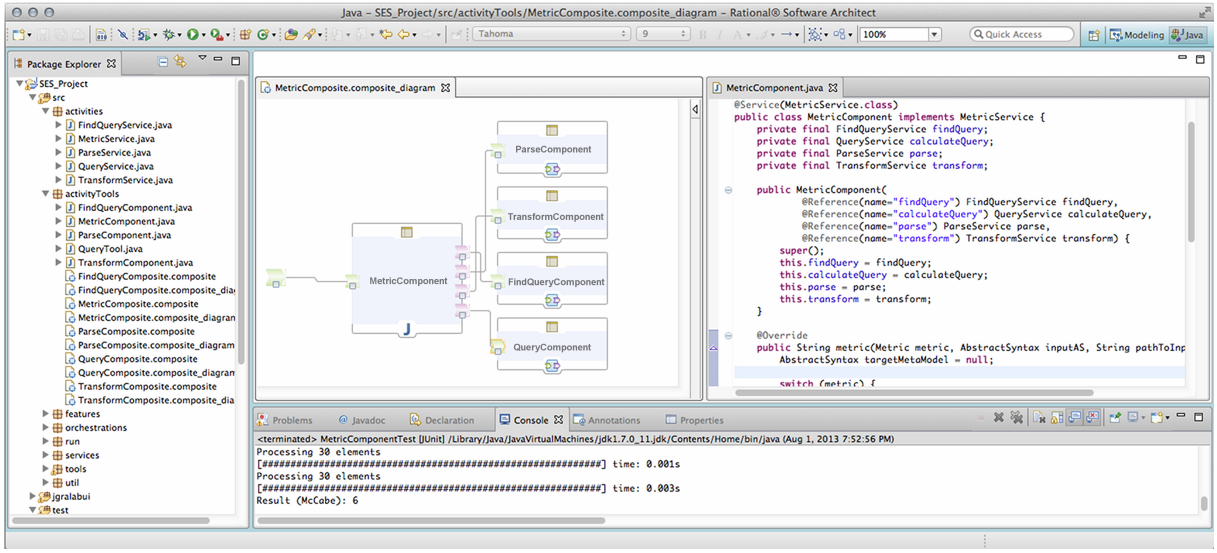

Figure 7: Screenshot of the prototype viewed in an IDE, showing wired-up SCA components, and a code snippet from the implementation.

MetricComponent's implementation. In the pane at the bottom, the tail of the output of a test run can be seen, reporting a McCabe value of 6 for a Java test application.

The prototype works and shows that SCA is a viable target platform. While implementing, attention was paid to only derive the hand-written logic from the service-level artifacts created according to the model.

VII. Conclusion

This paper presented two case studies to model software evolution activities as services, and the description model derived from it, as a step towards the SENSEI-catalog of software evolution services for tool interoperability. A basic method has been devised to conduct the case studies. It is based on two different perspectives on software evolution activities – viewing them *as a service*, and looking at *services for* them. The case studies are further guided by the goals of service *reusability*, *interoperability*, and *automation* of tool integration. While the main goal here was to elicit requirements and derive a service description model, the same method can be applied to discover and describe further services to fill the catalog.

The central contribution of this paper is the developed *service description model*. It enables the creation of a catalog of software evolution services, aimed at providing standardized service interfaces for software evolution activities and techniques, and serving as basis for the implementation of a tool integration framework. The description model introduces service *capabilities*, which enable a generic description of services, while allowing practitioners to designate their specific *required capabilities* for a given project's tool support, and tools to specify *provided capabilities*. *Constraints* on these capabilities can be used to model their interdependencies, and relations with services' data structures in a declarative way. This facilitates the realization of a framework automating tool integration by mapping service orchestrations to concrete components, and resolving the associated constraints to wire them up.

References

[1] A. Raza, G. Vogel, and E. Plödereder, "Bauhaus–a tool suite for program analysis and reverse engineering," in *Reliable Software Technologies – Ada Europe 2006*, LNCS 4006. Berlin, Heidelberg: Springer, 2006, pp. 71–82.

[2] J. Borchers, "Erfahrungen mit dem Einsatz einer Reengineering Factory in einem großen Umstellungsprojekt," *HMD Themenheft Migration*, vol. 34, no. 194, pp. 77–94, Mar. 1997.

[3] G. Ghezzi and H. C. Gall, "A framework for semi-automated software evolution analysis composition," *Automated Software Engineering*, vol. 20(3), pp. 463–496, 2013.

[4] M. Wirsing and M. Hölzl, Eds., *Rigorous Software Engineering for Service-Oriented Systems*, LNCS. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, vol. 6582.

[5] F. A. Kraemer and P. Herrmann, "Transforming Collaborative Service Specifications into

Efficiently Executable State Machines," *EASST*, vol. 6, 2007.

[6] F. A. Kraemer, H. Samset, and R. Braek, "An Automated Method for Web Service Orchestration Based on Reusable Building Blocks," in *IEEE International Conference on Web Services*. 2009, pp. 262–270.

[7] C. Hein, T. Ritter, and M. Wagner, "Model-driven tool integration with modelbus," in *Workshop Future Trends of Model-Driven Development*, 2009.

[8] E. Armengaud, M. Zoier, and A. Baumgart, "Model-based toolchain for the efficient development of safety-relevant automotive embedded systems," *SAE World Congress & Exhibition, 2011*.

[9] M. Biehl, J. El-Khoury, F. Loiret, and M. Törngren, "On the modeling and generation of service-oriented tool chains," *Software & Systems Modeling*, Sep. 2012.

[10] S. Ducasse and D. Pollet, "Software architecture reconstruction: A process-oriented taxonomy," *Software Engineering, IEEE TSE*, vol. 35, no. 4, pp. 573–591, 2009.

[11] V. Tzerpos and R. C. Holt, "Mojo: A distance metric for software clusterings," in *Proceedings of the Sixth Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 1999.

[12] B. S. Mitchell and S. Mancoridis, "On the automatic modularization of software systems using the bunch tool." *IEEE TSE*, vol. 32, no. 3, pp. 193–208, 2006.

[13] B. S. Mitchell and S. Mancoridis, "Craft: A framework for evaluating software clustering results in the absence of benchmark decompositions," in *Proceedings of the Eighth Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 2001.

[14] V. Tzerpos and R. C. Holt, "ACDC : An algorithm for comprehension-driven clustering," in *Proc. of the 7th Working Conf. on Reverse Engineering*. IEEE, 2000, pp. 258–267.

[15] X. Xu, S. Huang, Y. Xiao, and W. Wang, "Scuv: a novel software clustering and visualization tool." in *SPLASH*, G. T. Leavens, Ed. ACM, 2012, pp. 29–30.

[16] T. A. Wiggerts, "Using clustering algorithms in legacy systems remodularization." in *WCRE*, 1997, pp. 33–43.

[17] M. Trifu, "Architecture-aware, adaptive clustering of object-oriented systems," Master's thesis, Forschungszentrum Informatik Karlsruhe, September 2003.

[18] M. Shtern and V. Tzerpos, "Methods for selecting and improving software clustering algorithms." in *ICPC*. IEEE Computer Society, 2009, pp. 248–252.

[19] IEEE, "IEEE standard for a software quality metrics methodology," 1998. http://cow.ceng.metu.edu.tr/Courses/downloadcourseFile.php?id=2681

[20] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, 2nd ed. Boston, MA, USA: PWS, 1998.

[21] T. McCabe, "A Complexity Measure," *IEEE TSE*, no. 4, pp. 308–320, 1976.

[22] J. Siedersleben, *Moderne Software-Architektur: Umsichtig planen, robust bauen mit Quasar*. Dpunkt.Verlag GmbH, 2004.

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

[24] S. Cohen, "Ontology and Taxonomy of Services in a Service-Oriented Architecture," *The Architecture Journal (Online Publication)*, vol. 11, Apr. 2007. http://msdn.microsoft.com/en-us/architecture/bb410935

[25] "Business Process Model and Notation (BPMN) Version 2.0," 2011. http://www.omg.org/spec/BPMN/2.0/PDF/

[26] D. Jordan, J. Evdemon, A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland, A. Guízar, N. Kartha, C. K. Liu, R. Khalaf, D. König, M. Marin, V. Mehta, S. Thatte, D. van der Rijn, P. Yendluri, and A. Yiu, "Web Services Business Process Execution Language Version 2.0," 2007. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

[27] A. H. M. ter Hofstede, W. M. P. van der Aalst, M. Adams, and N. Russell, *Modern Business Process Automation: YAWL and Its Support Environment*. Berlin: Springer, 2010.

[28] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[29] M. Edwards and M. Chapman, "Service Component Architecture Assembly Technical Committee," 2013. https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=sca-assembly

[30] J. Ebert, "Metamodels Taken Seriously: The TGraph Approach," in *12th European Conf. on Software Maintenance and Reengineering*, K. Kontogiannis, C. Tjortjis, and A. Winter, Eds. IEEE, 2008, pp. 2–2.

[31] R. Barack and F. Budinsky, "Service Data Objects (SDO) Version 3.0 Comittee Draft 02," 2009. https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=sdo#technical

[32] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework*, ser. The Eclipse Series, E. Gamma, L. Nackman, and John Wiegand, Eds. Addison-Wesley Professional, 2008.

[33] S. Laws, M. Combellack, R. Feng, H. Mahbod, and S. Nash, *Tuscany SCA in Action*. Manning Publications, 2011.

[34] U. Erdmenger and D. Uhlig, "Ein Translator für die COBOL-Java-Migration," *Softwaretechnik-Trends*, vol. 31, no. 2, 2011.

[35] J. Jelschen and A. Winter, "A Toolchain for Metrics-based Comparison of COBOL and Migrated Java Systems," *Softwaretechnik Trends*, vol. 32, no. 2, 2012.