Service-Oriented Toolchains for Software Evolution

Jan Jelschen
Carl von Ossietzky Universität Oldenburg, Germany
jelschen@se.uni-oldenburg.de

***Abstract*—Software evolution projects need to be supported by integrated toolchains, yet can suffer from inadequate tool interoperability. Practitioners are forced to deal with technical integration issues, instead of focusing on their projects' actual objectives. Lacking integration support, the resulting toolchains are rigid and inflexible, impeding project progress. This paper presents SENSEI, a service-oriented support framework for toolchain-building, that clearly separates software evolution needs from implementing tools and interoperability issues. It aims to improve interoperability using component-based principles, and provides model-driven code generation to partly automate the integration process. The approach has been prototypically implemented, and was applied in the context of the Q-MIG project, to build parts of an integrated software migration and quality assessment toolchain.**

I. INTRODUCTION

Large software evolution, migration, reengineering, and modernization projects require a combination of different techniques to analyze, reverse engineer, transform, and visualize (legacy) software systems under evolution. As each project has different goals, toolchains supporting their processes need to be tailored individually to their specific requirements [1]. Many tools exist, yet mostly only implement a single technique, and are usually not designed for interoperability. The lack of interoperability of software evolution tools is a general challenge of the field, recognized as such, e.g. by Borchers [2], Sim [3], Müller et al. [4], Jin and Cordy [5], Mens et al. [6], as well as Ghezzi and Gall [7].

For each software evolution project, a toolchain has to be built by selecting the techniques required, finding appropriate tools implementing them (or creating custom tools), and then integrating these tools. With little to no means of interoperability, this involves creating a lot of *glue code* for adapters, data transformers, and "wiring", a tedious and error-prone task. It yields brittle and inflexible toolchains, as extending or changing the toolchain, or swapping one tool for an alternative implementation, will require to also write new glue code. Consequently, this code is also non-reusable, as it is usually hard-wired to specific interfaces of the tools glued together.

This rigidness of ad-hoc-integrated toolchains poses a problem, as software evolution projects must usually follow iterative processes (e.g. SOAMIG [8]). Toolchains which cannot be easily adapted and extended according to changing project parameters impede experimentation, and slow down overall progress.

To address these toolchain-building challenges, practitioners have to be freed from technical interoperability issues, as much as possible. For this, an abstraction layer is needed, that clearly separates the concerns of software evolution practitioners from those of tool developers and integration specialists. Furthermore, standardization mechanisms able to cover the whole field of software evolution techniques and tools are needed for better interoperability, and to promote reusability. In addition, standardized, uniform interfaces enable automation of large parts of actual tool integration, facilitating reduced effort and rapid toolchain adaptation for increased project agility.

This paper presents the SENSEI approach ("*Software EvolutioN SErvices Integration*"). Its core objective is the provision of a support framework for the complete toolchain creation process in software evolution projects.

Based on *service-oriented* principles, SENSEI introduces a conceptual abstraction layer over software evolution tools, which only reveals their provided functionality – its *services*, but hides interoperability issues that arise due to different implementation technologies and concrete data formats. *Component-based* techniques are used to standardize tools behind unified interfaces. *Capabilities* are introduced as a means to organize services, and to declaratively specify properties required of services, or provided by components. This information is leveraged to automatically match services to appropriate component implementations, which are then combined into an executable toolchain using a *model-driven* transformation and code generation approach.

The data integration aspect (cf. Wasserman [9]), i.e. integrating and synchronizing models of exchanged data, a complex field of study in itself, is not a focus of SENSEI. It is (to differing extents) addressed by other approaches [5], [10]–[12], which can be considered complementary to SENSEI.

To show feasibility and applicability, SENSEI's concepts and automation tools have been prototypically implemented, and have been used to build parts of the toolchain of the software evolution project Q-MIG [13]. Q-MIG[1] was aimed at "building a quality-driven, generic toolchain for software migration". To this end, both existing and newly developed tools had to be integrated into a single, two-part toolchain: a migration toolchain for COBOL-to-Java translation, and a complementing quality control center, for the measurement, comparison, and prediction of software quality of systems undergoing migration. The project was run iteratively, and its course had to be adapted several times due to unforseen circumstances, which also forced the toolchains to be adapted, accordingly. The project will be referred to throughout the paper as an example.

The remainder of this paper is structured as follows: Section II elicits requirements for a support framework, along the steps needed to build toolchains. Next, previous and related work is presented in Section III, and assessed with respect to those requirements. The SENSEI approach is detailed in Section IV. Its prototype implementation, and its application to build the Q-MIG toolchain, is explained in Section V. The paper closes in Section VI with a summary of SENSEI's contributions.

[1]Q-MIG was a joint venture of *pro et con GmbH* and the *University of Oldenburg*. It was funded by the Central Innovation Program SME of the German Federal Ministry of Economics and Technology – BMWi (KF3182501KM3).

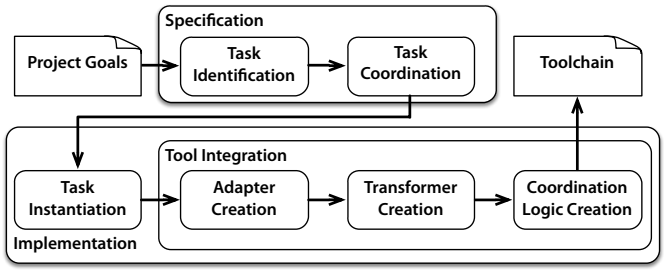

Fig. 1. Steps necessary to build toolchains.

II. Requirements

This section elicits requirements that a support framework for software evolution toolchain creation has to meet, by breaking the toolchain building process down into steps, and by analyzing them for obstacles and opportunities for improvement.

The steps considered here are depicted in Figure 1, and will be explained in detail in the following sections. The process can conceptually be divided into *specification* and *implementation* phases. Regardless whether there is such a strict separation of activities in practice, each step has to be performed to some degree to yield a working toolchain in the end. The actual *tool integration* is broken down further into *adapter creation*, *transformer creation*, and *coordination logic creation*, distinguishing between adapting to common interface standards to enable invocation, transforming data to tool-specific formats, and coordinate tools towards executing in the desired order, respectively.

Two requirements have already been mentioned in the introduction, and can be phrased for the process as a whole: One, the framework must provide means for proper separation of concerns, so different stakeholders can be supported properly **(R1)**. And two, because every software evolution project is different, and has individual objectives and demands, a universally useful framework must support the whole field of software evolution **(R2)**. The remainder of this section is outlined following the six prime steps of the toolchain-building process, describing and analyzing them for support framework requirements; all requirements are summarized thereafter in Section II-G.

A. Task Identification

This step identifies the necessary tasks on the way towards the defined goal. A *task* refers to an activity of the process that needs to be performed to achieve the overall goal. For example, creating side-by-side comparison charts for quality metrics of COBOL and migrated Java systems, metrics have to be calculated on COBOL (first task) and Java systems (second task), and the results have to be presented as quality reports (third task).

Tasks can often be performed using standard software evolution techniques, for which tools already exist. Such standard techniques have to be inspected for their appropriateness given project-specific needs, and alternatives have to be gathered and compared. However, the information relevant for supporting the decision for or against certain techniques and tools is not organized in a way which is easily queried for particular properties. The support framework should therefore aid software evolution practitioners in finding existing techniques relevant to a given task **(R3)**, and provide them with a means to describe required properties of tasks in a standardized way for querying **(R4)**.

B. Task Coordination

Coordinating tasks appropriately has to be possible without having to deal directly with implementation details of corresponding tools, to separate concerns and facilitate automation. Task coordination can be broken down into two sub-steps:

First, the tasks consume and produce data. To achieve an overall goal, tasks have to build on each other's results. Therefore, a way to specify data flow is required **(R5)**.

Second, the tasks have to be performed in a certain order to achieve a common, meaningful goal. For simple, batch-like data-processing jobs, a data-driven chain of tasks is sufficient. More complex processes might require the definition of optional, alternative, concurrent or iterative paths. Such workflows require an additional means for control flow specification **(R6)**.

C. Task Instantiation

This step represents decisions to be made for each task whether to realize it with an existing tool, or implement a custom one. A trade-off is to be made between the effort required to implement a custom tool, and the level of control, flexibility and adaptability a generic tool may not offer. However, even with a readily available tool, the effort required to integrate it into the overall toolchain can also be prohibitive.

Analogous to the issue of finding tasks in the first place, matching them up with appropriate tools is not straight-forward, because the information necessary is not arranged for the purpose of software evolution toolchain building, so a framework should provide support for discovering appropriate tools, given the needed capabilities **(R7)**.

Tool developers also need to be supported in providing information about their tools' capabilities, to make it easier for software evolution practitioners to find appropriate tools **(R8)**.

Another prerequisite for easy integration is the tools' level of interoperability. Tool developers can provide *generic* interoperability means, whereas software evolution practitioners will focus on integrating the tools according to their specific needs, leading to tight coupling, and non-reusable integration logic. A support framework for toolchain building therefore needs to aid tool developers in making their tools directly compatible with it **(R9)**.

D. Adapter Creation

This and the following two steps aim at combining all tools into an integrated toolchain. *Adapters* are about the tools' interfaces and the way they can be addressed. For example, one tool might offer a command-line interface, another has a programming interface (API) in a certain programming language, and yet another can be accessed as a web service. To enable different tools to interoperate with each other, or be coordinated by some central controller, a translation is needed, so that the tools can speak each others languages, or are made to all speak a common "lingua franca". This metaphor of "speaking the same language" does not extend to data the tools are exchanging, however, which is what transformers are concerned with. To avoid having to create custom adapters, all tools have to agree on a single interface standard **(R10)**.

E. Transformer Creation

This step complements adapters with *transformers* to take care of differing ways in handling data. There are different levels

of such a transformation, from just changing the representation or format of the data (e.g. data marked up as XML vs. JSON), to transforming the way the data is modeled (e.g. between tool- or vendor-specific schemas for modeling an abstract syntax tree of a Java program). Translating the data into something entirely different (e.g. deriving a control-flow graph from an abstract syntax tree) can be considered a task or tool in itself.

The creation of transformers is kept conceptually distinct from the creation of adapters in the previous step: Adapters are about imposing uniform interfaces so tools can be addressed and *controlled* in a consistent, generic way. Transformers are about the *data* that gets consumed, produced, and exchanged between tools. Existing tools often expect their data in specific, non-standardized formats, so to interoperate, data has to be transformed to be exchanged.

The minimum requirement for a proper support framework for toolchain building is to make transformers first-class citizens, so they can be built in a standard way, and used to fill up a library for future reuse **(R11)**.

F. Coordination Logic Creation

This step concludes the toolchain building process by "tying everything together". It aims at the creation of an application that encompasses all the tools, as well as the required adapters and data transformers, implements the toolchain's use cases, and coordinates the tools accordingly, i.e. invokes them in the right order, and passes data between them as specified. This allows toolchains to be used as seamless, single entities.

When toolchains need to change because of evolving project parameters, parts of the coordination logic usually have to be adapted, or discarded and rewritten from scratch. This limits the agility of software evolution projects. Therefore, practitioners need to be able to evolve their toolchains at specification level, only, to avoid being bogged down by interoperability issues. For this, the actual, technical integration of tools, and the execution of toolchains according to specifications must be completely taken care of by the support framework **(R12)**.

G. Summary

The first and last requirements are arguably special: The first is a *sine qua non* for most of the following requirements. The latter is the opposite, being enabled by the conditions set by the previous requirements, and providing the most marked payoff by demanding automation, to reduce *effort* and increase *agility*.

The requirements are used in the following to assess related work, and delimit it from SENSEI, as well as to derive the approach's concepts and design decisions. While no claims regarding the completeness of this list can be made, the structured, step-by-step consideration of the whole process, gives a certain measure of coverage.

III. Related Work

Standard exchange file formats such as GXL [10] provide basic support for creating interoperable tools (R9), but only with respect to data exchange, not regarding control flow. Due to its generic nature, GXL can be assumed to cover the whole field of software evolution (R2), but GXL can only enforce a common syntax on data. Each tool still has to be taught the semantics used by other tools (R11).

Workbenches include, e.g. Bauhaus [14] and GUPRO [15]. These systems come with a fixed set of tasks they support, mostly limited to software analysis and reverse engineering, with varying visualization capabilities. Software transformation, restructuring, or refactoring are not supported, so that they cannot cover complete software reengineering, migration, or modernization projects (R2). These workbenches are not really aimed at interoperability. They can be extended, but are built under the assumption that all tools included adhere to the infrastructural standards set by it. Additional tools would have to be specifically designed for a particular workbench.

Scientific workflow frameworks provide means to define workflows out of several components. These frameworks are aimed at processing and analyzing data (unstructured data, or simple tuple-based data series), and not tailored towards software evolution. Examples include Apache Taverna [16] and TraceLab [17].

Closer to this paper's approach are *SOFAS* [7] in the field of software evolution, and *TIL* [18], [19], which is aimed at toolchain building support for regular software development, especially in the domain of embedded systems. Both achieve abstraction from technical details, and separation of concerns (R1) with service-oriented principles.

SOFAS (*Software Analysis as a Service*) supports arranging services in workflows for software analysis. It does not extend to the whole field of software evolution (R2), as the framework explicitly leverages the uniformity of analysis tools, by making corresponding assumptions for simplification. In its area, SOFAS provides ontology-based means of data integration, which is explicitly excluded from this paper's approach. The service term is used to refer to RESTful implementations; as such, there is no strong separation between implementation-agnostic service specifications and implementing components.

TIL (*Tool Integration Language*) is service-oriented and uses model-driven techniques for automatic generation of toolchain integration code. It depends on standards set by OSLC (Open Services for Lifecycle Collaboration) [20], which do not extend to the field of software evolution (R2). Discovery and description of relevant services by practitioners (R3, R4) is not covered.

In summary, approaches for toolchain-building support in and outside the software evolution domain exist, and are partly built around ideas which are also part of SENSEI. Using services and their orchestration as high-level descriptions, and model-driven techniques to derive toolchains has therefore been proven viable for tool interoperability. However, these approaches have been tailored for different application domains, are less generic than SENSEI, or are limited to subfields of software evolution. To the best knowledge of the author, the clear separation of services and components, a key to increased flexibility and reuse, and the ability to model required capabilities and have them automatically matched to providing components, is unique to the SENSEI approach.

IV. The Sensei Approach

At the core of SENSEI lies the idea of *software evolution services*: a standardized body of well-defined tasks and techniques, serving as reference for tool development, and enabling the creation of an integration framework, that allows processes to be defined based on the orchestration of services. A clear distinction is made between services and components (cmp. e.g. [21]): The service term is used to refer to abstract

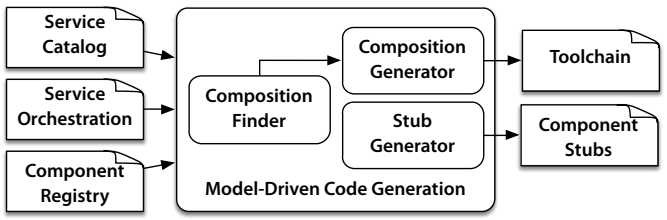

Fig. 2. Artifacts and automation tools in SENSEI's toolchain generation process.

descriptions of functionalities. Components are viewed as concrete implementations of provided services.

A high-level overview of SENSEI is provided by Figure 2. On the left, it shows the central **artifacts** the approach distinguishes: *service catalogs*, *service orchestrations*, and *component registries*.

This distinction facilitates separation of concerns (R1) of different stakeholders: **Service catalogs** are concerned with standardization, curated by a catalog maintainer. **Service orchestrations** are used by software evolution practitioners to model processes to be tool-supported. **Component registries** provide means for developers to describe their tools' provided capabilities.

To leverage model-driven techniques, all artifacts have to be represented by models conforming to meta-models. A centerpiece of SENSEI is its integrated meta-model, whose main concepts are illustrated in simplified form in Figure 3, consisting of three main layers, each corresponding to one of SENSEI's central artifacts. Integrating these three layers are **capabilities**. SENSEI formalizes capabilities as a central concept to control the granularity level of service descriptions, and enable automatic mapping of orchestrated services with *required* capabilities (*service instances*), to components with matching *provided* capabilities.

The artifacts are used as input for a set of **model-driven code generation** tools (cmp. Figure 2): a) A **stub generator**, to assist tool developers in creating or adapting SENSEI-conforming interfaces for their tools (R9), b) a **composition finder**, to match services to implementing components providing required capabilities (R7), inserting data transformers as needed (R11) and c) a **composition generator**, to automatically produce an integrated, executable toolchain (R12). In the following, the main artifacts are explained in more detail, referring to covered requirements, and their meta-model-based definition as depicted in Figure 3. SENSEI's automation tools are described in Section V.

For illustration, parts of a toolchain built in Q-MIG is used as example. Its goal is **quality measurement**: given a set of metrics, and a COBOL or Java software system, values for all given metrics are to be produced, associated with the sub-system elements (e.g. packages, files, classes, and methods in the case of Java) they were calculated on.

A. Service Catalog

Having a catalog of software evolution services is a fundamental prerequisite for all other concepts that make up SENSEI, and thus it indirectly supports all requirements gathered in Section II. It is mainly aimed at establishing an abstraction layer to separate toolchain design from tool integration and implementation (R1), providing a central directory to collect services of the whole field of software evolution (R2), which can be browsed for appropriate services for a given task or activity (R3), and queried for standardized properties and capabilities (R4). Data transformers are modeled as services, as well, so transformer implementations can be collected in the component registry for increased reusability (R11).

An existing, filled service catalog mainly supports *task identification* (Fig. 1). For the quality measurement example, four services are identified: first, the source code has to be *parsed* to provide an abstract syntax tree (AST), which is more appropriate for further analysis. The next step is to actually *calculate metrics*. At the same time, a simple kind of "re-architecting" is needed to *extract the basic structure*, i.e. a hierarchy of packages, files, classes, etc. (and corresponding concepts for COBOL). Then, the measured metric values can be *mapped to the structural information* to produce the desired output format.

The content of the service catalog is structured according to the corresponding meta-model viewpoint (top layer in Figure 3). A *service* has a name (e.g. "`Calculate Metric`") and a description, as well as arbitrarily many input and output parameters. `Calculate Metric` is modeled with two inputs, to accept a metric and a software system, and one output to return the calculated values. Apart from services, the catalog contains a type hierarchy of *data structures*, to which the services' *parameters* refer. Standardization of data structures is not a part of SENSEI, so the meta-model serves only to establish different types and specializations by unique names.

The software system input of `Calculate Metric` is typed as AST. Its sub-types include Java-AST and COBOL-AST. Notice that no information regarding technical representation is stored at this level – this is deferred to the component registry. Data structures could serve as an extension point for data integration approaches (e.g. based on universal meta-models [11], reference meta-models [12] or ontologies [5]) to complement SENSEI, but it can function without it, relying on reusable data transformers.

Services further possess *capability classes*, consisting of one or more *capabilities*. In the catalog, capabilities serve to control service granularity. `Calculate Metric`, for example, represents a commonly used technique in software evolution, yet as a service it is too generic to be useful in concrete projects, where specific metrics (e.g. SLOC, cyclomatic complexity, etc.) need to be calculated over systems written in a certain programming language (e.g. Java, COBOL, etc.). `Calculate Metric` has an input parameter for metrics, but that does not mean that implementations of the service can be expected to support every software metric conceivable. Instead of cluttering the catalog with a service for each possible combination of metric and programming language, capabilities enable the modeling of "degrees of divergence". `Calculate Metric` has two capability classes: *programming language* (COBOL, Java, etc.) and *supported metrics* (SLOC, McCabe, etc.)[2]. The `Parse` service only has the former.

There is one more important concept in the catalog's meta-model, which connects service capabilities with their parameters and data structures: *restrictions*. Restrictions are interpreted like logical predicates, modeling the relationship between data (sub-)types and capabilities. `Calculate Metric` has a restriction defined for its software system input parameter, connected to the capability *COBOL* (of class programming language) and the data structure *COBOL AST*,

[2]Note that this is a simplified example. In practice, additional aspects, like the supported version of a programming language, might have to be considered.

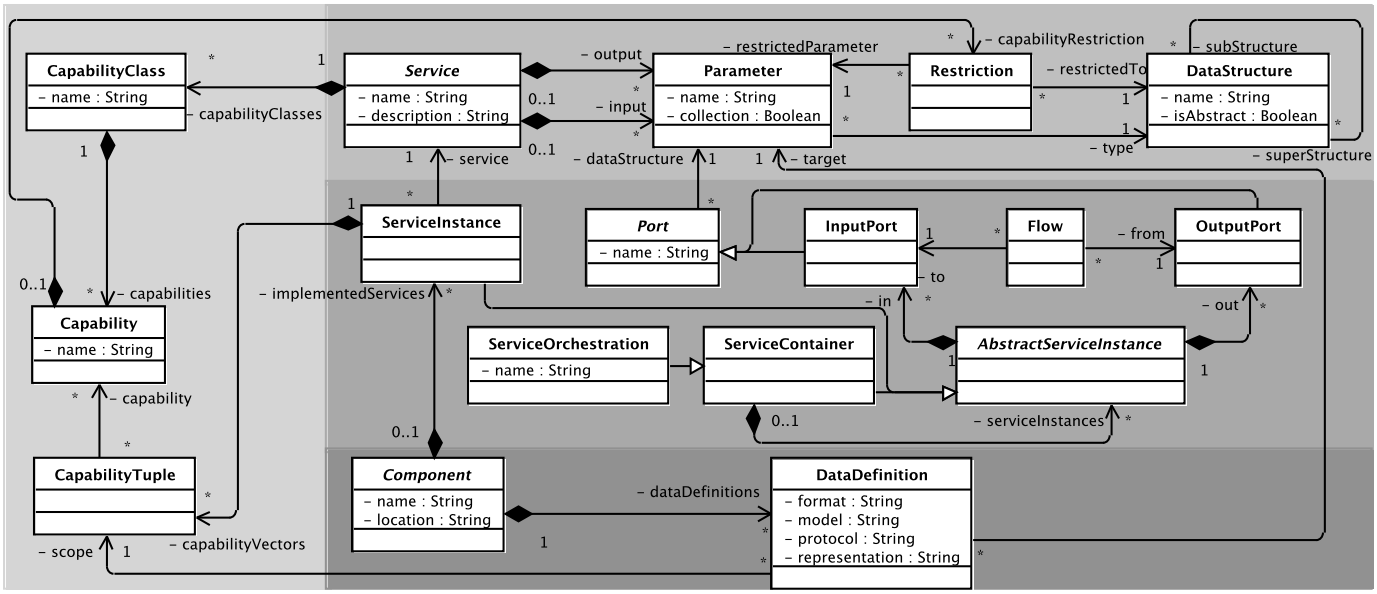

Fig. 3. Excerpt of SENSEI's integrated meta-model, depicting its layers (viewpoints) and their core concepts.

saying: "if the capability is *COBOL*, then the type of the parameter must be the (more restrictive) sub-type *COBOL-AST*." This mechanism is leveraged on the service orchestration level (Section IV-B) to substitute explicit control flow specification with a declarative approach based on required capabilities.

While there often is a direct relationship between a service's capabilities and its ports' data types, made explicit by restrictions, capabilities are used to declare properties requested of a service, and can be independent of data types. For example, a code clone detection service could have a capability class for the types of clones [22] it is able to recognize, which should not have an impact on the kind of input and output data.

To fill the service catalog, either a *top-down* or a *bottom-up* approach can be used. The former approach identifies services from relevant publications and diverse software evolution projects, to create a catalog of generic, standardized services. Services can be picked from the catalog instead of being created for the project. A top-down process for service discovery and description, based on mining publication databases and clustering techniques, is presented by Jelschen [23], along with classification schemes and a corresponding description model, details which have been omitted here (Figure 3) for clarity.

Lacking a comprehensively filled catalog, services can be created *bottom-up* instead, only for a project's required functionalities, giving full control over service design, but potentially leading to project-specific services with lower reuse value. E.g., in the Q-MIG example, the sub-system level (package, file, etc.) is assumed to be coded into metrics (which is why it does not rely directly on the structure extraction) in a project-specific manner. Still, this approach can be used to fill a catalog incrementally, and refine and generalize its services in the process.

B. Service Orchestration

The service orchestration viewpoint (middle layer in Fig. 3) caters to the concerns of software evolution practitioners (R1), i.e. finding appropriate services (R3) based on descriptions dictated by project-specific needs (R4), and designing processes in terms of services and their data (R5) and control flow (R6) interoperation. The service catalog supports part of these requirements, but is used by software evolution practitioners less directly, and in a read-only fashion.

Service orchestrations represent the *task coordination* step (Fig. 1). A graphical representation of an orchestration for the quality measurement example is visible in the center of the editor screen-shot in Figure 4.

The central concept of this part of the SENSEI meta-model (Fig. 3) is the *service instance*: whereas the catalog contains service "blueprints", its instances represent concrete usages or invocations in particular scenarios, with certain capabilities selected. The example orchestration contains a service instance (rounded boxes with an encircled "S"-symbol in Fig. 4) for each service introduced before, to which they *conform*. E.g. parameters of services dictate their instance's *ports* (small boxes on the service instance's borders). Ports can be connected by data *flows*, e.g. the `Parse` instance outputs an AST, which is used as input for the instances of both `Calculate Metric` and `Extract Structure`.

Control flow is dictated by the order in which service instances appear in an orchestration (the meta-model's corresponding association is *ordered*; in Fig. 4, this is represented by grey arrows). Special kinds of orchestrations exist to represent conditional branching, concurrency, and loops. The corresponding classes are omitted from Fig. 3 for clarity, but examples are shown in Fig. 4: `Calculate Metric` and `Extract Structure` can run concurrently. `Calculate Metric` is furthermore executed once for each metric using the *map* control structure to split collection-typed input data, run the nested orchestration once for each input element, and then map its results back into a collection.

Services instances are refined by specifying *required capabilities*. This allows software evolution practitioners to choose one capability from each of the service's capability classes to create a *capability tuple*, representing a specific functionality. To express that the `Calculate Metric` instance must be able to evaluate metrics on both Java and

COBOL code, and calculate McCabe on both languages, and SLOC on Java only, capability tuples would look like this:

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

This mechanism allows to specify what functionality is required declaratively, i.e. without having to state how it is provided, and is thus completely independent of tools and their implementation and integration. The required capabilities could be provided by a single tool, or (for example) by three, each providing the functionality represented by one of the capability tuples. In the example, there is only a single instance of `Parse`, with capability tuples (COBOL) and (Java). The Q-MIG project had individual parsers for each language, though. The selection is taken care of by SENSEI's tooling using the information provided by restrictions from the service catalog, and runtime introspection of input data to determine its type.

C. Component Registry

The component registry is a separate viewpoint (bottom layer in Fig. 3) aimed at tool developers (R1), to register their tools with SENSEI, and describe their functionality (R8). Data transformers are registered as implementations of the corresponding transformer services, as well, increasing their reusability (R11).

A component registry supports *task instantiation* (Fig. 1): it provides the necessary infrastructure and information to automate this step completely. In the quality measurement example, there are three components: parsers for COBOL and Java ("*COBOLFE*", "*JavaFE*"), and the custom-built *MetricCalculator*.

Components are associated to one or more service instances (Fig. 3), not directly to catalog services: They generally do not implement a service's whole spectrum of possible functionality, but provide a subset of it. This is declared using *provided capabilities*, reusing the mechanism for required capabilities.

In Q-MIG, *MetricCalculator* actually implemented all services of the quality measurement example, except for `Parse`, i.e. a single component can implement multiple services. For `Parse` it is the other way around, as both parser components implement it, but none to the full extent required by the service instance in the orchestration, i.e. *COBOLFE* declares only (COBOL) as capability, and *JavaFE* only (Java).

Service instances of orchestrations, and those used to describe what functionality a component provides, are separate. Finding matches between the two is done automatically by the composition finder, described in Section V.

The component registry further requires *data definitions* to be specified for each parameter of each implemented service, to map the conceptual data structures defined in the service catalog to concrete technical realizations. For example, *JavaFE* outputs an AST in a proprietary XML format. *MetricCalculator* uses the same model, but requires a TGraph [24] format.

A special kind of service, "`DDTransform`", representing *data transformers*, takes care of such discrepancies. In the Q-MIG example, there are actually two more components which implement this service, each realizing a data transformation from XML-ASTs to TGraph-ASTs, one for COBOL, the other for Java (specified by capabilities). If available in the registry, SENSEI will insert them automatically on data flows.

V. APPLICATION

With the information from a service catalog, an orchestration, and a component registry as input, a set of tools for *model-driven code generation* (cf. Fig. 2) automate the actual tool integration work. The *composition finder* matches orchestrated service instances to implementing components, which provide the required capabilities (R7), and automatically inserts data transformers into the data flow where needed (R11). The *stub generator* supports tool developers in providing SENSEI-compatible tool interfaces (R9), and enforces conformance to the target framework's component model to provide uniform interfaces (R10). The *composition generator* transforms the input models into code, referencing the components found by the composition finder, and invoking them in a manner and order conforming to the orchestration. The generated code is embedded in an appropriate component framework, and can be compiled into executable toolchains (R12).

To show feasibility and applicability, the concepts of SENSEI have been implemented, and the resulting, prototypical toolchain-building support framework has been used to recreate parts of the Q-MIG toolchain. This section first describes, how the SENSEI meta-model has been realized technically, and presents the editors which have been build for easier filling of the service catalog, registering components, and graphically modeling of orchestrations. Then, *SCAffolder* is introduced, a prototype implementation of the model-driven code generation tools of SENSEI, targeting the *Service Component Architecture* (SCA) as component framework.

A. SENSEI Models and Editors

To represent the SENSEI meta-model and its instances, JGraLab's TGraphs [24] are used, providing meta-modeling facilities, query and transformation languages (GReQL and GReTL, respectively), and tooling. This includes a bridge to the alternative technological space spanned by the ECore-based Eclipse Modeling Framework (EMF) [25]. The TGraph infrastructure is essential for the code generator *SCAffolder* (Section V-B).

An editor for SENSEI models has been created using the Eclipse Sirius [26] framework, based on a previous prototype [27]. A screenshot of the resulting editor is shown in Figure 4. The editor manages models as instances of the meta-model as a whole. The edited service catalog, orchestration, and component registry can therefore be saved to a single file, which SCAffolder accepts as input.

The meta-model's three viewpoints are kept separated in the editor. In the left pane, a tree-view of the service catalog is visible, showing data structure hierarchies and services with input and output parameters. The component registry can be viewed, browsed, and edited in a similar manner.

The orchestration view has been equipped with a graphical editor. An example of an orchestration is depicted in the center of Figure 4, corresponding to the quality measurement example introduced earlier. The editor allows to graphically model control and data flow (differentiated by color in the tool), orchestrating service instances (rounded boxes), which can be picked from a palette on the right. Required capabilities can be selected, and are listed inside the corresponding box, as visible on the *Parse* service instance on the left. Also visible are several concepts of the underlying meta-model, that have been omitted

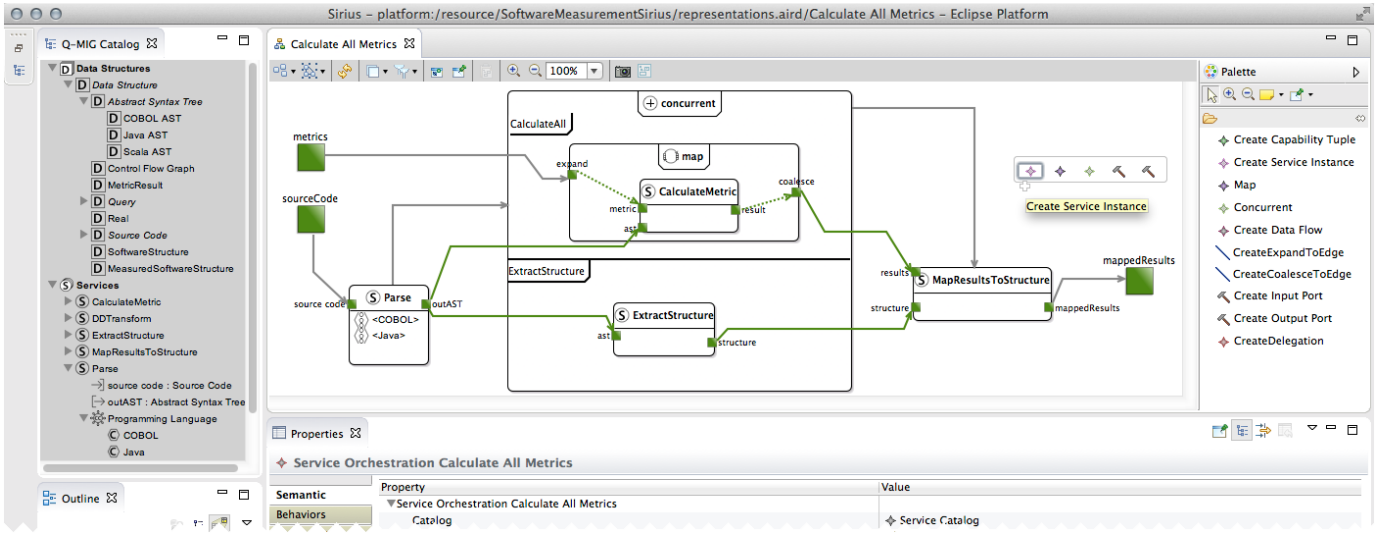

Fig. 4. Screenshot of the editor for visually designing SENSEI orchestrations.

from the simplified depiction of the meta-model in Figure 3 for clarity: E.g. control flow blocks to model concurrency, and a *map*-operator to split up array-typed input data, execute the nested orchestration once for each element, and map the results back into an output array. Here, this is used to invoke the *CalculateMetric*-service once per metric listed in the input.

B. SCAffolder

SCAffolder is a demonstrator implementing the model-driven code generation tools of SENSEI, which support the *implementation phase* of the toolchain-buiding process (cf. Fig 1). It uses the TGraph-library JGraLab, the graph transformation language GReTL, and Apache Velocity [28] templates for code generation. SCAffolder targets the Service Component Architecture (SCA) as infrastructure framework, as it provides means to integrate tools realized in a wide range of programming languages and for different platforms, requiring only lightweight wrapping for conformance to its component model. SCAffolder complements the platform-independent SENSEI meta-model with a platform-specific SCA application meta-model.

SCAffolder takes an instance of the SENSEI meta-model as input, i.e. a service catalog, an orchestration, and a component registry, and transforms it to an SCA model. First, the service instances of the orchestration each have to be paired up with one or more components from the registry, providing their required capabilities. Furthermore, data compatibility has to be ensured, or remedied with the insertion of transformers, where possible. The result is called a *composition* of components, which together can realize the service orchestration. Finding a composition for a given orchestration has initially been implemented in GReTL, yet it soon became evident that a model transformation language is an ill fit for such a task.

This early prototyping experience resulted in identifying the *Composition Finder* as a separate entity in SENSEI's tooling. Meier [27] investigated constraint-solving approaches towards this task, and produced a composition finder implementation using Prolog. Clearly establishing composition finding as a prerequisite step to the model transformation process also makes it easier to interrupt it early in case no composition can be found, e.g. if there are no appropriate components for a particular service instance, or there are data transformers missing. The composition finder will try to chain data transformers, if no single transformer providing direct conversion from one component's output format to another's input format is available.

The core of SCAffolder realizes the *composition generator*. The transformation to an SCA model is based on creating a single new component, called *composer*, which references all the components provided by the composition finder (which looked them up in the component registry). The composer is realized as a Java program. It gets filled, for example, with variable declarations to hold and pass on data, based on the orchestration's data flows, and with method invocations to call components' functionalities, corresponding to the services.

The composer assumes that the referenced components provide SCA-conforming interfaces. SCAffolder can generate the required SCA boilerplate code to base tool wrappers on (*Toolstub Generator*). No orchestration is needed for this, only a service catalog and one or more component descriptions.

SCAffolder integrates with the Maven [29] build and dependency management tool. It can be used as a Maven plugin to integrate toolchain-creation and execution into a larger build process. A Maven archetype is provided, as well, which can be used to create a new Maven project referencing a SENSEI meta-model instance. When built, SCAffolder will automatically be invoked to generate the toolchain integration code, before Maven will compile and package the project, constituting the toolchain. Currently, SCAffolder generates an application program interface, but it can easily be extended to also provide a command-line or graphical user interface on top of that. Large parts of that would be static, and could be provided as a library.

VI. CONCLUSION

This paper presented SENSEI, a framework to support toolchain-building in software evolution projects. It is based on service-oriented and component-based principles to separate toolchain specification from tool interoperability issues, and uses model-driven code generation to partly automate integration.

SENSEI shares some commonalities with related approaches like SOFAS and TIL: SOFAS is focused on software analysis, whereas SENSEI makes no constricting assumptions to be able to target the complete field of software evolution. Other differences include a focus on RESTful web services, whereas SENSEI is, on a conceptual level, completely technology-agnostic. Its prototypical implementation (*SCAffolder*) is based on SCA (Service Component Architecture), which supports REST as one of many binding technologies. As far as can be told from publications on SOFAS, it enacts (interprets) its orchestrations at runtime. SENSEI uses a code generator, instead. SOFAS goes beyond SENSEI in terms of semantic data integration using an ontology-based approach, while SENSEI intentionally uses a more primitive approach to identify data of different kinds, but without a means to describe its concepts in more detail. SOFAS still requires data transformers, but they always map to the existing ontologies, facilitating standardization. SENSEI is specifically designed to be complemented by such mechanisms.

TIL is not aimed at software evolution at all, but at integrating software development toolchains in the area of embedded systems. Like SENSEI, TIL uses a model-driven code generation approach. It also provides a custom, graphical editor for modeling data and control flow of toolchains. While SENSEI is process-centric, TIL's semantics are mapped to state machines [18], and thus are reactive and event-driven. TIL's backing meta-model is fundamentally different from the SENSEI meta-model. For example, no distinction between services and components exists, focusing instead directly on concepts on the tool level. Both implementations target SCA, however, TIL relies on OSLC standards, which do not cover software evolution.

A distinctive feature of SENSEI is its capability concept, which enables to partly specify software evolution service orchestrations *declaratively*, reducing complexity, and can provide automatic data type conversion, as well as automatic discovery of tools appropriate for a specified task.

So far, experiences with using SENSEI and SCAffolder to model and generate parts of the Q-MIG toolchains indicate that, without pre-existing service catalogs and corresponding component registries, an initial modeling overhead is created. With both artifacts becoming filled, this is expected to be offset by increased *reusability* of previously defined services and already registered components. The same can be said for the *effort* required to build the toolchain.

The effort to adapt the toolchain is greatly reduced once the infrastructure is in place. Extending an existing toolchain can be as simple as declaring an additional capability requirement: for example, to extend the quality measurement toolchain with a basic ability to also analyze SQL code, the `Parse` and `Calculate Metric` instances only have to be equipped with an additional (SQL) capability tuple. This is, off course, assuming that corresponding tools are available, and have been made available to the SENSEI framework. This ability to flexibly modify toolchains allows to react to changes during a project's run quickly, thus facilitating project *agility*.

The SENSEI-based Q-MIG toolchain is currently being extended to include additional services and tools. Furthermore, SENSEI will be used to create toolchains for analyzing and optimizing the energy efficiency of (mobile) applications [30], to further confirm its general applicability and usefulness.

REFERENCES

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

[2] ——, "Reengineering-Factory — Erfolgsmechanismen großer Reengineering-Maßnahmen," in *Softwarewartung und Reengineering*. Deutscher Universitätsverlag, 1996, pp. 19–29.

[3] S. Sim, "Next generation data interchange: tool-to-tool application program interfaces," in *WCRE 2000*. IEEE Comput. Soc, 2000.

[4] H. A. Müller, J. H. Jahnke, D. B. Smith, M.-A. Storey, S. R. Tilley, and K. Wong, "Reverse Engineering: A Roadmap," in *ICSE 2000*. ACM Press, 2000, p. 47–60.

[5] D. Jin and J. Cordy, "Ontology-based software analysis and reengineering tool integration: the OASIS service-sharing methodology," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*. IEEE, 2005.

[6] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri, "Challenges in Software Evolution," in *Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*. IEEE, 2005.

[7] G. Ghezzi and H. C. Gall, "A framework for semi-automated software evolution analysis composition," *Autom Softw Eng*, vol. 20, no. 3, pp. 463–496, Apr. 2013.

[8] A. Fuhr, A. Winter, U. Erdmenger, T. Horn, U. Kaiser, V. Riediger, and W. Teppe, "Model-Driven Software Migration," in *Migrating Legacy Applications*. IGI Global, 2013, pp. 153–184.

[9] A. I. Wasserman, "Tool integration in software engineering environments," in *Lecture Notes in Computer Science*. Springer, 1990, pp. 137–149.

[10] R. C. Holt, A. Schürr, S. E. Sim, and A. Winter, "GXL: A graph-based standard exchange format for reengineering," *Science of Computer Programming*, vol. 60, no. 2, pp. 149–170, Apr. 2006.

[11] S. Demeyer, S. Ducasse, and S. Tichelaar, "Why Unified Is not Universal," in *Lecture Notes in Computer Science*. Springer, 1999, pp. 630–644.

[12] A. Winter and J. Ebert, "Using Metamodels in Service Interoperability," in *13th IEEE International Workshop on Software Technology and Engineering Practice (STEP'05)*. IEEE, 2005.

[13] J. Meier, D. Kuryazov, J. Jelschen, and A. Winter, "A Quality Control Center for Software Migration," *Softwaretechnik-Trends*, vol. 35, no. 2, pp. 19–20, May 2015.

[14] A. Raza, G. Vogel, and E. Plödereder, "Bauhaus – A Tool Suite for Program Analysis and Reverse Engineering," in *Reliable Software Technologies – Ada-Europe 2006*. Springer, 2006, pp. 71–82.

[15] J. Ebert, B. Kullbach, V. Riediger, and A. Winter, "GUPRO - Generic Understanding of Programs An Overview," *Electronic Notes in Theoretical Computer Science*, vol. 72, no. 2, pp. 47–56, Nov. 2002.

[16] D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M. R. Pocock, P. Li, and T. Oinn, "Taverna: a tool for building and running workflows of services," *Nucleic Acids Research*, vol. 34, pp. W729–W732, Jul. 2006.

[17] (2015) Tracelab. [Online]. Available: http://www.coest.org/index.php/tracelab

[18] M. Biehl, "Semantic Anchoring of TIL," KTH Royal Institute of Technology, Tech. Rep., Oct. 2012. [Online]. Available: http://www.matt-biehl.de/research/publications/semantics.pdf

[19] ——, "A Modeling Language for the Description and Development of Tool Chains for Embedded Systems," Ph.D. dissertation, KTH, 2013.

[20] (2015) Open Services for Lifecycle Collaboration. http://open-services.net/.

[21] M. P. Papazoglou, V. Andrikopoulos, and S. Benbernou, "Managing Evolving Services," *IEEE Softw.*, vol. 28, no. 3, pp. 49–55, May 2011.

[22] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, May 2009.

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

[24] J. Ebert, "Metamodels Taken Seriously: The TGraph Approach," in *CSMR 2008*. IEEE, Apr. 2008.

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

[26] V. Viyovic, M. Maksimovic, and B. Perisic, "Sirius: A rapid development of DSM graphical editor," in *IEEE 18th International Conference on Intelligent Engineering Systems INES 2014*. IEEE, Jul. 2014.

[27] A. Meier, "Ein Composition-Finder für Service-Orchestrierungen," Bachelor's thesis, University of Oldenburg, 2014.

[28] (2015) Apache Velocity. [Online]. Available: http://velocity.apache.org

[29] (2015) Apache Maven. [Online]. Available: https://maven.apache.org/

[30] J. Jelschen, M. Gottschalk, M. Josefiok, C. Pitu, and A. Winter, "Towards Applying Reengineering Services to Energy-Efficient Applications," in *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE, Mar. 2012.