SPViz: A DSL-Driven Approach for Software Project Visualization Tooling

Author name(s) omitted

***Abstract*—For most service architectures, such as OSGi and Spring, architecture-specific tools allow software developers and architects to visualize otherwise obscure configurations hidden in the project files. Such visualization tools are often used for documentation purposes and help to better understand programs than with source code alone. However, such tools often do not address project-specific peculiarities or do not exist at all for less common architectures, requiring developers to use different visualization and analysis tools within the same architecture. Furthermore, many generic modeling tools and architecture visualization tools require their users to create and maintain models manually.**

We here propose a DSL-driven approach that allows software architects to define and adapt their own project visualization tool. The approach, which we refer to as Software Project Visualization (SPViz), uses two DSLs, one to describe architectural elements and their relationships, and one to describe how these should be visualized. We demonstrate how SPViz can then automatically synthesize a customized, project-specific visualization tool that can adapt to changes in the underlying project automatically. Furthermore, we discuss and analyze different tools that follow this concept.

I. Introduction

This is joint work with an industrial partner who has to maintain large software projects for a long duration, a setting that is quite common in industry. To maintain a good understanding of complex software architectures is a non-trivial task. Furthermore, one regularly has to make new team members familiar with the existing software. *Diagrams* can aid understanding concrete connections and ideas and the broader architecture of a system [1], [2]. However, it is still common practice to create such diagrams manually. This requires significant maintenance effort [3] and bears the risk of becoming inconsistent with the actual project.

There are multiple approaches to combat this issue. There are tools to reverse-engineer the actual architecture of projects to compare it to the modeled architecture and adapt them to each other [4], [5]. Other solutions require a direct inclusion of the architecture design into the actual source artifacts. Such architecture inclusion is usually done via extended languages [6], Architecture Description Languages (ADLs) that are developed alongside the architecture [7], or language-specific architecture systems such as OSGi [8], [9]. For example, Figure 1, a view of a tool generated by SPViz, provides a very high-level view of a specific OSGi project highlighting its architecture consisting of services, features, products, and bundle dependencies that can be browsed to provide customizable views as shown later in the paper.

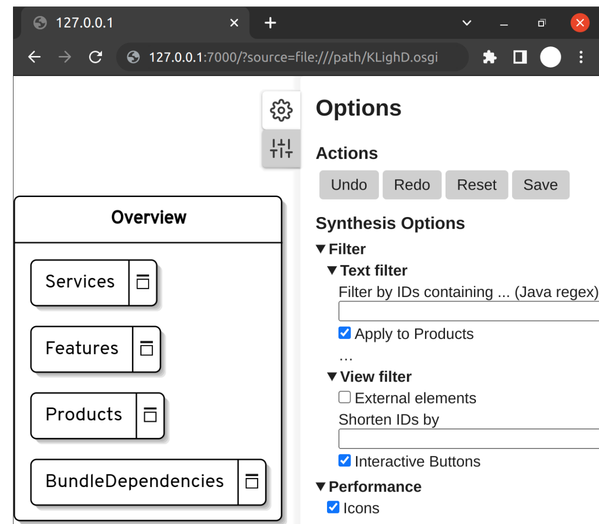

Fig. 1. Screenshot of an architecture visualization tool synthesized by SPViz, here for OSGi projects. The overviews can be interactively expanded to show connections as shown in Figures 2 and 3. The open sidebar allows to customize the view with filters and interactive features.

Irrespective of the taken approach, architecture visualization tools [9]–[12] provide insights into legacy code. However, most of them are *specific* for one task or project style, making them unusable for most other projects. As the use of programming languages and project structures surrounding them changes over time, new tools to work with those languages and structures must be developed. The problem for the developers is not only to keep up with the technology, but also with all tool support outside of rather general IDE and debugging tooling. For each new project structure, developers will ask how project artifacts relate to each other and which hierarchies exist, to explore and explain the projects. Alternatively to that project-specific approach, one may use tools that support very *generic* visual languages such as defined in the UML standard. As stated by a survey by Lange et al. [13], such languages can be used and understood by many developers, architects, and other users of code. However, such generic techniques may fail to be specific enough to describe the needs of domain experts and require too much manual effort to yield pleasing and meaningful diagrams. Another survey by Malavolta et al. [14]

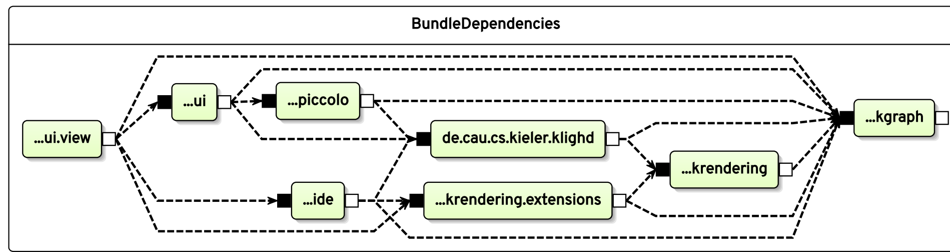

Fig. 2. View of the internal bundle dependencies originating from the *klighd.ui.view* bundle of the KLighD[2] project, synthesized by the tool generated by SPViz based on structural and visual descriptions in Figures 7 and 10.

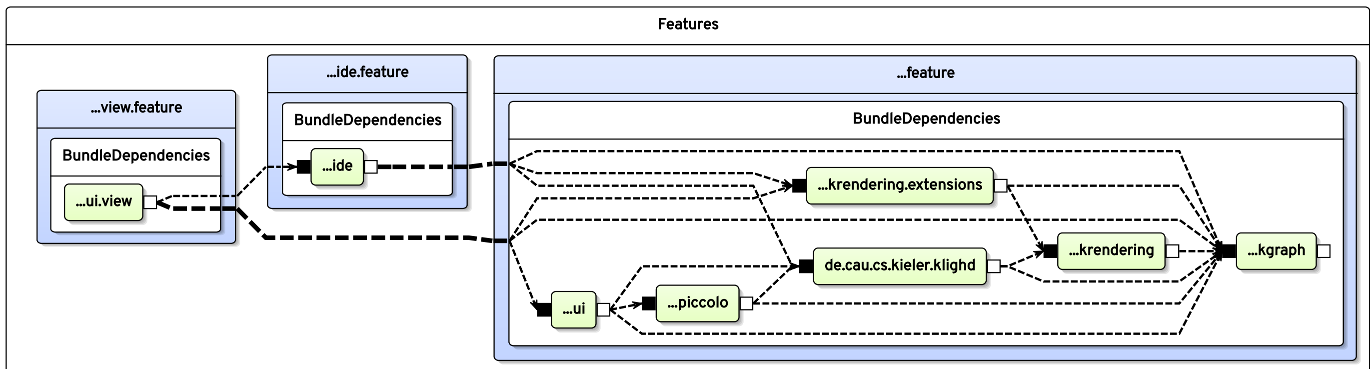

Fig. 3. View on the same bundle dependency hierarchy as in Figure 2, filtered by their categorizing features.

highlights what industry lacks in current architectural tools. They mention *support for multiple views* for architectures and adequate *support for specific architectural styles and patterns*.

In the current state of the practice, for projects where no good specific tools exist and architects do not want to use the traditional way of using manually designed UML diagrams, they need other tool support. Thus, the research question we address in this paper is: **How can one create customized architecture visualization tools with minimal effort**. In answer to that question, we here propose a Domain Specific Language (DSL)-driven approach, referred to as Software Project Visualization (SPViz). SPViz uses project meta modeling with two DSLs, one to describe architectural elements and their relationships, and one to describe how these should be visualized. Provided with such architecture and visualization descriptions, we propose to automatically synthesize a customized, project-specific visualization tool. We have validated this approach with an open-source library, also termed SPViz[3]. An outline of an initial view for further configuration of a concrete architecture visualization tool synthesized by SPViz is shown in Figure 1.

As proposed in our previous work [9] for architecture visualization specific to OSGi, the concept for that specific tool follows the *modeling pragmatics* approach [15]. Specifically, it uses the Model-View-Controller (MVC) paradigm [16] as a guiding principle. The approach separates the *views* for exploration and navigation from the *models* of the underlying projects and applies filtering and interaction possibilities as the *controller*. This previous work only works for a single architecture and therefore lacks the applicability to other architectures. SPViz automates the process of designing such tools and adapts the concepts to arbitrary architectures. A dependency hierarchy of an example OSGi project, where the visualization tool was generated by SPViz can be seen in Figures 2 and 3. Here we emphasize the importance of filtering in both the architectural description and visualization tooling to not show everything at once, but focus on individual components of a system and their context. The figures are interesting and readable, because they do *not* show many details, but filtered views, here for the investigation of the project-internal dependency hierarchy of a specific bundle artifact.

Outline: In Section II we recapitulate how the visualization and interaction style proposed in our previous work [9] works for OSGi projects. Next, we cover the **main contributions**:

- We present the SPViz approach via two DSLs describing arbitrary architectures and their visualization as a generalization of the OSGi visualization tool in Section III.
- We propose how to automatically generate a complete visualization tool, as well as template code for a model generator akin to the OSGi visualization using the DSLs based on example project architectures in Section IV.
- We illustrate the DSLs using example projects, both open source and ones from industrial partners, and present the generated tools with their feedback in Section V.

Finally, we compare related work in Section VI, discuss further directions in Section VII and conclude in Section VIII.

[2]https://github.com/kieler/klighd

[3]Link omitted, for review availability see Section IX.

II. Visualizing Specific Project Architectures

In our previous work [9], we presented a visualization tool specific to the OSGi architecture to aid developers in understanding legacy OSGi projects and to document actively developed systems. The use of the diagrams, according to the previous paper, allows to move from manually drawn diagrams such as UML to automatically created diagrams that are useful for system documentation purposes. The visualization tool utilizes the tooling of the KIELER Lightweight Diagrams (KLighD) framework [17] with automatic layout using the Eclipse Layout Kernel (ELK)[4]. KLighD provides a visualization of OSGi projects given a *model synthesis*, which is implemented in that tool. This visualization uses node-link diagrams to represent structural relationships between architectural artifacts. This follows the *graph-based* visualization technique, a term coined by Shahin et al. in a systematic literature review of software visualization [1]. This section summarizes the concepts and visualization techniques, the later sections generalize these concepts for arbitrary architectures.

The *view context model (VCM)*, depicted in Figure 5, is the central model that allows users to actively interact with the tool by modifying and filtering views to be reusable for documentation purposes in evolving (software) projects. This model is entirely hidden from the user and only modified by interaction with the UI. The *project model (PM)* contains the extracted data of a concrete project, here for an OSGi project. It is the data source of the VCM and describes the project at its state in time when the PM was generated. The PM conforms to the meta model of the OSGi architecture, therefore we also call this the *architecture meta model (A2M)*. It models all possible PMs for OSGi projects. When initially visualizing a PM, a VCM is created. Together with the model synthesis and KLighD, this allows to visually browse different views sensible in the OSGi environment.

Figure 1 shows the view of such an initial VCM as it is shown for any OSGi project. Interacting with the view via the options, filters, or the UI will change the VCM directly to reflect the currently shown and connected elements. Figure 2 is such a pre-configured view investigating the KLighD framework, which also uses OSGi as its project architecture. The view was configured to focus on the bundle dependencies view to show all bundles that are directly or indirectly required by its *klighd.ui.view* bundle. All connections of the project can be interactively added or removed to show any hierarchy.

Filtered views based on e.g. features can be browsed as shown in Figure 3. Here the features show child views for their bundle dependencies, which can be expanded to browse the bundles filtered to that feature.

By interacting with the views with the mouse, the VCM and with that the view can be modified. Overall, the user can

- (un-)focus and expand/collapse views,
- connect the connected artifacts of an artifact following a connection once, or for all artifacts at once,
- undo/redo the last action or reset the view to its default,
- filter to show or hide the individual artifact views,
- filter artifacts based on their IDs,
- export a VCM to document specific parts,
- and more standard browsing interactions.

Overall, this previous work [9] can be used for OSGi projects, but lacks usage for any other architecture. Therefore, we now generalize this visualization tool and make it applicable to arbitrary architectures.

[4]https://www.eclipse.org/elk/

III. The SPViz DSLs as Architecture and View Meta Meta Models

To allow domain experts to conceptualize a visualization for software projects following arbitrary architecture meta models (A2Ms), we define meta meta models to describe the general structure of a software architecture and an abstract way of visualizing that architecture. We illustrate the meta modeling hierarchy of visualization tools relative to our previous work [9] with our proposed abstraction in Figure 4. Section II explains the project, view, and their respective project model (PM) and view context model (VCM). We refer to the meta model for describing the architecture the *architecture meta model (A2M)*, and the meta model for the possible types of shown connections and views the *view context meta model (VC2M)*. This section introduces two DSLs for defining such A2Ms and VC2Ms, thus making the DSLs themselves an *architecture meta meta model (A3M)* and a *view context meta meta model (VC3M)*.

A. The Architecture Meta Meta Model

Domain experts can describe a project architecture using our definition of the *architecture meta meta model (A3M)* DSL. The grammar is defined using the Xtext [18] framework and shown in extended Backus–Naur form (EBNF) in Figure 6.

All project structures are different in their concrete realization in the sense of which files and which configurations define the underlying project. However, in an abstract sense projects always contain different *artifacts* and *references* between these artifacts. *Artifacts* can be coarse- or fine-grained parts of a software system such as entire products, features, classes, or even statements, which may refer to other artifacts. *References* can be further specialized into *connections*, e.g. dependencies connecting different artifacts, and *containments*, e.g. some product artifact containing a set of packages. We define these components as the A3M. The A3M can be applied to most architectures to show how all their different artifacts relate to each other. This concept is comparable to other meta models used for Model Driven Engineering (MDE) such as the Meta Object Facility (MOF) [19], in a simplified version. We do not need its relations special to object orientation, such as class inheritance to describe arbitrary software architectures. We have found that the A3M suffices to describe interrelations. In fact, class inheritance can be modeled with our model through the use of references from a class artifact to itself. This is due to the A3M being self-describing, just as the MOF. Furthermore, the A3M restricts the designer to design their

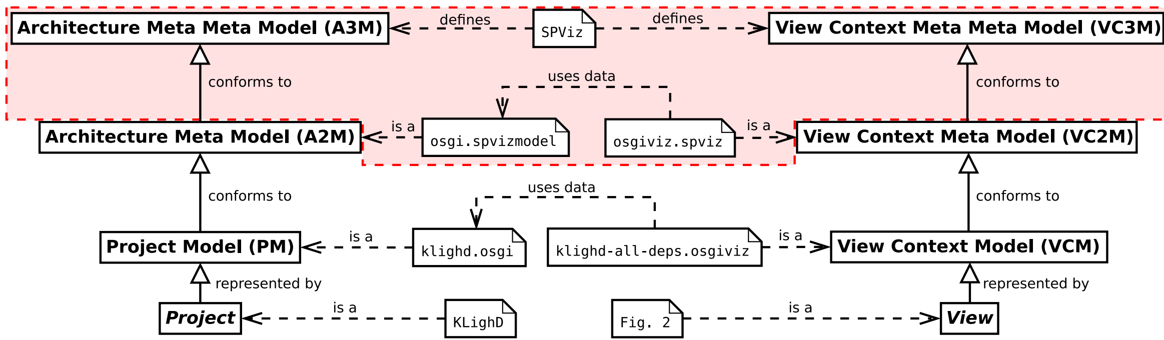

Fig. 4. The meta modeling hierarchy of SPViz. The shaded top is our proposed abstraction, the center contains examples for the different models.

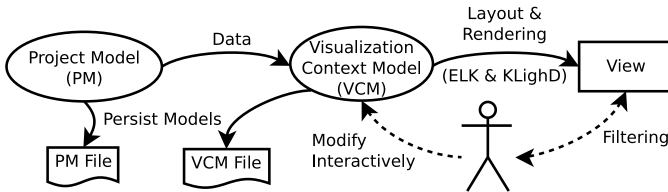

Fig. 5. The usage process of the view tools with its core, the view context model. Used for configuring and filtering views for later reuse. Solid arrows depict data flow, dashed ones the interaction paths to control the VCM. Adapted from anonymous authors [9].

```
SPVizModel = "package" Name
    "SPVizModel" Name "{"{Artifact}"}"
Artifact = Name ["{"{Reference}"}"]
Reference = Containment|Connection
Containment = "contains" Artifact
Connection = Name "connects" Artifact
Name = ? Letters, numbers, full stop ?
```

Fig. 6. A3M DSL grammar.

architecture in a minimal way and to restrict it to a unique way for later code generation.

Figure 7 shows an example use of the A3M DSL to describe the OSGi architecture A2M. This example intentionally revisits the OSGi architecture to show that the previous tool [9] can entirely be generalized. The information in the example consists of the name, artifacts, their hierarchy and connections. The resulting meta model describes coarse- and fine-granular artifacts of the OSGi architecture modeling bundle dependencies and service hierarchies. Blocks within the *SPVizModel* block define the artifacts that the architecture contains, here *features*, *bundles*, *service interfaces*, and *service components*. These artifacts are structured according to the artifacts that they contain. In this example, the features are structured by the bundles that they contain. This artifact structure can be used to automatically generate a class hierarchy, which is part of the subsequent code synthesis.

A corresponding class diagram with containments modeled as references is shown in Figure 8. The class diagram for the OSGi example would then have classes for a *OSGiProject* and all artifacts defined in the DSL. Every artifact has lists that refer to all artifacts they are in a containment or connection

```
// generate code into this package
package com.spviz.osgi

// name of the project structure is OSGi
SPVizModel OSGi {
  // the artifacts the project contains
  Feature {
    // features give structure to their bundles
    contains Bundle
  }
  Bundle {
    // bundles may connect to other bundles
    // as a connection called "Dependency"
    Dependency connects Bundle
    // services are defined within bundles
    contains ServiceInterface
    contains ServiceComponent
  }
  // service components can require and provide
  // other service interfaces. Inverting the
  // provision for a consistent "requires" direction
  ServiceInterface {
    ProvidedBy connects ServiceComponent
  }
  ServiceComponent {
    Required connects ServiceInterface
  }
}
```

Fig. 7. Example A3M DSL usage.

relation to. I.e. each artifact refers to the source or target artifact for each connection, as well as the parent or child artifact for each containment it is used in.

PM instances of this OSGi A2M describe information of the structure of concrete projects to model dependencies between bundles from the project itself and external ones. One can also get an insight into otherwise more obscure connections of different service components, as defined in the service layer of the OSGi specification [8] as *service objects*.

This example OSGi A2M taken from the A3M DSL is now comparable to the manually described one from Section II. However, only using the A3M is insufficient to define how different views should be configured and filtered. For that, we define the *VC3M* as another DSL.

B. The View Context Meta Meta Model

The *view context meta meta model (VC3M)* allows to describe which of the artifacts and their connections from

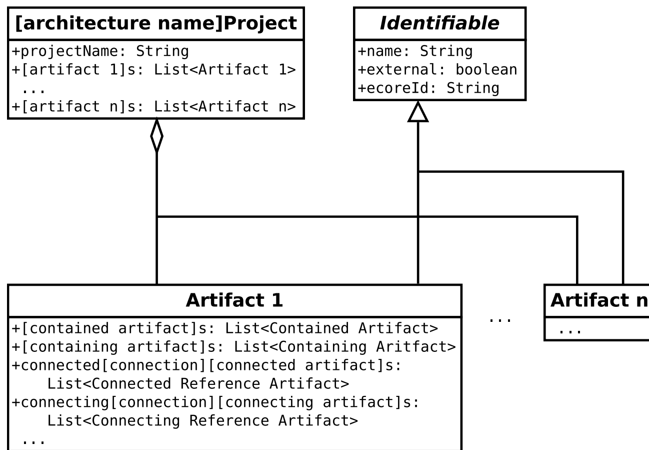

Fig. 8. Abstract class structure of the generated code for the A2M.

```
SPViz = "package" Name
    "import" URI
    "SPViz" Name "{"
        {View}
        {ArtifactShows}
    "}"
View = Name "{"
    {ShownElement}
    {ShownConnection}
    {ShownCategoryConnection}
"}"
ShownElement = "show" Artifact
ShownConnection = "connect" Connection
ShownCategoryConnection = "connect" Connection
    "via" Artifact {">"Artifact} "in" View
ArtifactShows = Artifact "shows" "{"
    {ArtifactView}
"}"
ArtifactView = View "with" "{"
    {Artifact "from" Artifact {">"Artifact}}
"}"
```

Fig. 9. VC3M DSL grammar, referring to the A3M DSL grammar in Figure 6.

the A3M should be visualized. Typically, not all possible connections should be shown in any view, and not all artifacts of the same type should be in the same view part. Just showing everything at once is typically not the best visualization for project structures, but filtered subsets are.

Therefore the VC3M allows the architects to further structure their visualization. Its grammar defined by an Xtext DSL is shown in Figure 9. The VC3M, connecting to an existing A3M, describes *views*. These views refer to the A3M and define which artifacts and which of their connections can be visualized within them. This allows for configurability, to have views specifically intended to show one kind of connection between artifacts, or multiple connection types for broader overviews. Furthermore, the VC3M allows to configure how the artifacts themselves should be visualized. As artifacts from the A3M may contain other artifacts, we allow the definition of *artifact views* as views filtered to the context of this parent artifact. Artifact views can be used especially for artifacts that have the purpose of organizing other artifacts, such as bundle categories as subsets for bundles in the OSGi specification [8].

```
// generate code into this package
package com.spviz.osgiviz
// refer to the "OSGi" model above
import "osgi.spvizmodel"

// this visualization is called "OSGiViz"
SPViz OSGiViz {
  // the available views for OSGiViz
  // this is the view for services
  Services {
    // show the service hierarchy with all
    // its artifacts and connections
    show    OSGi.ServiceInterface
    show    OSGi.ServiceComponent
    connect OSGi.ServiceInterface.ProvidedBy
    connect OSGi.ServiceComponent.Required
  }
  // view for bundles and their dependencies
  BundleDependencies {
    show    OSGi.Bundle
    connect OSGi.Bundle.Dependency
  }
  // view of features, for filtering
  Features {
    show OSGi.Feature
    //connect features via their bundle dependencies
    connect OSGi.Bundle.Dependency via OSGi.Feature
        in BundleDependencies
  }

  // features can show filtered artifact views,
  OSGi.Feature shows {
    // inner views connected as defined above
    BundleDependencies with {
      // only bundles contained in the feature
      OSGi.Bundle from OSGi.Feature>OSGi.Bundle
    }
  }
}
```

Fig. 10. Example VC3M DSL usage, referring to the example in Figure 7.

The artifact views refer to the views defined above with their artifact and connection types and filter the artifacts to the context of this artifact view's parent artifact. Finally, the *category connections* are another way of organizing and filtering views. They are part of views and show the implicit connections between categorizing artifacts that are in a relation to each other via some contained artifact type. Category connections are defined by the connection they represent, the artifact hierarchy following the parent-child hierarchy to get from the connected artifacts to the defined connection, as well as the inner view to connect to. The following example will further illustrate this concept.

Continuing the OSGi example, Figure 10 shows a possible use of the VC3M DSL to describe a VC2M for the OSGi A2M. The configuration matches and extends the OSGi visualization described in Section II, thus showing that the previously manually designed OSGi visualization can be generalized using our DSLs. The example defines a new visualization for the OSGi architecture called *OSGiViz* and defines what views can be shown in general, as well as how artifacts can reuse these views to filter views down into a sensible context. The views called *services* and *bundle dependencies* clarify that the artifacts and the connections related to them from the

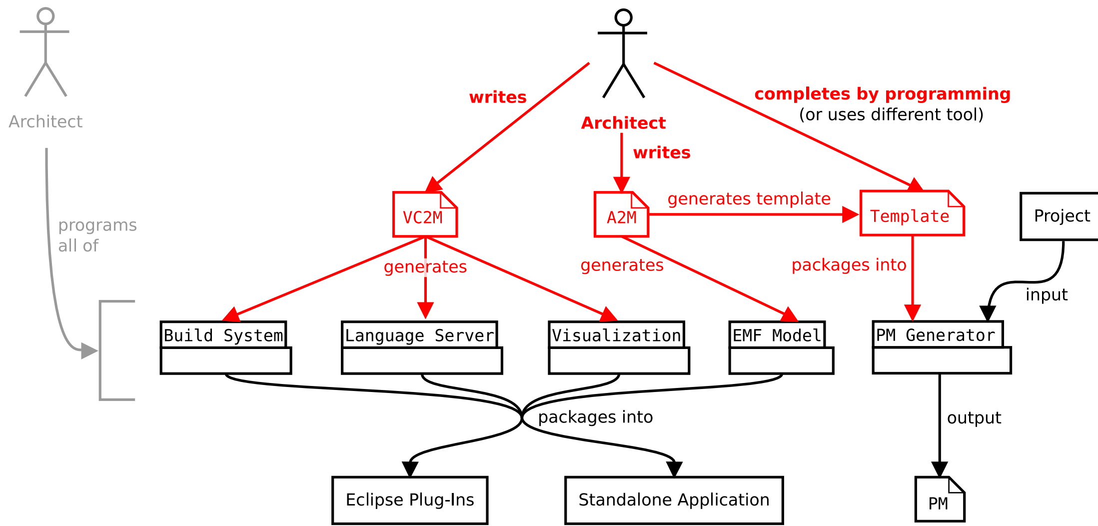

Fig. 11. Traditional (left, gray) and proposed (red) process for developing a new software project visualization tool. A2M and VC2M are written in the DSLs proposed here. Boxes represent files, software packages, and applications in UML style.

underlying OSGi model should be shown. An example view of the bundle dependencies that this allows to show can be seen in Figure 2. The view named *features* shows an overview of all possible features and a category connection.

Through configuration of an *artifact view*, the features displayed in their overview show filtered views specific to the individual features defined within an OSGi project. This artifact view defines the filters for a bundle dependencies view. It defines how the bundles that should be shown in a bundle dependencies view as defined above are filtered. The DSL allows to refer to the requested artifacts by chaining the artifacts together via their *contains* relation of the A3M. In this example it means that all bundles are shown that are listed in the feature's child bundles. This way, whereas a general view for bundle dependencies would show all bundles as they are used and defined in the whole project, the feature-specific artifact view for bundle dependencies shows the filtered view, only with bundles relevant for the feature.

The *category connection* defined in the features view allows to show the relation between features, although features do not have any direct connections as of the A2M. Because features contain bundles, which themselves define the bundle dependencies connection, this category connection allows to show the relation between the features in terms of their bundle dependencies. That is, a connection between features will be shown, if a bundle contained in one feature has a connection to a bundle contained in another feature. Figure 3 illustrates this. Here the *view* and *ide* features have a shown dependency, because the *ui.view* bundle within the *view* feature has a dependency to the *ide* bundle within the *ide* feature, as also seen in Figure 2. Edges are drawn thicker if they bundle multiple dependencies into a single edge. This is an important filtering view that does not require more effort in the later PM extraction, but just a single line in the VC2M.

IV. Project Visualization Tool Synthesis with SPViz

Figure 11 shows the proposed development process for software architects who want to apply the visualization technique from our previous work [9] to their project. They need to describe the architecture and its visualization as described in Section III and extract information about the real artifacts of the project on their file system into a PM. To design the models, questions such as 'What connections and hierarchies in the code should be made visible?' and 'What kinds of artifacts define and order these?' need to be answered. This first layer of modeling a visualization and the architecture is the main task of an architect when using SPViz. Traditionally, the architect does not have access to this first layer and needs to develop all code in the second layer manually. This traditional way was done in our previous work [9], showing the simplification the DSLs provide compared to that process.

A. Visualization Framework Generated from the DSLs

Our implementation of the SPViz approach separates the generated visualization into multiple parts and uses IDE support and the code generation features of Xtext for DSLs for that purpose. This code generation is the second layer of steps in Figure 11 and will be automatically executed when developing the DSLs in Eclipse. Once the user is finished designing their A2M, SPViz will create an Eclipse Modeling Framework (EMF) model of that A2M following the class structure in Figure 8.

The second part created by SPViz is a template for a PM generator. The template is a complete program with a dependency on the generated EMF model of the A2M. It comes with a Maven build, which can bundle it into an executable. The template contains a file *ReadProjectFiles.java* which is missing the architecture-specific extraction of data from the project's sources. It provides methods to create and connect

all artifacts as defined in the A2M as well as a checklist of all artifacts, connections, and containments that need to be extracted in the generator. Examples described in Section V implement this template to show its feasibility. Alternatively, an extractor can be a separate program or tool. Our API requires an EMF model instance following the class structure of Figure 8 in the open XML Metadata Interchange (XMI) format proposed by OMG [20], which can be implemented by any tool. For more information about the XMI model API, see the EMF book [21]. This open format allows other code mining and reverse engineering tools to work together with our A2M and therefore with the visualization tool.

Once the user is finished designing their possible views in the VC2M, SPViz will create four more modules: the EMF model and code to support that VC2M that yields a visualization using the KLighD framework [17], a language server to enable viewing KLighD diagrams in web environments, as well as a Maven build system. This can be packaged together either as Eclipse plug-ins or as a standalone application to be used together with the KLighD CLI [22][5].

As different concept, one could think of visualizing systems just using the DSLs without code generation, thus having a single generic SPViz tool that could handle all visualizations, given the DSLs. However, that would complicate writing a model generator, as it could not programmatically depend on the A2M model code, but would require more complex meta-parsing of models and it would ignore the benefits of the MDE workflow with the EMF models. Furthermore, the tool itself would need the DSLs to run and parse the configurations, making the tool less concise for its use case. However, such a generic tool would remove the need to generate, build, and execute the built tool first from the DSLs and allow the direct execution of such a tool, thus creating a tradeoff. We opted for the generation approach, mainly to make the subsequent implementation/adaption of the model generator simpler.

B. View Your Visualization Anywhere

SPViz is designed to work with any project architecture, causing the range of such an architecture's main development IDE to be pretty much any IDE. The tool support for development in any language should try to follow the IDE, as for example requested by Charters et al. [23] for visualization tools or presented in tools such as VisUML [11]. As there is no widely-adopted framework or API to add diagrams in many IDEs yet, such an integrated support is not feasible. However, the use of web tools and tools developed with web technology in mind is growing and allows for an easy access. Therefore, we also make use of web technologies for the configuration and deployment of architectural views.

While individual deployment of tools such as the individual tools generated by and developed for SPViz can vary between projects, we allow for the configuration of a largely automated process after its initial setup to view the models in the web or some IDEs directly.

The initial setup of the individual visualization tool is supported mainly in the Eclipse IDE. This includes the development of both DSLs and the implementation/adaption of a model generator for the PM. From there, most other use cases can be automated and integrated into existing web documentation. We think that after the initial setup, the tool is best used by first integrating the model generator into the CI/CD chain of the project. Architects can use the generated PM as well as the visualization tool to configure some specific views and to put their corresponding VCMs into the documentation. Here the architects are free to configure the views anywhere the final visualizations can be displayed as well. That is, the diagrams can currently be shown in the Eclipse IDE, in VS Code, or embedded in web sites (see Figure 1) by using KLighD or the KLighD CLI. The view configuration of that PM can be stored as its VCM and deployed to the documentation, making it possible to update the PM generated by the CI/CD to always have most up-to-date documentation without the need to adapt the VCM.

As another use case, the visualization applications and PM generator designed for the architecture can also be distributed to other developers of the project for exploration outside of the pre-configured documentary views.

V. Preliminary Evaluation and Validation

We evaluate our proposed concept in two ways. First, we show its flexibility and usability for diverse project architectures by realizing four different A2Ms and VC2Ms via the DSLs, motivated by open source projects and projects developed by our industrial partner. For each resulting tool, we evaluate the tool usability with these open source and industrial projects and do a quantitative analysis on the effort reduction using SPViz for the OSGi example. Second, we asked two users of different projects of our industrial partner for feedback on their goals with the generated project visualizations and their successes and criticisms.

A. Testing with Real-World Examples

We answered the design questions as mentioned in Section IV for four different project architectures and modeled the A2Ms and VC2Ms accordingly. As some examples are rather specific on the project configurations, e.g. being for a specific build and dependency system with a specific Dependency Injection (DI) framework, they do not apply to many other projects directly. However, they are easily configurable and combinable, so that tools working for other architectures with their specific use cases are built quickly. The models and generators for the examples can be found in the SPViz examples repository[6].

1) OSGi: For the OSGi visualization, the created models aim to visualize dependencies within the *module layer* and service relations within the *service layer* of the OSGi specification [8]. The example, which is slightly extended compared to the OSGi example from Figures 7 and 10, also uses *products*

[5]Available on GitHub: https://github.com/kieler/klighd-vscode

[6]Link omitted, for review availability see Section IX.

to further organize the individual components and allows to view the service connections in relation to their parent bundle artifacts as category connections. This example architecture visualization was designed as a first proof of concept. It indicates that the interactive visualizations can in fact be generalized from the implementation of our previous work [9] and result in equivalent views conveying the information that the architecture-specific visualization does. The PM generator for OSGi projects scans through a repository and parses the data of *META-INF/MANIFEST.MF* files for bundles, *feature.xml* files for features, and **.product* files for products. Furthermore, it parses *OSGI-INF/* folders and the Java files for definitions and connections of service components and interfaces and creates an OSGi PM based on that data for any OSGi project.

This example was verified with a project from our industrial partner, as well as the KLighD and Semantics frameworks of the KIELER[7] project. The partner project consists of 144 bundles plus 109 additional dependent bundles, as well as 285 service artifacts. The KLighD and Semantics frameworks consist of 25 plus 196 bundles and 166 plus 144 bundles, respectively. Example views of this have been shown in Figures 2 and 3.

To compare the effort of a manual architecture-specific implementation and the SPViz approach, we compare the manually written lines of code (LoC) using the SPViz approach for the OSGi example and the code generated from our tool. We compare to the generated code instead of the manual implementation because of slight differences and improvements between the tools and similar LoC in the generated and manual tools. The A2M and VC2M descriptions using the DSLs for this OSGi example have a combined 76 LoC and the PM generator template was extended by 466 LoC in Java to extract all OSGi-specific artifacts, their connections, and containments of any such OSGi project. The generated project consists of 318 LoC of EMF model code and 8654 LoC of Xtend and Java code that would have to be written manually without SPViz, plus the 466 LoC of above to make the generator template functional. These numbers do not include the 25 000 lines of Java code that are further generated from the generated EMF models, as they do not add to manual work in either setup. Comparing the effort on a LoC basis, this yields a reduction in the manually written LoC of $1 - \frac{76+466}{318+8654+466} = 94.3\%$. The effort is further reduced because the generated tool comes with a release engineering configuration that allows to build and package the generated tool as an Eclipse plugin or as a web tool to run in web pages in conjunction with the KLighD CLI.

2) Maven and Spring DI: A second example visualization using SPViz has the goal to visualize two kinds of connections that may occur within Java projects—dependencies between *modules* as defined by its Maven[8] build system, as well as dependencies and provisions of *service components* and *interfaces* using DI as defined in the Spring Framework[9]. These connections are structured via *Maven artifacts* bundling together multiple child modules. Remember to not confuse these Maven artifacts with the artifacts of our A3M. The modeled Maven artifacts are one kind of artifact as defined by the A3M. This modeled structure is similar to the OSGi example and shows that different systems can be modeled and therefore visualized. The main difference here is how both examples have their different artifacts persisted in the file system. For the Maven modules and artifacts, the generator parses the *pom.xml* files to build the PM of the hierarchy and the module dependencies. The DI artifacts are extracted by parsing the Java files looking for interfaces and classes with `@Inject` and `@Named` annotations.

This example was designed by our industrial partner and verified by them with a project consisting of 34 modules and 175 service artifacts. Their goal was to use such a visualization tool for automatically updating documentation and for onboarding of new developers and maintenance. Other projects can also reuse parts of this design and model extractor, as the project structure based on the Maven build and the DI information are separate and can be used with other systems. This example does not visualize all possible DI configurations of the Spring Framework, as the use case of our partner only uses the annotation-based configuration of components and requirements via `@Inject` and `@Named` annotations directly in the Java code. Therefore this example supports to extract these annotations, but no further XML-based configuration or other annotations.

3) Gradle: Many projects use build tools such as Gradle[10] to define the project structure, dependencies, and to automate the build process and tests. This build system is extensible and allows to have a look into the project structure, dependencies, etc. as a direct part of the build process, thus making it easier to reuse that data for analysis tools. Designing the DSLs and generating the visualization code for any project architecture can be done quite quickly. However, programming an extractor for any project style can require deeper insight into the project structure and its build tools and used frameworks. Reusing the data provided by frameworks and build tools such as Gradle can cut down the cost to develop such an extractor and also make it less error-prone. With Gradle, data can be extracted by the same tool that is used to execute or build the project.

This Gradle example was verified with the open source project Spring Boot[11], which contains a total of 187 so-called *projects* plus 729 further dependent projects. To make this example work together with its Gradle build system, we added a task to be executed for each sub-project during the build. This collects the information of the *Configuration API* of Gradle and presents it to the model generator in a more accessible JSON format. Similarly, other code miners and extensible project build systems can be used to extract the project information for other project types. This example shows the feasibility of using the build system to our advantage.

[7]https://github.com/kieler/semantics

[8]https://maven.apache.org/

[9]https://spring.io/projects/spring-framework

[10]https://gradle.org/

[11]https://github.com/spring-projects/spring-boot

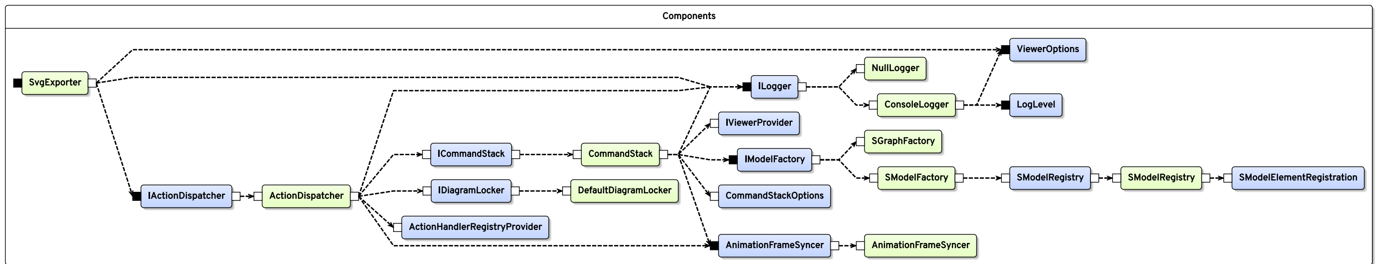

Fig. 12. The interfaces and resolved classes providing those interfaces for the *SvgExporter* class as they are pre-configured by Sprotty's main repository.

4) Yarn and InversifyJS: Finally, to show another full visualization combining dependencies and services outside of the Java programming language, we combined the dependency management of TypeScript projects using Yarn[12] with the InversifyJS[13] DI framework. The artifacts for this example are parsed by analyzing the *yarn.lock* file generated by Yarn to extract the packages and their dependencies and by parsing the TypeScript source files to search for interface-to-class bindings and `@inject` and `@injectable` annotations to create the hierarchy as described in the other examples.

We verified this with the open source framework Sprotty[14] with a dependency tree of 906 packages and 126 defined service artifacts. An example view showing the configuration of injected services as they are required by the *SvgExporter* class is shown in Figure 12. Here every other layer represents either a class (green), which was configured to provide its service to some interface, or an interface (blue) that is injected to a class. Being a quite small framework, InversifyJS has no tool to visualize such hierarchies to our knowledge, so this is a good showcase to implement novel visualizations.

B. Industry Feedback

From our industrial partner we gathered feedback on the usage of the OSGi and Maven + Spring DI examples, being applied to internal projects. We interviewed two participants, the product owner and one of the architects of the projects, which are summarized here.

One visualization goal the participants want to solve is to *explore* the modules of their architecture to get an overview, either overall or from some specific view point. Another goal is to *explain* the architecture and specific hierarchies to others by creating architectural descriptions, without the need to update such descriptions manually. Both participants stated that previously such diagrams were crafted and updated by hand. While there are many visualizations out there, this shows that at least for this questionnaire the architects were not happy with other tools they used so far. Other tools did not provide exactly what was required, because they were not usable as well, or because the architects did not find the right tool yet.

Both had the problem that views for larger projects start to require more effort to use and that clustering or pooling of artifacts into categories can induce a better hierarchical view on parts of the system. This is especially the case when there are many artifacts of the same type being visualized in the same view. The artifact views and category connections we described help to find the right context, as long as the model provides enough context via some categorizations. Further ideas indicate that diagram layouts can become a little too large for what is shown, which can be solved in future work.

Overall, their feedback indicates that the tool can and already has been used to understand parts of different system architectures. As mentioned above and further discussed in Section VII, some improvements regarding the actual views and their interaction can be added, though that does not impair the proposed approach to create visualizations for any project.

VI. Further Related Work

As elaborated in the following, many related proposals go in the direction of visualization for software architectures or using meta models to describe and visualize architectures.

Architectures of projects are often described by Architecture Description Languages (ADLs) in the literature. Medvidovic and Taylor [24] classify and describe the use of ADLs in general. Our approach is not an ADL, but a way to describe project-specific architecture descriptions to create an easier step-in into creating project-specific visualizations, or a *meta ADL*. SPViz can be used for existing software architectures, ADLs and Module Interconnection Languages (MILs) [25].

A related tool for general architecture visualization using an ADL is described in work by Buchgeher et al. and Weinreich et al. [7], [26]. While they focus on continuous co-development of the software and a formal architecture description of it, they require such a direct integration and only partly enable the analysis of legacy code that was not initially designed with their tool. Furthermore, as their tooling is restricted to the Eclipse IDE, its use for projects developed outside the Eclipse environment is limited. However, they allow constraint analysis to automatically detect architectural errors and inconsistencies and have immediate visual feedback on system change, giving them some advantages in such an integrated system. We want to bridge the gap of generic architecture ADLs and an easy way to describe meaningful visualizations for them. Our tool allows to interface with existing architectures to provide visualizations, while not requiring any concrete language or syntax on the architecture description itself. Other

[12]https://yarnpkg.com/
[13]https://github.com/inversify/InversifyJS
[14]https://github.com/eclipse/sprotty

ADLs tightly integrated into specific languages, requiring the underlying projects to conform to their constraints are for example ArchJava [6] for Java and Codoc [27] for Python. There are other similar tools for domain-specific graphical modeling tools, partly based on DSLs applicable to different architectures.

Nimeta [28] is a tool for architecture reconstruction based on views. They build graphical views based on so-called *viewpoints* for arbitrary descriptions. They clearly split the data extraction from the visualization step to allow different tools to visualize the same data, whereas we with SPViz integrate the architectural description in the view descriptions, allowing for further filtering based on the architecture.

The CINCO tool [29] generates domain-specific graphical modeling tools from abstract specifications. They use a similar meta meta modeling hierarchy to describe architectures. Their use case, however, is to generate a full modeling suite for such a model with the graphs editing to modify the underlying models. They do not aim to visualize existing architectures in a way SPViz does. Other graphical editors for models such as Epsilon [30], Sirius [31], and Spray [32] map domain-specific models to graphical views, but require explicit descriptions on how views should be displayed and model elements should be connected in the first place. The models to graphical views concept is similar to the KLighD framework that we use, while we utilize the KLighD framework to make the way of connecting elements more implicit via the VC3M DSL. These related papers style different parts of the visualizations for more configurability. The SPViz approach may also support similar individual stylings above the current box and arrow graphs as shown above, but the tool does not support such individual element styling.

Another term under which visualizing architecture is understood is the reconstruction of software architecture from the area of reverse engineering. El-Boussaidi et al. [33] use the Knowledge Discovery Meta-Model (KDM) [34] to describe legacy projects to visualize them, while other use clustering algorithms to try to infer architectural meaning from otherwise non-structured code [35]–[37]. We think approaches like these are a good way to reverse engineer and structure unstructured legacy code which can be combinable with our visualization techniques, if they can output the results to be imported in some PM generator described with our approach.

VII. Discussion

While this paper presents our proposed concept for describing arbitrary architectures and their visualization, some parts in the configurability, the visualization, and the handling of the DSLs can be improved in the future. We would like the generated visualization modules to be extensible to make the final visualizations more configurable to fit aesthetics criteria of the architecture to visualize. A first idea is to open up an API that allows the individual renderings for the overviews, the artifacts, or even the connections or the arrow heads to be configurable. This could be achieved for example via some injection mechanism similar to the configurability of many functional aspects in the Xtext framework or an extension to the DSL syntax. This could also allow the modification of automatic artifact colors. Another part that could be configurable is mapping the visualized artifacts to some web-link leading to documentation of that artifact or being integrated into different IDEs such as Eclipse and VS Code.

An improvement to the tooling to design a new visualization would be to allow the use of the DSLs outside of Eclipse in e.g. VS Code or a command line tool, and generate the required code from there. Furthermore, support to write a model generator using other languages could help developers. Another improvement would be to make the A3M compatible with other meta models such as the KDM [34].

To address threats to validity of the industry feedback, that part of the evaluation is not meant to be the final study to validate the usability of our proposed SPViz tool. The questionnaire was not structured in a controlled manner and is meant to be viewed as a initial argument towards showing the usefulness of SPViz for generating customized visualization tools. We plan on conducting a further user study investigating how users design and adapt new visualization tools using our proposed approach in the future.

VIII. Conclusion

SPViz is a new approach for software architects to quickly create a visualization tool they can use to explore any otherwise obscure architecture. It allows to create automatically updating architectural views for documentation purposes and to explain relations to developers. We built a tool following this approach to make a previously presented approach for visualizing, exploring, and documenting OSGi projects available for arbitrary software architectures, highlighting the usability of such a concept. The visualizations use state-of-the-art and well-accepted views on connections within software systems such as dependencies and service structures. We compared the tool to other meta modeling tools and architectural visualizations, such as ADLs, which usually require projects to adapt to. We do not require projects to use any specific architecture, but support the description of the architecture for any project. SPViz can be used as a visualization tool generator for legacy systems to visualize specific parts that other tools do not cover. It can also be used to quickly set up a visualization for new and emerging languages and system structures. To be applicable to projects that have no real own architecture and are just a collection of source files, a combination with other tools clustering and organizing specific artifacts is recommended.

Overall, the tool has been used and evaluated on multiple projects, showing its benefits. However, as discussed in Section VII, some areas can still be improved in future research to widen the use cases of this architecture-agnostic software visualization tool generator.

IX. Data Availability

Blinded code, data and executable programs are available at https://figshare.com/s/c5a8524421fad96c8338

References

[1] M. Shahin, P. Liang, and M. Ali Babar, "A systematic review of software architecture visualization techniques," *Journal of Systems and Software*, vol. 94, pp. 161–185, Aug. 2014.

[2] P. Eades and K. Zhang, *Software Visualisation*, ser. Software Engineering and Knowledge Engineering. Singapore: World Scientific, 1996, vol. 7.

[3] B. P. Lientz, E. B. Swanson, and G. E. Tompkins, "Characteristics of application software maintenance," *Communications of the ACM*, vol. 21, no. 6, pp. 466–471, Jun. 1978.

[4] G. C. Murphy, D. Notkin, and K. J. Sullivan, "Software reflexion models: Bridging the gap between design and implementation," *IEEE Transactions on Software Engineering*, vol. 27, no. 4, pp. 364–380, 2001.

[5] J. B. Tran, M. W. Godfrey, E. H. Lee, and R. C. Holt, "Architectural repair of open source software," in *Proceedings IWPC 2000. 8th International Workshop on Program Comprehension*. IEEE Computer Society, Jun. 2000, pp. 48–59.

[6] J. Aldrich, C. Chambers, and D. Notkin, "ArchJava: Connecting software architecture to implementation," in *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, W. Tracz, M. Young, and J. Magee, Eds. IEEE, May 2002, pp. 187–197.

[7] G. Buchgeher and R. Weinreich, "Integrated software architecture management and validation," in *Proceedings of the Third International Conference on Software Engineering Advances*. IEEE Computer Society, Oct 2008, pp. 427–436. [Online]. Available: https://doi.org/10.1109/ICSEA.2008.21

[8] *OSGi Core Release 8 Specification*, The OSGi Alliance, Oct. 2020, Accessed: Dec. 16, 2022. [Online]. Available: https://docs.osgi.org/download/r8/osgi.core-8.0.0.pdf.

[9] Anonymous Authors. Omitted per double-blind reviewing.

[10] S. Boersma and M. Lungu, "React-bratus: Visualising react component hierarchies," in *2021 Working Conference on Software Visualization (VISSOFT)*. Luxembourg: IEEE, Sep. 2021, pp. 130–134.

[11] M. Duruisseau, J.-C. Tarby, X. Le Pallec, and S. Gérard, "VisUML: a live UML visualization to help developers in their programming task," in *Human Interface and the Management of Information. Interaction, Visualization, and Analytics - 20th International Conference, HIMI 2018*, ser. Lecture Notes in Computer Science, vol. 10904. Springer, Jul. 2018, pp. 3–22.

[12] L. Georget, F. Tronel, and V. V. T. Tong, "Kayrebt: An activity diagram extraction and visualization toolset designed for the Linux codebase," in *2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT)*. IEEE, Sep. 2015, pp. 170–174.

[13] C. F. J. Lange, M. R. V. Chaudron, and J. Muskens, "In practice: UML software architecture and design description," *IEEE Software*, vol. 23, no. 2, pp. 40–46, 2006. [Online]. Available: https://doi.org/10.1109/MS.2006.50

[14] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, and A. Tang, "What industry needs from architectural languages: A survey," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 869–891, 2013.

[15] H. Fuhrmann and R. von Hanxleden, "On the pragmatics of model-based design," in *Proceedings of the 15th Monterey Workshop 2008 on the Foundations of Computer Software. Future Trends and Techniques for Development, Revised Selected Papers*, ser. LNCS, vol. 6028. Budapest, Hungary: Springer, 2010, pp. 116–140.

[16] T. Reenskaug, "Models – Views – Controllers," Dec. 1979, xerox PARC technical note.

[17] C. Schneider, M. Spönemann, and R. von Hanxleden, "Just model! – Putting automatic synthesis of node-link-diagrams into practice," in *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '13)*. San Jose, CA, USA: IEEE, Sep. 2013, pp. 75–82.

[18] S. Efftinge and M. Voelter, "oAW xText: A framework for textual DSLs," in *Eclipse Summit Europe*, Esslingen, Germany, Oct. 2006.

[19] Object Management Group, "Meta Object Facility (MOF) Core Specification, Version 2.5.1," Oct. 2019, https://www.omg.org/spec/MOF/2.5.1/PDF.

[20] ——, "XML metadata interchange (XMI) specification, version 2.5.1," Jun. 2015, https://www.omg.org/spec/XMI/2.5.1/PDF.

[21] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF – Eclipse Modeling Framework*, second edition ed., ser. Eclipse Series. Addison-Wesley, 2009.

[22] C. Fricke, "Standalone web diagrams and lightweight plugins for web-IDEs such as Visual Studio Code and Theia," Bachelor's thesis, Kiel University, Department of Computer Science, Sep. 2021, https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/fricke-bt.pdf.

[23] S. M. Charters, N. Thomas, and M. Munro, "The end of the line for software visualisation?" in *Proc. 2nd IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*. Amsterdam, The Netherlands: IEEE, Sep. 2003, pp. 110–112.

[24] N. Medvidovic and R. N. Taylor, "A classification and comparison framework for software architecture description languages," *IEEE Transactions on Software Engineering*, vol. 26, no. 1, pp. 70–93, Jan. 2000.

[25] R. Prieto-Díaz and J. M. Neighbors, "Module interconnection languages," *Journal of Systems and Software*, vol. 6, no. 4, pp. 307–334, 1986.

[26] R. Weinreich, C. Miesbauer, G. Buchgeher, and T. Kriechbaum, "Extracting and facilitating architecture in service-oriented software systems," in *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*. IEEE, Aug. 2012, pp. 81–90.

[27] C. W. Bang and M. Lungu, "Codoc: Code-driven architectural view specification framework in Python," in *2021 Working Conference on Software Visualization (VISSOFT)*. IEEE, Sep. 2021, pp. 120–124.

[28] C. Riva, "View-based software architecture reconstruction," Dissertation, Technische Universität Wien, Oct. 2004.

[29] S. Naujokat, M. Lybecait, D. Kopetzki, and B. Steffen, "Cinco: a simplicity-driven approach to full generation of domain-specific graphical modeling tools," *International Journal on Software Tools for Technology Transfer*, vol. 20, no. 3, pp. 327–354, Jun 2018.

[30] D. S. Kolovos, R. F. Paige, and F. A. Polack, "Eclipse development tools for Epsilon," in *Eclipse summit Europe, Eclipse modeling symposium*, vol. 20062, 2006, p. 200.

[31] V. Vujović, M. Maksimović, and B. Perišić, "Sirius: A rapid development of DSM graphical editor," in *IEEE 18th International Conference on Intelligent Engineering Systems INES 2014*, 2014, pp. 233–238.

[32] M. Gerhart and M. Boger, "Concepts for the model-driven generation of graphical editors in Eclipse by using the Graphiti framework," *International Journal of Computer Techniques (IJCT)*, vol. 3, no. 4, pp. 11–20, Aug. 2016.

[33] G. El Boussaidi, A. B. Belle, S. Vaucher, and H. Mili, "Reconstructing architectural views from legacy systems," in *2012 19th Working Conference on Reverse Engineering*. IEEE Computer Society, Oct. 2012, pp. 345–354.

[34] Object Management Group, "Architecture-Driven Modernization: Knowledge Discovery Meta-Model (KDM), Version 1.4," Sep. 2016, https://www.omg.org/spec/KDM/1.4/PDF.

[35] C. Stoermer, L. O'Brien, and C. Verhoef, "Moving towards quality attribute driven software architecture reconstruction," in *10th Working Conference on Reverse Engineering*, A. van Deursen, E. Stroulia, and M. D. Storey, Eds. IEEE Computer Society, 2003, pp. 46–56.

[36] C. Riva, *Architecture Reconstruction in Practice*. Boston, MA: Springer US, 2002, pp. 159–173.

[37] T. A. Wiggerts, "Using clustering algorithms in legacy systems remodularization," in *Proceedings of the Fourth Working Conference on Reverse Engineering*, I. D. Baxter, A. Quilici, and C. Verhoef, Eds. IEEE Computer Society, Oct. 1997, pp. 33–43.