Retriever: A View-Based Approach to Reverse Engineering Software Architecture Models

Yves R. Kirschner[a,*], Moritz Gstür[a], Timur Sağlam[a], Sebastian Weber[b], Anne Koziolek[a]

[a] *KASTEL – Institute of Information Security and Dependability, Karlsruhe Institute of Technology (KIT), Am Fasanengarten 5, 76131 Karlsruhe, Germany*

[b] *FZI – Research Center for Information Technology, Haid-und-Neu-Str. 10–14, 76131 Karlsruhe, Germany*

Abstract

Software systems, which have become an integral part of our daily lives, are constantly evolving and growing in complexity. The challenge of understanding and managing these systems has led to a shift towards automated extraction of models from software artifacts. However, extracting architectural models for web service or microservice systems is challenging due to the heterogeneity of formats, languages, and semantics. This challenge arises from the independent deployment and loose coupling of services, as well as the diverse technologies and platforms that comprise the heterogeneous artifacts. This paper addresses this issue with two key ideas: a knowledge representation model for diverse artifacts and a framework for integrating individual views into a unified architectural model. The former involves model-to-model transformations that consider technology-specific relationships and concepts. At the same time, the latter uses model transformation and fusion techniques to create a unified model based on an existing metamodel. This work goes beyond the current state of the art by introducing an approach for reconstructing component-based software architectures and combining model-driven reverse engineering processes to capture information across multiple views. As empirical evidence of its applicability and accuracy, we report on its successful application in real projects of different sizes. Our evaluation shows scalability in generating architectural models and can reconstruct the components of a software system and their interactions with F_1 values up to 0.94 for structural properties and up to 0.88 for behavioral properties.

Keywords: Automated Reverse Engineering, Component-Based Software Engineering (CBSE), Model-Driven Reverse Engineering (MDRE), Model Transformation, Software Architecture Model, View Model

*Corresponding author

Email addresses: yves.kirschner@kit.edu (Yves R. Kirschner), moritz.gstuer@student.kit.edu (Moritz Gstür), timur.saglam@kit.edu (Timur Sağlam), sebastian.weber@fzi.de (Sebastian Weber), anne.koziolek@kit.edu (Anne Koziolek)

1. Introduction

In today's digital age, software systems have entered every aspect of our lives, from smartphone applications to complex infrastructures that power the internet. These systems are constantly evolving, growing larger, more complex, and more mission-critical every day [1]. Understanding and managing these software systems has become a major challenge for academia and industry [2]. To understand a software system, one must first uncover its architecture, which describes its structure, its components, and their interconnections. Architecture is the key that holds the system together and influences its performance, scalability, maintainability, and security [3]. As systems become more complex, it becomes increasingly difficult to create a comprehensive and accurate representation of the architecture description. Historically, creating and maintaining architectural description models has been a largely manual and error-prone process [2], in which software architects and engineers painstakingly document system architectures, often in the form of diagrams and textual descriptions. This manual approach not only consumes valuable time and resources, but is also prone to human error and inconsistency. In response to these challenges, interest in automatically extracting models from software systems has increased recently [4]. This involves using automated tools to extract models from software artifacts such as source code, documentation, and configuration files. Automatic extraction of models from software systems has several advantages over manual modeling. It is much faster, less prone to error, and makes more fine-grained modeling possible for systems that are too complex for manual modeling [5].

The automatic extraction of architectural models from software artifacts for web service or microservice systems is challenging due to the heterogeneity of formats, languages, and semantics [6]. On the one hand, the independent deployment and loose coupling of services in such systems make it difficult to extract a consistent architectural model. On the other hand, the heterogeneous software artifacts are composed of different technologies and platforms [7], which also requires an understanding of the interaction of different components. Different software artifacts, such as source code and configuration files, can exist in different formats and programming languages. This diversity makes it difficult to develop automated tools to extract models from all types of software artifacts. In addition, the different syntax and semantics of the various formats and languages further complicate the development of a unified extraction approach. For example, the semantics of source code may differ significantly from those of configuration files. This can make it difficult to automatically infer relationships between different concerns of a software system. Finally, software artifacts often contain incomplete information [8], such as missing deployment in the source code. This requires additional information to build a complete system model. In addition, software systems are characterized by complex relationships between components and configurations that require advanced analysis techniques to be accurately captured and represented in a model. Although the current state of the art supports a wide range of programming languages, upport for heterogeneous systems consisting of multilingual source code or other types of artifacts, such as configuration files, is still being explored [9, 10].

The research questions are derived from this problem definition as formulated objectives. The basic research questions to be answered by the implementation of the approach are as follows:

RQ1 What is an effective method for reverse engineering software architecture from disparate artifacts across technologies?

RQ2 How can different perspectives of a software system be integrated into a unified model that addresses different concerns?

Based on these research questions, we present a **R**everse **E**ngineering **T**echnique with **R**efinement **I**ntegration of **E**xtracted **V**iews with **E**lements and **R**elationships (*Retriever*) to answer them. Our *Retriever* approach is based on two key ideas, which we present in this paper, that address the concerns and perspectives of different views of a software system. The first idea we present is to develop a knowledge representation model for software architecture views that can cover the different technologies, formats, and languages used in the different artifacts. Artifacts in this context can take various forms, such as Java or JavaScript source code files, YAML or JSON configuration files, or Maven or Gradle files used for build automation. The idea is to model this knowledge representation as model-to-model transformations to reconstruct views from the system artifacts, considering technology-specific relationships and concepts. This knowledge representation can cover the different types of views and their concerns for that view. The second idea we present is a framework for integrating individual views into a unified, overarching architectural model. This framework uses model transformation and composition techniques to map concepts between view-specific models and incorporate them into a unified model based on an existing metamodel. To this end, we have developed a view composition algorithm that harmonizes the different views of a software system and produces a unified software architecture model.

Based on these two ideas, we have formalized and developed our *Retriever* approach that can handle different formats and languages and translate them into a common representation for architectural modeling. Based on this formalization and development, we make the following contributions that go beyond the state of the art:

C1 A technique for transforming reverse-engineered models into more accurate representations that approximate the component-based software system architecture.

C2 A technique that combines model-driven reverse engineering processes to capture information while processing multiple views and resolving overlaps between them.

The approach has been evaluated on real projects of various sizes. Execution time increases with the size of the input code base. The approach can recover most of the components of a software system and their interrelationships, with F_1 values for structural properties up to 0.94. It can also effectively analyze the internal structure of a component, with F_1 values for behavioral properties up to 0.88. The ratio of source files to the number of components in the final architectural model indicates that the approach produces valid architectural models that abstract from the source code.

2. Foundations

This section presents the foundations of this work, focusing on model-driven software engineering, component-based software engineering, architecture description languages, and software architecture reverse engineering. It emphasizes the intersections between our reverse engineering work and these key concepts.

2.1. Model-Driven Engineering

Model-driven engineering (MDE) emphasizes the use of models as fundamental elements throughout a development process, so that they represent the system at different levels of abstraction [11]. Models represent natural or artificial originals for a specific purpose (pragmatism) and include only attributes that serve that purpose (reduction) [12]. A metamodel, in contrast, is a model of models, describing their properties and elements [13]. In model-driven software development (MDSD), models and metamodels are employed alongside code during software development, emphasizing modeling and *model transformation*. This entails refining abstract models and translating them into technical implementations [14]. The Eclipse Modeling Framework (EMF) is the most widely used framework for model-driven development. It provides a meta-metamodel called *Ecore*, which enables the development of domain-specific metamodels [15].

Model transformations enable the transformation of one model into another of a different type or level of abstraction. They can be categorized into two types. Endogenous transformations involve source and target models of the same metamodel and are typically used for refinement or abstraction. Conversely, exogenous transformations involve source and target models of different metamodels and are often used for platform-specific implementations [16]. Typical transformation languages are QVT [17] or ATL [18]. Besides these specialized transformation languages, general purpose languages (GPLs) such as Java or Xtend [19] can also be used for model transformations [20]. These languages offer a wider range of features and high flexibility, making them suitable for various tasks. The advantage of GPLs is their adaptability [21].

2.2. View-based Modelling

Stakeholders typically focus on a particular subset of the system, and their ability to hide the inherent complexity of the system allows them to focus solely on their concerns [22]. In addition, certain stakeholders may have limited access to the model or be prevented from modifying parts of it. View-based development facilitates the separation of concerns and provides stakeholders with only the information necessary for their individual concerns [23]. Views are part of the larger domain of architectural description languages and are used to effectively manage the complexity of large software systems. These techniques divide systems into multiple views or perspectives, each of which addresses different aspects of the system. Views can represent structure, behavior, or usage, while perspectives offer different interpretations of these views. Model transformations allow defining and updating views, enabling tasks such as code generation, behavior simulation, or system property analysis. The *4+1 View Model* introduced by Philippe Kruchten [24] and the rational unified process (RUP) [25] are among the most recognized methods in this area.

ISO/IEC 42010 is a standard for architectural description practices, outlining the relationship between models and views in representing system architecture. It mandates model-view consistency to ensure coherence across different representations, with a view focusing on specific concerns and a model as an abstract representation depicting architectural elements, relationships, and properties [26]. The standard distinguishes between *projective* architectural views and *synthetic* architectural views. Projective views are derived from an underlying model, whereas synthetic views and their associated relationships constitute the underlying model. *Projective Views* provide a means of establishing perspectives on the underlying model, allowing stakeholders to examine the model from different *viewpoints* [27]. A view is projected from a model by applying a particular *view type* to the model [28]. As a result, a view type acts as a metamodel for the view [29], outlining the elements and relationships that it can contain. Therefore, views abstract from the totality of the modeled information and its complexity, making them an optimal solution for managing stakeholder interactions with the model [23].

2.3. Component-Based Software Engineering

Component-based software engineering (CBSE) is a software engineering paradigm that advocates the construction of software systems using reusable, interchangeable elements called components. This methodology promotes modularity, reusability, and separation of concerns, providing an efficient strategy for managing the complexity of large software systems. CBSE can be implemented at various levels of granularity, ranging from small components within a software system to large subsystems that make up an entire application. It emphasizes the development of reusable components that encapsulate different functionalities and interact through well-defined interfaces [30]. Components are independent software units that are loosely coupled, which promotes flexibility and adaptability in system design [31].

2.4. Architecture Description Languages

Architecture description languages (ADLs) are specialized modeling languages to describe the structure and behavior of a software system and the non-software entities with which it interacts. A system is represented as a collection of software components, their interconnections, and their significant behavioral interactions [26].

2.4.1. Unified Modeling Language

The Unified Modeling Language (UML) is an architecture description language (ADL) that serves as a visual language for designing, documenting, constructing, and presenting software and other systems [13]. UML model elements, the basic units of a model, are used to represent the architectural design of a system in diagrammatic form. A named element, a model element that can be assigned a name, supports the use of a string expression for naming, allowing template parameters to be included in model element names. UML relationships represent the connections between structural, behavioral, or grouping elements. Often referred to as a link, a UML relationship describes how multiple elements can interact during system execution. Artifacts in UML models represent a piece of information that is used or produced by the operation of a system. Artifacts can take the form of model files, source files, or development

deliverables [13]. The UML can be effectively represented using the PlantUML tool [32]. This open-source tool is specifically designed to facilitate the creation of UML diagrams and supports a wide range of diagram types.

2.4.2. Palladio Component Model

The Palladio Component Model (PCM) is an ADL designed to model, analyze, and predict quality attributes of software systems. It encapsulates both the static and dynamic structure of component-based software architectures, enabling software architects and engineers to reason about system properties and identify optimization opportunities during the design phase [33]. The Palladio approach is model-centric, with structural, behavioral, usage, and deployment views serving as Palladio's first-class entities, representing the primary elements of information about a software system. The structural view focuses on the static architecture, capturing components, interfaces, and assembly connectors. The behavioral view focuses on the dynamic aspects, capturing interactions and message flows during runtime. The usage view focuses on the use of the system by users or external systems, capturing usage scenarios and profiles. The deployment view focuses on the allocation of components to resource containers and the distribution of the system across the hardware infrastructure. By fusing and analysing information from these views, software architects gain a comprehensive understanding of system behavior and properties, helping them make informed design decisions. Key features and concepts of the PCM include components, assembly connectors, resource containers and requirements, and ServiceEffectSpecifications (SEFFs). Components serve as the basic building blocks of the system, with defined interfaces for interaction. Assembly connectors define the communication patterns between components. Resource containers represent the environments in which components operate, providing CPU, memory, and storage resources. Resource requirements detail the resource needs of components, while SEFFs define the impact of component services on resources [34].

2.5. Software Architecture Reverse Engineering

Reverse engineering of software architecture is a process that attempts to reveal the underlying structure of a software system, including its components and how they are interconnected. This process is particularly useful for understanding systems that have evolved or that lack formal documentation [35]. However, manual reverse engineering of a software architecture can be a challenging and labor-intensive task, especially for large and complex systems. Automating this process typically involves code analysis to identify components, interfaces, dependencies, and other architectural elements [36]. Although automated reverse engineering can be helpful, the resulting architectural model is not always complete or accurate, and may need to be manually reviewed and refined. Reverse engineering is always ambiguous because the correct results depend heavily on the chosen mapping between architecture and source code, which is often implicit and exists only in the minds of software developers or architects. Furthermore, the quality of the extracted architecture model depends on the quality of the techniques used to extract it. Since an architecture contains aspects that are not present in the source code, reverse engineering must use more than just source code as input to achieve the best possible results [37].

2.5.1. Model-Driven Reverse Engineering

Model-driven reverse engineering (MDRE) uses the principles of MDE to analyze and understand existing software systems. It uses models to represent the structure, behavior, and other aspects of the system, providing a high-level abstraction that facilitates analysis, understanding, and modification. Compared to traditional reverse engineering methods, MDRE offers improved scalability, maintainability, and reusability of results [38]. MDRE enables reverse engineers to focus on understanding the system without being overwhelmed by the complexity of the code [39]. MDRE aims to transform disparate software development artifacts into unified models. The MDRE process involves two primary steps [40]: 1) transforming the software system into a set of models of software development artifacts without omitting necessary information, and 2) generating the required output models through model transformations.

2.5.2. Metrics for Evaluating Reverse-Engineered Models

The evaluation of reverse-engineered models against a gold standard model provides insights into the accuracy and effectiveness of various reverse engineering techniques and tools. The metrics commonly used for this purpose include precision, recall, and F_1 score. The precision, recall, and F_1 metrics facilitate a comparison between the model's predictions and a gold standard, providing an objective means to compare the performance of reverse engineering approaches [41]. Precision (also called positive predictive value) is the proportion of relevant instances among the instances found, i. e. $p = \frac{TP}{TP+FP}$. In the context of reverse engineering, it can be interpreted as the ratio of correctly reverse-engineered elements to all elements that the model identified as belonging to a given class. Recall (also called sensitivity) is the proportion of relevant instances retrieved, i. e. $r = \frac{TP}{TP+FN}$. In this context, it can be interpreted as the ratio of correctly reverse-engineered elements to all elements that truly belong to a given class according to the gold standard. The F_1 score is the harmonic mean of precision and recall, i. e. $F_1 = 2 \times \frac{p \times r}{p+r}$.

3. Approach

We now present our main contribution, a **R**everse **E**ngineering **T**echnique with **R**efinement **I**ntegration of **E**xtracted **V**iews with **E**lements and **R**elationships (*Retriever*). This approach enables reverse engineering of software architecture models from heterogeneous artifacts. In this section, we present our *Retriever* approach conceptually, while the technical implementation is described in the following section 4. The presentation of our approach consists of several subsections, including terminology, extracting views from artifacts, and composing and refining views. The reverse engineering approach uses existing software artifacts to extract structural and behavioral views using rule-based processes. These views are then refined to ensure completeness and consistency, with model-driven composition connecting the various elements. The process concludes with a transformation that produces output views tailored to user needs. These views provide a comprehensive understanding of software systems and support model-based analysis and quality prediction.

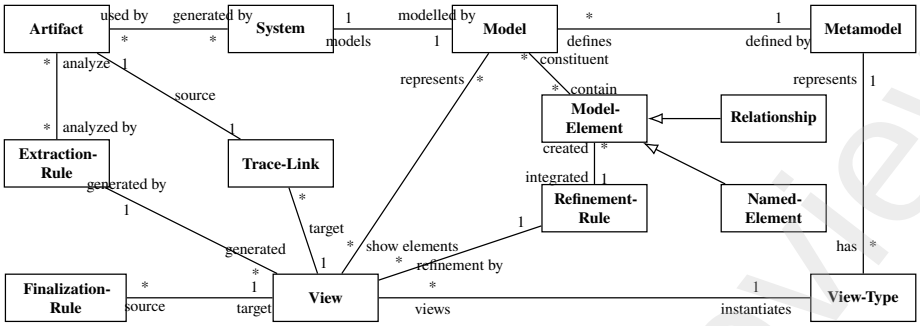


Figure 1: View-based reverse engineering terminology used in this approach.

3.1. Reverse Engineering Terminology

As part of our approach, we present a new terminology that serves as the approach conceptual framework, providing a nuanced understanding of its critical elements. A high-level view of the terminology concepts and their relationships is shown in fig. 1. Our terminology is based on the terms *Metamodel*, *Model*, *ViewType*, and *View* of the UML [13] and the terms *System*, *Artifact*, *ModelElement*, *NamedElement*, *Relationship* of the ISO/IEC 42010 [26] already introduced in section 2. For our terminology, we have related these terms to each other and extended the existing terminology to include the terms *TraceLink*, *ExtractionRule*, *RefinementRule*, and *FinalizationRule*.

A *TraceLink* is an association that connects model elements shown in a view (the target) to artifacts used or produced by a software development process or by the deployment and operation of a system (the source). An *ExtractionRule* defines how a particular concept is applied in an artifact and how it affects the system. These extraction rules are implemented as model-to-model transformations that create instances of views based on the artifacts that represent parts of the system. A *RefinementRule*, another type of model-to-model transformation, first checks whether a model element in the view meets certain criteria. If the criteria are met, the view remains the same. If not, the transformation generates new model elements, called implications, that must be incorporated into the view through this transformation to satisfy the criteria. A *FinalizationRule* describes how the model elements shown in a view are transformed into a new view. This transformation, which occurs during the final transformation process, acts like a filter, selecting elements from the original view and merging them into a new target view based on specific conditions.

The separation of concerns principle helps control complexity in developing large software systems. Applying this principle in model-based reverse engineering results in more than one system model being developed from different perspectives. Therefore, our approach to model-based reverse engineering can also be viewed as a series of refinement steps: More abstract models are transformed into more detailed models closer to the actual system. The critical requirement of this refinement process is that the transformations preserve semantics. These models can be refined independently along a single path. However, these models must be integrated at some point to obtain a complete system. The model composition can be considered the transformation of at least two

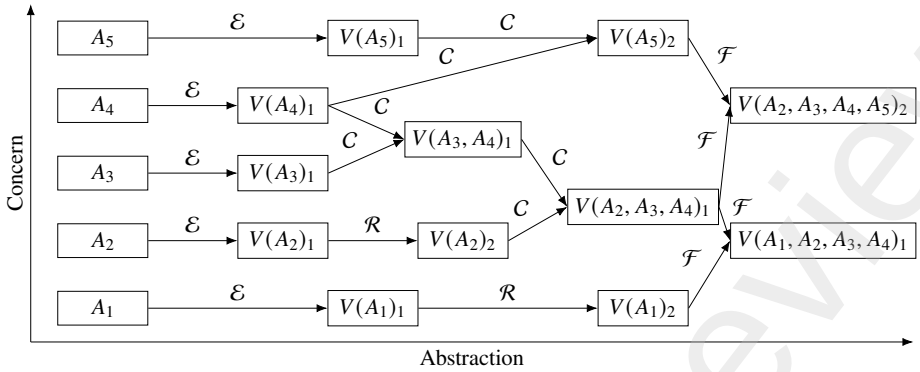


Figure 2: Two-dimensional arrangement of models, where the vertical dimension represents the concerns and the horizontal dimension indicates the level of abstraction of the models.

input models into a composite model. Refinement and composition transformations must be semantically preserving, since the resulting models represent the same system as the input models. This refinement and composition approach is based on the transformation scenarios in software and data engineering described by Kurtev [17], which we transfer to the scenarios of model-based reverse engineering.

Figure 2 shows an example of the arrangement of models in a two-dimensional space. The vertical dimension shows that views can be organized according to the problems they solve, called concerns. At each stage of our model-based reverse engineering approach, models can deal with multiple concerns. The horizontal dimension indicates the level of abstraction of the models. On the left are the artifacts A from which the more concrete views $V(A)$ of the system are transformed. An extraction $\mathcal{E}$ creates an instance of views based on the artifacts that represent parts of the system. In a refinement $\mathcal{R}$, the information from the artifact is transformed into a more concrete view. This view can then be refined or combined C with information from the artifact or other views. A transformation $\mathcal{F}$ selects items from all views and merges them into a new target view based on the selected view. Our idea here is that only $\mathcal{E}$ is technology or project specific. C, $\mathcal{R}$ and $\mathcal{F}$ are generic for component-based systems. Therefore, the approach allows new combinations of the extracted views, where only E needs to be adapted or extended.

3.2. Reverse Engineering Process

Algorithm 1 describes the entry into our reverse engineering process. This process aims to transform existing artifacts, some of which have different concerns, into views, some of which also have different concerns, to represent the system in a comprehensive model. The input variable E is the set of all extraction rules applied to the artifacts to generate views on them. The input variable R is the set of all refinement rules that check for model elements in the views whether they meet certain criteria and, if not, create new model elements that must be integrated into the view to meet the criteria. The Finalization F is the set of all finalization rules that ultimately combine model elements from all existing views into a new target view. The $\leftarrow$ symbol describes assigning a value to a variable.

Algorithm 1 The reverse engineering process of our approach.

```
function ReverseEngineering(E, R, F)
    A ← FindAllArtifacts
    V_e ← ExtractViews(A, E)
    v_q ← CompositionRefinement(V_e, R)
    return Finalization(v_q, F)
end function
```

The first step in our reverse engineering approach is identifying and capturing the existing artifacts related to the software system under analysis. This includes source code, configuration files, test cases, and other relevant artifacts. This is done by so-called finders, so that the existing artifacts can be analyzed by the extraction rules. These finders are tasked with finding all files within the specified locations and converting them into a model that serves as the basis for further extraction. For example, a finder can use third-party source code parsers to analyze the found source code and convert it into a model, which can then be used for further extractions.

Our approach provides the infrastructure for this, which can be extended by custom finders. Finders can be registered and then called to find and analyze artifacts. The FindAllArtifacts function identifies and collects all artifacts and returns them in a set by calling the existing finders. These artifacts can be defined by one or more pathnames on a file system, or by one or more URLs to a repository in a version control system. For example, components and interfaces could be implemented in a programming language such as Java. This could involve integrating a finder based on a third-party parser library that reads Java files to analyze them for extraction. Components could also be deployed in a container virtualization such as Docker. In this case, a Docker Compose artifact finder could be integrated to read the YAML files.

In the second step, views of the software system are extracted from the artifacts. To accomplish this, the ExtractViews function applies the extraction rules passed to it as model-to-model transformations to the artifacts passed to it. The extracted views are returned in a new set. These views can include the component structure, dependencies, and distribution to each node, among other things. This step allows a higher level of abstraction and further model-driven processing of the artifacts. This step is detailed in section 3.3. The extracted views may be incomplete or inconsistent. Therefore, in the third step, the various information from the previous modeling steps is refined and merged to ensure completeness and consistency. To accomplish this, the CompositionRefinement function applies the refinement rules passed to it as model-to-model transformations to the views passed to it. This function refines and extends the composition of elements and relationships within the views, returning it as the set of all elements. This step is detailed in section 3.4.

In the final step, the refined and composed elements of the extracted views are transformed into a target representation, such as a unified architectural model. To accomplish this, the Finalization function applies the final model-to-model transformations to the view passed to it. The output models thus transformed, suitable for model processing

such as quality prediction, are returned in a new set. The result of this last step is the comprehensive outcome of our *Retriever* approach. This step is detailed in section 3.5.

3.3. Extract Views from Artifacts

The goal of this step is to automatically extract individual views of a software system from existing artifacts such as source code and configuration files. Algorithm 2 shows our implementation for extracting views on the specified artifacts. First, the structural view types are created, representing different views of the system structure being modeled. The views typically include components with their interfaces or ports. They also include how the components are connected to each other through dependencies and connectors, or distribution relationships of computer nodes and components with connectors. Based on this, the behavioral view types are created for these previously discovered elements to preserve the interactions between the components. The views typically include the description of the externally visible actions of a component service as an abstraction of its internal behavior. They also include the relationship between the provided component interface and the required component interface, thus capturing the system dynamic behavior by including component interactions and changes in component internal states.

Algorithm 2 The rule-based extraction of views from given artifacts.

```
function ExtractViews(A, E)
    L ← ExtractStructuralViews(A, E)
    return ExtractBehavioralViews(L, E)
end function
```

To create a behavioral view for each signature a component provides, trace links must be created. These trace links contain information about how classes, their interfaces, and methods are mapped to architectural elements such as components, interfaces, and signatures. To create these trace links, the mapping of classes to components is linked when components and interfaces are created, and the mapping of code methods to architectural signatures is linked when signatures are created.

3.3.1. Extract Structural Views

The goal of this step is to automatically extract structural views of a software system from existing artifacts such as source code and configuration files. This includes, for example, the component structure of the system, the dependencies between components, and the distribution of components across nodes. This model transformation in model-driven software development is a process by which textual descriptions or specifications of a system are transformed into views that can be further processed and analyzed using model-driven development tools and techniques. This enables further processes at higher levels of abstraction. This step includes reverse engineering through knowledge modeling of the technologies used and helps to determine, for example, how components are implemented with specific technologies. When transforming artifacts into models, model transformations express certain aspects of a system requirements or design through so-called mapping rules. In our approach, our model transformations capture rules about how concepts are implemented in technologies and how they affect the system

architecture. These rules can be technology-specific or project-specific, promoting flexibility and reusability. Our approach, therefore, supports different technologies and can be used in different ways in other software projects. Trace links allow tracing how classes, interfaces, or methods in the source code were mapped to architectural elements such as components, interfaces, or signatures. This means that the trace links are stored as references to the original artifacts for the extracted view elements. In our approach, when the extraction rules detect components or interfaces, a trace link is created from classes to components, and when the rules detect signatures, a trace link is created from methods to signatures. Trace links are limited to extraction, so they are not needed for composition, refinement, or finalization.

Algorithm 3 shows our implementation for rule-based extraction of views on the specified artifacts. For the Cartesian product of all used artifacts A with all extraction rules E, we first check whether a rule $r \in E$ applies to an artifact $a \in A$, where $r(a)$ is a predicate indicating that the rule r applies to the artifact a. If so, that rule is applied to the artifact. This extraction process takes as input an artifact a and is based on an extraction definition r that specifies how the artifact can be transformed into a view $v \in V$ $\mathcal{E}_r(a) \rightarrow V$. For each extracted view, it is stored as a trace link with the used artifact in the set L. This set is returned at the end of the function, containing all extracted views with their associated trace links.

Algorithm 3 The rule-based extraction of structural views from given artifacts.

```
function ExtractStructuralViews(A, E)
  L ← { }
  for all a ∈ A, r ∈ E do
    if r(a) then
      L ← L ∪ (E_r(a), a)
    end if
  end for
  return L
end function
```

3.3.2. *Extract Behavioral Views*

Applying quality prediction approaches to reverse engineering models requires a representation of component behavior. We transform the behavior descriptions available in the code model into abstract behavior at the component level. For example, all internal component actions are reduced to individual nodes in the behavior model. Any control flow that does not impact other components (no calls to other components) is abstracted. Algorithm 4 shows our implementation for rule-based extraction of behavioral views for the specified artifacts, with the associated links to the previously extracted structural views. After the trace links are created, the transformation rules are applied to generate the behavioral views atomically. The extraction rules in this step create a view of the system behavior. For the Cartesian product of all used trace links L with all existing rules E, it is first checked whether an extraction rule $e \in E$ applies to a trace link $(v_s, a) \in L$. Here $e(v_s, a)$ is a predicate that indicates whether the rule r applies to the structural view and its associated artifact a.

Algorithm 4 Rule-based extraction of behavioral views from given trace links between structural views and artifacts.

```
function ExtractBehavioralViews(L, E)
  V_b ← { }
  for all (v_s, a) ∈ L, e ∈ E do
    V_b ← V_b ∪ v_s
    if e(v_s, a) then
      V_b ← V_b ∪ E_e(v_s, a)
    end if
  end for
  return V_b
end function
```

If so, this extraction process is applied, which takes a trace link as input and is based on an extraction definition e that specifies how to transform the structural view and artifact into a behavioral view $\mathcal{E}_e(v_s, a) \rightarrow V$. These extraction rules perform a control flow transformation of the source code for each class or interface method corresponding to an interface signature. This transformation examines each statement and, if necessary, generates an action in the behavioral view. The most important statements in this context are calls from one component to another, representing data flow and control between components. Component-internal calls, that is, method calls that call a method of the same class or a method of a class within the same component, are also part of the control flow transformation. Other control flow elements, such as conditional statements or branching and looping or iteration, are also represented in the behavioral view. The statements are grouped for better abstraction from the source code. Each extracted behavioral view is stored, along with the views from the given trace link, in the set v_b and returned at the end of the function.

In addition to extracting behavioral views through static code analysis, our approach allows for further extraction of other views. In a previous work [42], we proposed a method for automatically annotating components of an architectural model with vulnerability information based on the trace link model. The goal of this method is to extract security information from source artifacts and integrate it into an existing architecture-based security model to enable model-based security risk assessment. In addition, our approach allows the composition and refinement of views from artifacts generated at runtime. For example, behavioral views can be integrated from log files, such as measurement logs from monitoring tools. These extracted behavioral views can then be composed and refined using our approach in the same way as behavioral views from static information.

3.4. Composition and Refinement of Views

The model-driven composition and refinement step links model-providing processes, such as reverse engineering, with model-consuming processes, such as quality prediction. To that end, we have developed and implemented a framework to automate model-driven knowledge transformation for software systems. Our framework is metamodel-independent and can be adapted to different target metamodels. It formulates refinement rules

using GPLs such as Java, Xtend, or transformation languages such as QVT or ATL. The framework supports model-consuming processes by linking model-providing processes with model-consuming processes. It consists of three processes: information discovery, information processing (composition and rule-based refinement), and transformation.

Algorithm 5 Processing pipeline for provided models including discovery, composition, and the rule-based refinement step.

```
function CompositionRefinement(V_e, R)
  v_q ← { }
  for all e ∈ Discover(V_e) do
    v_q ← ProcessElement(v_q, e, R)
  end for
  return v_q
end function
```

Algorithm 5 shows the processing pipeline for composing and refining the previously extracted views. In the Discover function, heterogeneous information is retrieved from model-providing processes, classified into named elements and relationships, and translated into the model-consuming domain. The resulting information is provided as a wrapper for the processing phase. The processing phase consists of two main steps: composition, where named elements and relationships are added to an intermediate model and their dependencies are recursively processed, and rule-based refinement, where views are manipulated to conform to specific rules. This comprehensive approach ensures seamless use of model-driven knowledge and improves software evolution processes in modern software systems.

Algorithm 6 The discovery function is responsible for extracting heterogeneous elements from previously extracted views and making them available to subsequent processes in a more uniform manner.

```
function Discover(V_e)
  v_q ← { }
  for all e ∈ V_e do
    v_q ← v_q ∪ e
  end for
  return v_q
end function
```

Algorithm 6 shows the discovery function, which extracts the previously heterogeneous views of the system to make them consistent for subsequent processes. This function extracts all elements from the set of views V_e and returns them as a new set of views v_q. Algorithm 7 shows the ProcessElement function, which takes as input the set of all already composed and refined elements, the new element to add to the given set, and the refinement rules to apply to the new element. The function works by first adding the new element to be refined to the given set. Adding the element to the set of refined elements is referred to as composition. In the next step, all given refinement rules

are applied to this element. This refinement process can produce new model elements as implications i, which are also processed recursively using the ProcessElement function. The function continues until there are no more refinement rules and resulting implications to process. At the end, the function returns the set of all refined elements, including the new element.

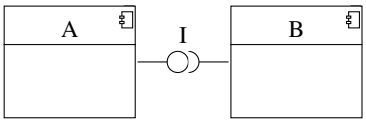
Algorithm 7 The process element function applies the refinement rules to the given element, and also applies these rules recursively to the resulting implications that need to be processed after the current element is processed.

```
function ProcessElement(v_q, e, R)
  v_q ← C(v_q, e)
  for all r ∈ R do
    for all i ∈ R_r(v_q, e) do
      v_q ← ProcessElement(v_q, i, R)
    end for
  end for
  return v_q
end function
```

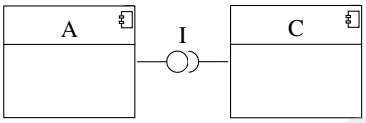
3.4.1. Composition of Views

Our model composition combines various model elements into a more comprehensive and complex system view. Let V be a view and e a model element that is not part of that view $e \notin V$. The model elements represented by V and the single model element e are based on the same metamodel and are valid accordingly. The symbol C represents the model composition. If e is not yet a member of the set V, then the model composition is a new view containing all elements of V and e. It is defined as $C(V, e) := V \cup \{e\}$. Identical model elements are composed into a single element. The element name is used to identify the named model element. Two elements with the same name are identical and are composed into a single element, where model composition is a recursive operation that can be applied to all model elements. Relationships to be composed are identified by their type and by comparing the identity of their referenced model elements. As elements are added to the model, they are combined into a richer and more complex system view.

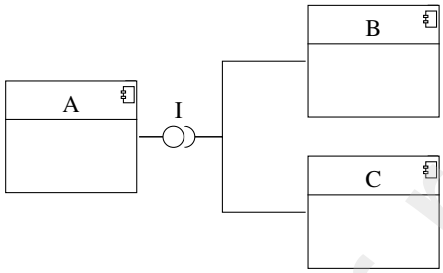
Figure 3 shows an example of combining a new element into a view. Figure 3a shows the simplest form of relationship between components A and B, where component B requires an interface I, which is provided by component A. Figure 3b shows the view of components A and C before the relationship AB was added. Component C requires an interface I from component A. Figure 3c shows the view created by the composition, which includes all three components, their relationships, AB and AC, and the interface I. In this example, a relationship AB is added to a view of components A and C with its relationship AC, creating a new view with three components. The two components A with the same name in fig. 3a and fig. 3b are identical and are combined into one component in fig. 3c.



(a) Simplified representation of a relationship AB as a UML assembly connector, where the two associated components are also shown for clarity.



(b) View of two components A and C and their relationship AC, into which relationship AB is to be inserted by composition.



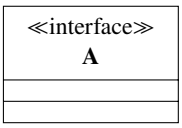
(c) The view created by the composition. This new view contains three components, A, B, and C, and their relationships, AB and AC.

Figure 3: UML-like graphical representation of an example of view composition. A relation AB is added to a view of components A and C with its relation AC, creating a new view with three components.

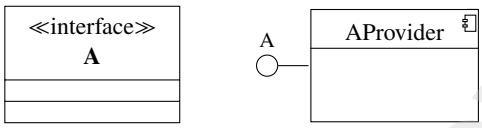
3.4.2. Refinement of Views

Model refinement describes the automated process of improving the quality of an extracted view by adding more detail or accuracy. The idea is to create models at different levels of abstraction and connect them through refinement to support the process of reverse engineering. This refinement process can create new model elements called implications. Let V be a view and e be a model element that is part of the view $e \in V$. The model elements represented by V are based on the same metamodel and are valid accordingly. The symbol $\mathcal{R}_r$ represents the model refinement for the refinement rule r. The model refinement is a transformation of the view V into a new view $\mathcal{R}_r(V, e) = V'$. A refinement rule is a model-to-model transformation that checks whether the given model element e in the view satisfies certain rules. If the rule is satisfied, the view remains unchanged. However, if the model element does not satisfy the rule, the transformation creates new model elements called implications. These new implications must also be added to the view in order for it to satisfy the rule.

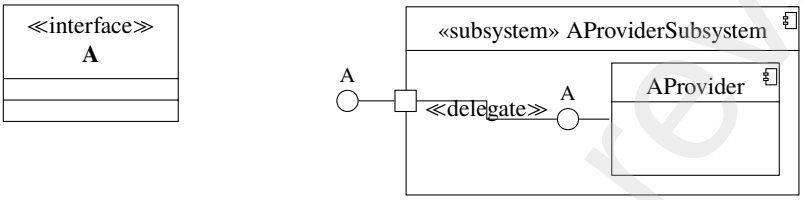
Figure 4 shows a UML-like example of a refinement process using two exemplary refinement rules. The first rule is called $r1$ and the second rule is called $r2$. An arbitrary view V satisfies the refinement rule $r1$ if each interface of V is provided by at least one component. The refinement rule $r2$ is satisfied if each component of V is encapsulated by a subsystem. Moreover, the provided interfaces of encapsulated components have to be delegated to the subsystem to satisfy $r2$. Figure 4a shows an arbitrary view V created by the composition of a view with an interface A. After the composition V does not satisfy $r1$ since no component provides A. A refinement for the refinement rule $r1$ is used to transform V to satisfy $r1$. Figure 4b shows the view after the transformation. The new view V' is created by the composition of V and the implications of the refinement for $r1$. The view V' satisfies refinement rule $r1$. A single implication is required to satisfy



(a) Simplified representation of an arbitrary view after composition of an interface A. No component provides the interface. Therefore, refinement rule r_1 is unsatisfied and the refinement process is started.



(b) The view after the composition of interface A and the refinement for refinement rule r_1. The refinement leads to an implication in the form of a component AProvider providing interface A. After the composition of the implication, the view satisfies refinement rule r_1 but leads to refinement rule r_2 being unsatisfied.



(c) The view after the composition of interface A, the refinement for refinement rule r_1, and the refinement for refinement rule r_2. The view satisfies all refinement rules. As a consequence, the recursive refinement process initially started by the composition of interface A terminates.

Figure 4: UML-like graphical representation of an example of recursive view refinement using two exemplary refinement rules r_1 and r_2. Refinement rule r_1 states that every interface has to be provided by at least one component. Refinement rule r_2 states that every component is encapsulated by a subsystem and its interfaces are delegated to the encapsulating subsystem.

$r1$. This implication is a component called $AProvider$ providing A. After the first refinement V' does not satisfy $r2$, leading to another refinement for refinement rule $r2$. Figure 4c shows the view V'' created by the composition of V' and the implications of the refinement for $r2$. V'' satisfies $r1$ and $r2$. To satisfy $r2$ an encapsulating subsystem as well as an interface delegation are composed with V' creating V''. This refinement does not break any refinement rule. As a consequence, the recursive refinement process terminates. The result of the refinement process is V'', which is a composition of V with the implications and recursive implications required to satisfy all refinement rules.

3.5. Final Transformation of Views

Finally, the last transformation process takes the intermediate view composed and refined during information processing and transforms it into one or more output views. These output views can be disjoint and based on different metamodels. These output views can then be used for system documentation or as input for other model-intensive processes such as quality prediction. Algorithm 8 describes a function called FINALIZATION, which takes two arguments, v_q and F. The function is responsible for transforming an intermediate view v_q into one or more output views V_f tailored to the end user's specific needs. The function iterates over all model-to-model transformations $f \in F$ and applies them to v_q, i. e. $\mathcal{F}_f\left(v_q\right)$. The output of the transformation $\mathcal{F}_f$ is then added to the set of target views V_f, which is returned at the end. A single transformation takes as input a source view v_q of the system, showing all named elements and relationships. It is based on a transformation definition f that specifies how the elements are transformed into a set of target views V_f. As part of the transformation

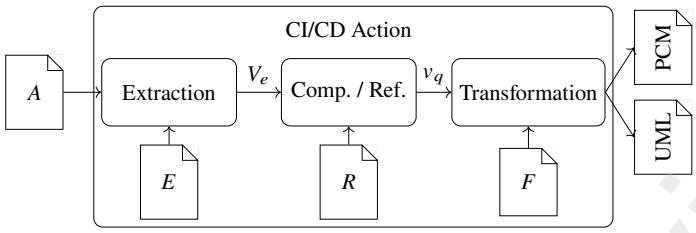


Figure 5: Schematic of the approach with three main steps, each processing different information.

process $\mathcal{F}_f$, this model-to-model transformation acts like a filter that selects elements $e \in v_q$, from the source view and combines them according to certain criteria into a new target view $\mathcal{F}_f\left(v_q\right) \rightarrow v_f \in V_f$.

Algorithm 8 The transformation function is responsible for transforming an intermediate view into one or more output views tailored to the end user's specific needs.

function FINALIZATION(v_q, F)
  $V_f \leftarrow \{\,\}$
  for all $f \in F$ **do**
    $V_f \leftarrow V_f \cup \mathcal{F}_f(v_q)$
  end for
  return V_f
end function

4. Technical Realization

To promote transparency and accessibility, we have released our implementation of the Retriever approach as open-source on GitHub [43] under the Eclipse Public License 2.0 (EPL2). This allows researchers, developers, and practitioners to access the code, contribute to its development, and adapt it to their specific needs. Our approach is implemented using the Eclipse Modeling Framework and is available as a plug-in to the Eclipse platform. The technical implementation of our approach combines third-party libraries for input processing, leverages the PCM metamodel for detailed system analysis, and provides flexibility by outputting both PCM and UML software architecture models. Our approach is available as an open-source Eclipse product with both graphical and command-line interfaces. We have also implemented the command-line interface as GitHub Actions for use in continuous integration (CI) and continuous delivery (CD) pipelines. With these GitHub Actions, our approach can be easily integrated into CI/CD pipelines on GitHub and, for example, automatically generate both PCM and PlantUML models for the existing artifacts as output on every commit.

Figure 5 shows the approach schematically and describes which parts process which information. The arrows represent the data flow, and the input and output data are represented as UML notes whose labels correspond to the pseudocode in algorithm 1. The rounded rectangles represent the three main steps of our approach. These three

steps are described in more detail in the following three subsections. The large rounded rectangle represents deployment as a GitHub Action, to make it easy to use our approach in the build, test, and deploy pipeline on GitHub.

4.1. Artifact Extraction Implementation

Our approach relies on various libraries to facilitate the initial discovery of artifacts. For example, our implementation currently includes a Java source code finder based on the Eclipse Java Development Tool and supporting Java 21 archives and properties files, an ECMAScript 6 finder based on the OpenJDK Nashorn engine, and a SQL statement finder based on the JSqlParser. There are also finders for JSON, XML, and YAML configuration files, also based on various third-party libraries. The extraction rules for model-to-model transformations take the models generated by the finders as input. The extraction rules are implemented using Xtend. We use Xtend, a general-purpose programming language that follows imperative and functional programming paradigms and allows flexible rule structuring with both imperative and declarative transformation definitions. Xtend, a versatile and eloquent dialect of Java, combines the strengths of both Java and script languages. We implemented extraction rules for the following technologies: Docker, EcmaScript, Gradle, JaxRS, Maven, and Spring. We preselected these technologies because they are among the most widely used technologies for component-based RESTful Web services [44, 45] and contain important architectural information. However, our approach also allows adding additional extraction rules for already supported or additional technologies that can be dynamically loaded at runtime.

4.2. Composition and Refinement Implementation

Our approach uses the PCM metamodel for processing. The PCM metamodel provides a standardized representation of software architectures, making it a suitable choice for our purpose. It allows us to capture essential architectural concepts such as components, connectors, and interfaces. The PCM serves as an intermediate representation for combining the extracted information from the extraction transformations for the composition and refinement of views. However, the concepts of our approach are also applicable to other ADLs. The model-to-model transformation rules are implemented in Java. We use Java, a general-purpose programming language that follows imperative and object-oriented programming paradigms and allows easy structuring of imperative rules using object-oriented concepts such as inheritance for transformation definitions. The choice of Java was influenced by its compatibility with our composition and refinement framework, which is also Java-based. This compatibility allows for better integration and uniformity of our components. It also simplifies maintenance tasks because the entire code base is in a single language. The underlying framework also allows the integration of transformation languages such as Xtend, ATL, or QVT. We have implemented these refinement rules generally for the domain of component-based systems, i. e. they can be reused in any system within this domain. They can, therefore, be combined with any extraction rules and are technology-independent.

4.3. Final Views Implementation

For the final transformation of views, our approach uses both PCM and PlantUML models as output. The PCM output captures information specific to software quality

and architectural aspects, allowing users to perform model-based quality analysis. The PlantUML output, on the other hand, provides a broader view of the software system's architecture and is used to create UML diagrams that provide a more widely recognized graphical representation of the system's structure. We have implemented the final view transformation based on finalization rules for outputting views based on the PCM and UML metamodels. The concepts of these finalization rules are also applicable to other ADLs. These finalization rules are implemented in the domain of component-based systems, i. e. they can be reused in any system within this domain. They can therefore be combined with any extraction and refinement rules and are technology-independent.

5. Evaluation

We structured our evaluation based on the goal question metric (GQM) approach [46]. The GQM approach is a goal-oriented method for improving and measuring software quality. It defines a three-stage measurement model: 1. A goal is set for an object considering various reasons, quality models, perspectives, and specific environments. 2. A set of questions is used to define models of the object under study and to characterize the assessment or achievement of a specific goal. 3. Each question is associated with a set of metrics to answer it measurably.

While the evaluation question aims to assess the value of the effectiveness of the *Retriever* approach, the research question aims to generate new knowledge or understanding about a topic. This approach ensures that measurements are top-down, goal-oriented, and model-based, providing a framework for interpreting data against stated goals. The following GQM plan outlines our evaluation goals, questions, and metrics:

G1: Applicability of the *Retriever* approach to relevant software systems to generate an abstraction of the system as an architectural model. *Q1.1:* Is the *Retriever* approach applicable to relevant software systems with different technologies? *(M1.1.1)* The relevance of GitHub projects is measured by four key metrics: commits, contributors, forks, and stars. These metrics serve as valuable indicators of a project's popularity and distribution. They can help to evaluate the relevance of a project and determine whether it is worth using or contributing to. More commits indicate active maintenance and a large developer base. More contributors indicate a robust developer community, and a higher likelihood of ongoing maintenance. More forks indicate that other developers find the project useful or interesting. More stars indicate widespread user interest. *(M1.1.2)* For technology usage, two metrics are used: the number of lines of code (LOC) (excluding whitespace and comments) and the number of source files. A larger code base may indicate increased code complexity, which can make maintenance and readability more difficult. More source files could indicate a highly modular system, which could increase the difficulty of MDRE. *(M1.1.3)* For system views, the chosen metric is the number of different technologies, each providing a unique perspective on the system. More technologies can lead to increased view complexity, making MDRE more difficult. *Q1.2:* To what extent is the approach scalable and transferable to larger real systems, in the context of industrial applications? *(M1.2.1)* In the

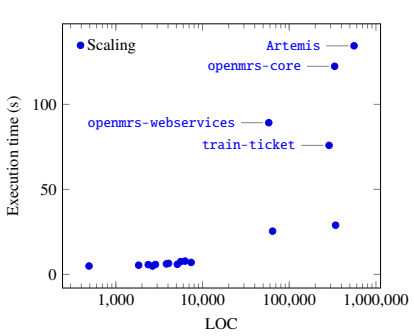
context of industrial applications, the first metric is the cumulative number of LOC across all artifacts of the software system considered in our approach. The complexity of this code can potentially hinder the speed of reverse engineering processes. *(M1.2.2)* In terms of scalability, the second metric is the average execution time of GitHub Actions for comprehensive model extraction. Using a fast reverse engineering tool can speed up results, saving engineers time and resources. *Q1.3:* Does the *Retriever* approach lead to valid architectural models that abstract from the source code? *(M1.3.1)* When assessing the validity of architectural models, the primary metric is the number of violated Object Constraint Language (OCL) constraints in the reverse-engineered models. OCL constraints, which are conditions built into EMF models to delineate constraints and limitations, signal potential architectural defects when violated. *(M1.3.2)* When it comes to abstraction, the secondary metric is the ratio of source files to the number of components in the architectural model. This ratio serves as an abstraction indicator and is critical in evaluating the reduction quality of architectural models.

G2: Accuracy of the *Retriever* approach in generating both structural and behavioral properties of relevant software systems, which together form a software architecture model. *Q2.1:* Have the structural properties of the component-based software architecture been correctly identified? *(M2.1.1)* The metrics used to measure this are precision, *(M2.1.2)* recall, and *(M2.1.3)* F_1 score for the components, connectors, and provided and required interfaces. *Q2.2:* Are the component-based software architecture behavioral properties (SEFF) correctly identified so that data and control flows exist across component boundaries that are not connected at compile time? The external calls are the link between the structure and the behavior view. Therefore, they indicate whether the approach was able to extract and combine these two views types correctly. *(M2.2.1)* The metrics used to measure this are precision, *(M2.2.2)* recall, and *(M2.2.3)* F_1 score for the external calls in the SEFF.

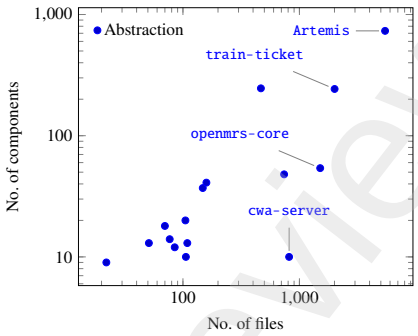
5.1. Applicability to Relevant Software Systems

For the first goal of the evaluation, we reviewed statistics [44, 45] that provide information about the most popular and widely used technologies for implementing and deploying web services systems. Based on these statistics, extraction rules were implemented for the following technologies: 1. Maven and Gradle for build automation. 2. Docker for container virtualization. 3. Spring MVC and Boot for web application development. 4. EcmaScript for web client application implementation. 5. Jakarta RESTful web Services (JaxRS) for RESTful web services implementation.

We chose these technologies not only because they are among the most widely used domain technologies for modern component-based systems, such as web services or microservices. We also chose these technologies because they provide established frameworks that may contain important architectural information that can be recovered using our extraction rules. In addition, these technologies have different concerns and each contributes a different view of the system. The rules for Spring MVC and Boot have also been implemented in a common extraction rule because the rules for



(a) The ratio of the average execution time in seconds to the sum of the LOC for all artifacts analyzed in the case study. The four case studies with the highest execution times are labeled.



(b) The ratio of the number of source files to the number of components in the architectural model, indicating the level of abstraction. The four case studies with the highest number of source files are labeled.

Figure 6: Evaluation results for the applicability and scalability of the approach to generate an abstraction of the larger real systems as a model.

both technologies are very similar. The Spring and JaxRS rules are both Java-based technologies, so they use the same artifacts as input.

Based on these relevant domain technologies, relevant systems for the evaluation scenarios were identified. This selection was based partly on related work and partly on open availability. Based on open availability, the evaluation scenarios were further selected based on their popularity on GitHub. The systems selected for evaluation are listed in table 1 with their GitHub identifiers, distribution metrics, technology metrics, and metrics for scalability and abstraction of our retriever approach.

For these open-source case studies, our approach was applied as a GitHub Action within a CI/CD pipeline, as described in section 4. The automatically generated architectural models were first checked for syntactic correctness. For the textual notation of the UML diagrams, PlantUML was able to generate the appropriate diagram for each case study without error messages. As the PCM models are much more extensive, they were validated using the EMF validation framework. In addition, validation was performed using OCL constraints to ensure the syntactic correctness of the models. The PCM instances also completed model validation in Eclipse with no errors or warnings.

To evaluate the scalability of our approach on larger systems, we performed a local test on a laptop with an i7-8550U processor, 16 GB of RAM, and Windows 11 as the operating system. For each case study, we independently executed the action 10 times with the same inputs and recorded the execution times for each run. The differences in execution times were negligible, always less than one second. To verify the plausibility of our results, we compared the execution times of our local test with those of the GitHub action run as a cloud-based solution on a Linux virtual machine. The scalability of the GitHub action execution times was comparable to the local execution times. Figure 6a compares the average execution time for each case study with the size of the input data, which is the sum of the LOC for all relevant artifacts analyzed in table 1. The execution times are plotted on the linear y-axis in seconds and the number of LOC is plotted on the logarithmic x-axis. As expected, the average execution time increases with the size

System-ID	Distribution		Technology (files / code)			Reverse Engineering	
	Commits	78	Ecma	5	708	Execution Time	7.028s ± 0.044s
	Contributors	2	Gradle	2	75	Components	20
acmeair	Forks	154	Java	81	6207		
	Stars	131	Maven	1	56		
			Xml	16	349		
	Commits	48	Docker	9	126	Execution Time	6.166s ± 0.141s
	Contributors	5	Ecma	8	153	Components	10
	Forks	437	Gradle	9	691		
microservices-basics-spring-boot	Stars	692	Java	37	1385		
			Xml	9	182		
			Yaml	34	1294		
	Commits	143	Docker	4	97	Execution Time	7.496s ± 0.165s
	Contributors	1	Java	116	3147	Components	41
java-microservice	Forks	305	Maven	11	1376		
	Stars	338	Xml	9	233		
			Yaml	19	743		
	Commits	239	Docker	7	39	Execution Time	5.825s ± 0.122s
	Contributors	1	Gradle	12	663	Components	14
blog-microservices	Forks	308	Java	37	1401		
	Stars	400	Xml	2	31		
			Yaml	19	731		
	Commits	1664	Docker	5	170	Execution Time	28.925s ± 0.691s
	Contributors	30	Java	615	40602	Components	10
cwa-server	Forks	386	Json	48	24554		
	Stars	1894	Xml	31	273343		
			Yaml	117	3638		
	Commits	211	Docker	1	8	Execution Time	5.920s ± 0.115s
	Contributors	30	Ecma	1	40	Components	12
	Forks	107	Java	58	3618		
cwa-verification-server	Stars	349	Json	1	409		
			Maven	1	142		
			Xml	1	294		
			Yaml	22	610		
	Commits	143	Docker	13	43	Execution Time	5.724s ± 0.072s
	Contributors	8	Java	42	1833	Components	18
microservice	Forks	348	Maven	7	380		
	Stars	692	Yaml	8	110		
	Commits	1186	Docker	46	249	Execution Time	75.867s ± 0.858s
	Contributors	18	Ecma	992	197869	Components	243
	Forks	191	Gradle	2	13		
train-ticket	Stars	605	Java	635	37738		
			Json	170	33274		
			Maven	45	2229		
			Xml	1	30		
			Yaml	114	16619		
	Commits	7	Ecma	1	36	Execution Time	4.921s ± 0.093s
	Contributors	1	Java	15	326	Components	9
apache-spring-boot-microservice-example	Forks	10	Maven	3	108		
	Stars	9	Yaml	3	21		
	Commits	17527	Docker	5	113	Execution Time	134.416s ± 0.985s
	Contributors	30	Ecma	2680	239802	Components	733
	Forks	262	Gradle	12	776		
Artemis	Stars	383	Java	1984	216149		
			Json	480	88131		
			Maven	11	896		
			Xml	140	8103		
			Yaml	126	5173		
	Commits	6	Java	37	931	Execution Time	5.396s ± 0.140s
	Contributors	1	Json	3	309	Components	13
spring-cloud-movie-recommendation	Forks	11	Maven	6	467		
	Stars	15	Xml	5	128		
	Commits	18874	Docker	1	79	Execution Time	122.385s ± 0.737s
	Contributors	30	Ecma	21	4346	Components	54
	Forks	3504	Java	1874	189296		
openmrs-core	Stars	1273	Maven	21	4566		
			Xml	308	134587		
			Yaml	10	612		
	Commits	1730	Ecma	3	79	Execution Time	89.202s ± 0.662s
	Contributors	30	Java	672	53997	Components	48
	Forks	492	Json	6	503		
openmrs-module-webservices.rest	Stars	76	Maven	14	2253		
			Xml	43	1182		
			Yaml	1	52		
	Commits	718	Docker	3	26	Execution Time	6.466s ± 0.407s
	Contributors	30	Ecma	21	263	Components	13
	Forks	1983	Java	46	1216		
spring-petclinic-microservices	Stars	1483	Json	12	816		
			Maven	8	861		
			Xml	3	579		
			Yaml	16	300		
	Commits	290	Docker	10	58	Execution Time	7.824s ± 0.165s
	Contributors	13	Ecma	8	1664	Components	37
piggymetrics	Forks	5987	Java	97	3292		
	Stars	12740	Maven	10	684		
			Yaml	23	588		
	Commits	2998	Docker	1	29	Execution Time	25.411s ± 0.674s
	Contributors	30	Ecma	90	44099	Components	246
	Forks	4731	Java	355	18201		
WebGoat	Stars	6120	Json	5	181		
			Maven	1	852		
			Xml	5	582		
			Yaml	10	317		

Table 1: Evaluation results regarding the applicability of the approach to relevant software systems to generate an abstraction of the system as an architectural model.

of the input data. Thus, the case study with the longest execution time is the one that requires the most LOC to analyze.

To evaluate the abstraction capability of our approach for larger systems, we compare the number of source files to the number of components in the final architectural model. This ratio serves as an indicator of the reduction quality of the reverse-engineered architectural models. The number of source files was taken as the sum from table 1. Figure 6b shows the ratio of the number of source files to the number of components. The number of source files is plotted on the logarithmic y-axis and the number of components is plotted on the logarithmic x-axis. As expected, the number of components increases with the number of source files. For example, our largest case study with 5438 source files was reduced by 86.52% to 733 components.

5.2. Accuracy

For the second goal of the evaluation, we first manually created a software architecture model. This architecture model serves as a reference (gold standard) for comparing the automatically generated architecture models with our retriever approach. It should provide information about the expected components, interactions, interfaces, dependencies, and connections in the case studies. To create an accurate software architecture model for a component-based open-source software system, we performed the following 5 iterative steps for each case study. These steps combine system understanding, architectural style selection, visual representation, pattern considerations, verification, documentation, and iterative refinement to produce a comprehensive and reliable architectural model.

1. Understand the system and identify its components: Obtain a comprehensive understanding of the system's functionality, purpose, users, and tasks through documentation, source code, design documents, and other publications. Identify key components through code analysis, documentation, and available resources.

2. Select an appropriate architectural style and define the component model: Select an architectural style (e. g., microservices, layered architecture) that meets the system requirements. Define rules and conventions for developing, deploying, and operating components. Specify interface syntax, semantic, lifecycle, deployment, and runtime environment.

3. Create a component diagram: Represent components, interactions, interfaces, dependencies, and connections in a component diagram. Use standard notation (e. g., UML) or custom notation to ensure clarity and consistency. Use tools such as PlantUML to visualize the system architecture.

4. Consider architectural patterns: Examine common software architecture patterns (e. g., n-tier, client-server, microservices) and how they can improve system properties. Integrate these selected patterns into the component model and architectural style.

5. Verify and refine (iterative process): Obtain feedback from stakeholders and verify the architectural model with them. Update the model based on stakeholder input to ensure that it meets evolving system requirements and constraints.

System-ID	Structural			Behavioral		
	p	r	F_1	p	r	F_1
acmeair	0.81	0.87	0.84	0.75	0.57	0.65
microservices-basics-spring-boot	0.90	0.64	0.75	0.83	0.63	0.71
microservice	0.78	0.88	0.82	0.82	0.82	0.82
apache-spring-boot-microservice-example	**1.00**	0.89	**0.94**	0.63	0.71	0.67
spring-cloud-movie-recommendation	0.80	0.73	0.76	0.83	0.59	0.69
spring-petclinic-microservices	**1.00**	0.76	0.87	**0.94**	0.83	**0.88**
piggymetrics	0.89	**0.91**	0.90	0.69	**0.85**	0.76

Table 2: Comparison results and metrics for the accuracy of the *Retriever* approach in generating both structural and behavioral properties.

Each of the five steps is performed individually on a subset of the previously evaluated case studies. We then compare the manual model with the model from our automated reverse engineering approach and categorize the comparison result to answer the questions related to the second goal of the evaluation plan. This subset allows for comparison with manually constructed architectural models, evaluates the accuracy of the automated approach, and reflects the variability and diversity of the case studies. The goal is to assess the accuracy of the reverse engineering approach and to identify potential inaccuracies and shortcomings in the reverse engineering results. The method of comparison involves contrasting the structural and behavioral properties of the manually analyzed component architecture with the corresponding reverse-engineered component architecture. The following elements are compared in the models: The boundaries of components and their groupings within components and nodes. Interfaces provided and required by components, along with their corresponding uniform resource identifiers (URIs). Relationships of calls between components via external calls, as specified in the SEFF.

The metrics used for measurement include precision p, recall r, and F_1 score. True positives are instances where the model elements identified in the manual analysis are also detected in the reverse engineering results. False positives are instances where model elements are discovered in reverse engineering but not in the manual analysis. False negatives are instances where model elements are found in the manual analysis but not in the reverse engineering. After categorizing the comparison results, the metrics are calculated for each case. These metrics can be used to evaluate the accuracy of the *Retriever* approach. Table 2 shows the detailed comparison results and metrics for each case.

5.3. Results and Discussion

Because reverse engineering large code bases results in large models, scalability becomes a critical factor. Our approach has already demonstrated its effectiveness in real-world projects ranging in size from small to large. As expected, the average execution time increases with the size of the input code base. However, only a linear increase is observed for the case studies. Furthermore, the execution time for the largest case study with over 559143 LOC is still very reasonable at 134.416 seconds. However,

additional efforts are needed to improve the performance for larger systems. To this end, we intend to explore various techniques, such as parallelizing model transformations. In future work to reduce execution time, transformation rules will be implemented concurrently whenever possible. The ratio of source files to the number of components in the final architectural model serves as an indicator of the reduction quality of the reverse-engineered architectural models. The graph indicates that the approach abstracts from the existing files and does not represent them 1:1, indicating that the approach produces valid architectural models that abstract from the source code.

The second goal was to evaluate the accuracy of the approach in generating both structural and behavioral views. The results in table 2 indicate that the approach is capable of recognizing most of the primary components of a software system and their interrelationships. For example, F_1 values for structural properties range from 0.75 to 0.94. In addition, the method can effectively analyze the internal structure of a component by identifying its subcomponents and their relationships. This is illustrated by the F_1 values of the behavioral attributes, which range from 0.65 to 0.88. However, the approach falls short in identifying components that do not directly contribute to the functions of the system domain, such as configuration-related components. At the same time, our approach often results in inadequate analysis of component interfaces, leading to incorrect or redundant paths and incorrect mappings between interfaces and components. These results suggest that the accuracy of the approach is primarily dependent on the initial phase of the extraction transformation, which converts artifact models into architectural views. Since the evaluation used only generic technology-specific rules for all extraction transformations, these rules significantly influence the result. The results indicate that frameworks such as Spring provide a better basis for technology-specific extraction rules than frameworks that enforce less stringent implementation requirements, such as JaxRS. A suboptimal result from the *Retriever* approach could signal a poorly structured system that lacks patterns. To remedy this, architects can formulate new project-specific rules. In a previous work, we have shown that the result of the extraction can be improved by project-specific rules [47].

5.4. Threats to Validity

We have structured our threats to validity based on the guidelines for case study research from Runeson and Höst [48].

Internal Validity: The evaluation aims to accurately capture the architecture of each reference system by analyzing source code, configuration files, and expert knowledge. The evaluation compares the constructed architecture with existing documentation and architecture diagrams to ensure that only the expected components and interfaces with their data types are identified. However, the approach to identifying components and their interfaces provides only one possible view of the system.

External Validity: The evaluation focused on the specific reference systems and the performance of the *Retriever* approach in the context of web services. To avoid overfitting, only external case studies were used. However, the number of case studies is limited, and we restricted ourselves to a handful of technologies. Nevertheless, they are heterogeneous systems that are representative of the class of systems we are interested in. The case studies were chosen to cover a wide range of different technologies and to

be as diverse as possible. In the future, we plan to apply the approach to more external case studies and to add extraction rules for more technologies.

Construct Validity: In our approach, this is the relationship between metrics and goals. Using a specific evaluation approach such as GQM reduces the risk because it shows the relationship between goals and metrics. The evaluation goals and metrics should ensure that the analyzed elements contribute to the achievement of the research goals. The evaluation process involves comparing the expected architecture with the extracted architecture, measuring precision, recall, and F_1 score as metrics commonly used in related reverse engineering approaches.

Reliability: The text describes the steps taken to identify and extract the architecture, and presents the evaluation results in terms of LOC, execution time, precision, recall, and F_1 score. Uniform metrics are used for the result, avoiding subjective interpretations and increasing reproducibility. This information allows other researchers to understand and potentially replicate the evaluation process and results. However, evaluation results related to the reference architectures of the case studies used may depend on the experience of the researcher and may be subjective. For this reason, we are publishing a data set [49] that includes all of our reference architectures, software architecture models, and raw data, and the source code will be publicly available.

6. Related Work

The work most closely related to our approach relates to rule-based and model-driven reverse engineering as well as view-based development and is discussed in more detail in the following three sections. We differentiate three categories: approaches solving a similar problem with a different technique, applying a similar technique to a different problem, or addressing a similar problem with a similar technique.

6.1. Solves a Similar Problem Using a Different Technique

Garcia et al. note that the dominant method for automated architecture reconstruction is the clustering of software entities [50]. They present a novel machine learning-based technique for recovering an architectural view that includes a system's components and connectors [51]. However, this approach does not allow for the integration of existing knowledge about specific technologies and the resulting views, or the combination of technology views into a single architectural view. Tzerpos and Holt present ACCD [52], a communication-driven clustering algorithm that organizes files based on naming patterns. Andritsos et al. present LIMBO [53], a software clustering algorithm based on information theory that uses hierarchical clustering based on the similarities of groups of files to reverse engineer an architecture. Mitchell et al. present Bunch [54], a tool that generates decompositions using similarity measures and uses upscaling algorithms to cluster files based on coupling and cohesion. Jordan et al. present AutoArx [55], an approach that automates the retrieval, integration, and co-evolution of architectural data, facilitating the creation and maintenance of a digital twin that is persistently and autonomously updated as the system evolves. Despite their different principles, all of these reverse engineering methods divide source code units into mutually exclusive clusters based on a dominant principle such as cohesion and coupling or naming

patterns, using homogeneous source code as input. Garzón et al. propose a method for reverse engineering object-oriented code into a unified language for both object-oriented programming and modeling [56]. Cai et al. propose a new perspective on architecture reconstruction by assuming that most well-designed systems follow strict architectural design rules [57]. Klint et al. develop RASCAL, a domain-specific language that integrates source code analysis, transformation, and generation at the language level [58]. However, the UML diagrams produced by Garzón et al. are not suitable for quality prediction, and the design rules proposed by Cai et al. take the form of special program constructs that none of the submodules use. In addition, RASCAL does not support the generation of models with a different metamodel than the analyzed code.

Krogmann presented SoMoX [59], which clusters existing classes into components using a graph-based technique based on metrics such as cohesion and coupling. Langhammer presented Extract and EJBmoX [60], reverse engineering tools that extract source code behavior using hybrid analysis. Walter presented PMX [61], which automatically builds the prediction model from form measurement logs from the Kieker monitoring tool [62]. Mazkatli presented CIPM [63], which integrates parametric dependency estimation with Langhammer's hybrid extraction and extends the hybrid extraction approach to incrementally extract models for use in a continuous integration pipeline. However, all of these approaches are specific to a particular mapping between source code and architectural model and do not support the extraction of mixed technology software in a system.

6.2. Applies a Similar Technique to a Different Problem

Our research is closely related to view-based development methodologies. View-based modeling techniques [64, 23, 65] allow the definition of views on architectural models to manage their complexity. Query-based views such as *ModelJoin* [66], EMF views [67], and others [68] can be dynamically created and their view types quickly changed. Meier et al. [69] proposes a method for defining viewpoints and views using generic operators, while Atkinson and Tunjic [70] present a framework for multidimensional views in projective modeling. However, these methodologies require existing architectural models that instantiate MDE-compliant metamodels (typically EMF-based), and thus do not address automated reverse engineering of architectural models from heterogeneous artifacts such as code and build files. Single-underlying models [71] allow the reuse of existing models and their treatment as a single model, such as a detailed architectural model. This is typically achieved by transformation networks [72] or multiary transformations [73, 74]. For example, in the case of Vitruvius [71], model transformations are used to link all models into a network where changes are propagated transitively. In the latter case, more than two models are synchronized with a single model transformation. A third alternative is the merging of existing models [75]. While single-underlying models can reuse and merge existing architectural models, they rely on MDE-compliant (meta)models and do not support reverse engineering from heterogeneous artifacts. In contrast, our methodology is designed to automatically reverse engineer a comprehensive and MDE-compliant architectural model from various artifacts such as code, models, build and configuration files. It also supports predefined views of different parts of the architecture for different stakeholders.

6.3. Solves a Similar Problem Using a Similar Technique

Raibulet et al. compare in depth 15 different model-driven reverse engineering approaches in their literature review and find that both these approaches and their areas of application are versatile [40]. In this respect, MoDisco is the most related approach in a comprehensive scope. Bruneliere et al. developed MoDisco [76], a generic and extensible model-driven reverse engineering approach. Although MoDisco is extensible with technologies, it does not support direct reuse of common concepts of a technology. Furthermore, the combination of several technologies to generate a view is not supported. In their systematic literature review, Mushtaq et al. examined 56 out of 3820 papers to identify research gaps in the area of multilingual code analysis [77]. Of these 56 papers, MoDisco and three others come closest to our approach. Lehnert et al. present a tool [78] that uses UML diagrams to present different views, each representing different facets of an application at different levels of abstraction. Di Lucca et al. present WARE [79], a tool that uses UML diagrams to model different aspects of an application at different levels of abstraction. Tonella et al. propose a dynamic analysis method [80] to derive the model of an application through its execution. These three works focus on the domain of web applications and have challenges in handling large and complex web applications with complex dependencies between artifacts. Furthermore, they do not provide comparable quality prediction models. Ali et al. show in their empirical study of architectural consistency [10] that current tools such as Structure 101 and Lattix support multiple programming languages and extensibility. However, they lack standard support for heterogeneous systems. Structure 101 can abstract a system into a language-independent graph for analysis, while Lattix can analyze UML and SysML models, which is potentially useful for reverse engineering heterogeneous systems. Fradet et al. present a concept of views [81], represented as graphical representations with multiplicities, and propose a simple algorithm for verifying consistency. Despite its simplicity, the proposed technique is not suitable for all types of software architectures. The effectiveness of this method can vary greatly depending on the complexity and unique requirements of the software system. Ducasse et al. present Moose [82], a platform-independent environment for reverse engineering and reengineering complex software systems. However, the success of Moose depends on the specific requirements and constraints of the software system to be reengineered. Müller et al. present JQAssistant, an approach to create a uniform data source for software analysis and visualization [83]. However, only dependencies on the level of methods are considered so that there is no possibility to process fine granular information, which is needed for the creation of models for quality prediction.

7. Conclusion

In conclusion, we presented a novel approach, known as *Retriever*, to the challenge of reverse engineering software architecture models, particularly for Web service or microservice systems. The *Retriever* approach provides comprehensive answers to the two research questions from section 1.

To address RQ1, the *Retriever* approach demonstrates an effective method for reverse engineering software architecture from disparate artifacts across different technologies.

It does this by developing a knowledge representation model for different artifacts that includes model-to-model transformations that consider technology-specific relationships and concepts. This allows the approach to handle different formats and languages and translate them into a common representation for architectural modeling.

To address RQ2, the *Retriever* approach provides a framework for integrating individual views into a unified architectural model. This is achieved through model transformation and composition techniques that map concepts between views and incorporate them into a unified model based on an existing metamodel. This results in a unified model that addresses different concerns of a software system.

The evaluation of the approach demonstrates its effectiveness in real projects of different sizes and shows promising scalability and reduction quality. It is applicable to relevant software systems with different technologies and can generate an abstraction of the system as an architectural model. The approach is scalable and transferable to larger real systems, making it suitable for industrial applications. However, it also reveals areas for improvement, particularly in the identification of components that do not directly contribute to the functions of the system domain and in the analysis of component interfaces. The accuracy of the approach is found to depend on the initial phase of the extraction transformation, which converts artifact models into architectural views. The results indicate that technology-specific extraction rules significantly influence the result.

Overall, the *Retriever* approach represents a significant contribution to the fields of software architecture reverse engineering, model-driven software engineering, and component-based software engineering. It offers a promising solution to the challenges posed by the heterogeneity and complexity of modern software systems. However, further refinement and optimization are needed to improve its performance and accuracy, especially for larger systems, and to improve the accuracy of component and interface analysis. Future work will explore various techniques, such as parallelization of model transformations and concurrent implementation of transformation rules, to address these issues. There may also be a focus on extending the approach to handle more complex systems and integrating it with other software engineering tools and techniques. This ongoing research has great potential to advance our understanding and management of complex software systems.

8. Declaration of Generative AI and AI-assisted technologies in the writing process

During the preparation of this work, the authors used Grammarly[1], LanguageTool[2], and DeepL[3] Translator and Write in order to recommend and insert replacement text, perform spelling or grammar checks and corrections, or do language translations. After using these tools/services, the authors reviewed and edited the content as needed and take full responsibility for the content of the publication.

[1] https://grammarly.com
[2] https://languagetool.org
[3] https://www.deepl.com

References

[1] S. Murer, B. Bonati, F. J. Furrer, Managed Evolution: A Strategy for Very Large Information Systems, 1st Edition, Springer Berlin Heidelberg, 2011.

[2] G. Canfora, M. Di Penta, New frontiers of reverse engineering, in: 2007 Future of Software Engineering, FOSE '07, IEEE Computer Society, 2007.

[3] R. Kazman, L. Bass, G. Abowd, M. Webb, Saam: a method for analyzing the properties of software architectures, in: Proceedings of 16th International Conference on Software Engineering, 1994.

[4] M. Lungu, Towards reverse engineering software ecosystems, in: 2008 IEEE International Conference on Software Maintenance, 2008.

[5] P. Selfridge, R. Waters, E. Chikofsky, Challenges to the field of reverse engineering, in: Proceedings Working Conference on Reverse Engineering, 1993.

[6] W. C. Chu, C.-W. Lu, C.-H. Chang, Y.-C. Chung, X. Liu, H. Yang, Reverse engineering, in: Handbook of Software Engineering and Knowledge Engineering: Volume II: Emerging Technologies, World Scientific, 2002.

[7] J. Bogner, J. Fritzsch, S. Wagner, A. Zimmermann, Microservices in industry: Insights into technologies, characteristics, and software quality, in: 2019 IEEE International Conference on Software Architecture Companion (ICSA-C), 2019.

[8] R.-H. Pfeiffer, What constitutes software? an empirical, descriptive study of artifacts, in: 17th International Conference on Mining Software Repositories, MSR '20, ACM, 2020.

[9] W. Hasselbring, Software architecture: Past, present, future, The Essence of Software Engineering.

[10] N. Ali, S. Baker, R. O'Crowley, S. Herold, J. Buckley, Architecture consistency: State of the practice, challenges and requirements, Empirical Software Engineering.

[11] S. Kent, Model driven engineering, in: Integrated Formal Methods, Springer Berlin Heidelberg, 2002.

[12] H. Stachowiak, Allgemeine Modelltheorie, 1973, Springer-Verlag, Wien.

[13] Object Management Group, Unified Modeling Language (UML), v2.5.1. URL https://www.omg.org/spec/UML/2.5.1

[14] M. Völter, t. Stahl, J. Bettin, A. Haase, S. Helsen, Model-driven software development: technology, engineering, management, John Wiley & Sons, 2013.

[15] D. Steinberg, F. Budinsky, E. Merks, M. Paternostro, EMF: eclipse modeling framework, Pearson Education, 2008.

[16] T. Mens, P. Van Gorp, A taxonomy of model transformation, Electronic notes in theoretical computer science.

[17] I. Kurtev, State of the art of QVT: A model transformation language standard, in: AGTIVE 2007: Applications of Graph Transformations with Industrial Relevance: third International Symposium, Springer, Springer Berlin Heidelberg, 2008.

[18] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, Atl: A model transformation tool, Science of computer programming.

[19] L. Bettini, Implementing domain-specific languages with Xtext and Xtend, Packt Publishing Ltd, 2016.

[20] R. Hebig, C. Seidl, t. Berger, J. K. Pedersen, A. Wąsowski, Model transformation languages under a magnifying glass: a controlled experiment with xtend, atl, and qvt, ACM, 2018.

[21] S. Götz, M. Tichy, R. Groner, Claimed advantages and disadvantages of (dedicated) model transformation languages: a systematic literature review, Software and Systems Modeling.

[22] J. W. Wittler, T. Sağlam, t. Kühn, Evaluating model differencing for the consistency preservation of state-based views, Journal of Object TechnologyThe 19th European Conference on Modelling Foundations and Applications (ECMFA 2023).

[23] A. Cicchetti, F. Ciccozzi, A. Pierantonio, Multi-view approaches for software and system modelling: a systematic literature review, Software & Systems Modeling.

[24] P. Kruchten, the 4+1 view model of architecture, IEEE Software.

[25] P. Kruchten, the rational unified process: an introduction, 3rd Edition, the @Addison-Wesley object technology series, Addison-Wesley Professional, 2004, literaturverz. S. 291 - 297.

[26] ISO Central Secretary, Software, systems and enterprise – architecture description, Standard, International Organization for Standardization (Nov. 2022).
URL https://www.iso.org/standard/74393.html

[27] C. Atkinson, D. Stoll, P. Bostan, Orthographic Software Modeling: A Practical Approach to View-Based Development, in: Evaluation of Novel Approaches to Software Engineering, Communications in Computer and Information Science, Springer, 2010.

[28] t. Goldschmidt, S. Becker, E. Burger, Towards a tool-oriented taxonomy of view-based modelling, in: Modellierung 2012, GI-Edition – Lecture Notes in Informatics (LNI), Gesellschaft für Informatik e.V. (GI), 2012.

[29] E. J. Burger, Flexible views for view-based model-driven development, in: 18th International Doctoral Symposium on Components and Architecture, WCOP '13, ACM, 2013.

[30] C. Szyperski, D. Gruntz, S. Murer, Component software: beyond object-oriented programming, Pearson Education, 2002.

[31] W. Hasselbring, Component-based software engineering, in: Handbook of Software Engineering and Knowledge Engineering: Volume II: Emerging Technologies, World Scientific, 2002.

[32] A. Roques, PlantUML (Nov. 2023).
URL https://plantuml.com

[33] S. Becker, H. Koziolek, R. Reussner, the palladio component model for model-driven performance prediction, Journal of Systems and Software.

[34] S. Becker, H. Koziolek, R. Reussner, Model-based performance prediction with the palladio component model, in: 6 international workshop on Software and performance, ACM, 2007.

[35] R. Kazman, S. Woods, S. Carriere, Requirements for integrating software architecture and reengineering models: Corum ii, in: Proceedings Fifth Working Conference on Reverse Engineering, IEEE Comput. Soc, 1998.

[36] G. Canfora, M. Di Penta, L. Cerulo, Achievements and challenges in software reverse engineering, Communications of the ACM.

[37] R. H. Reussner, S. Becker, J. Happe, A. Koziolek, H. Koziolek, Modeling and Simulating Software Architectures – the Palladio Approach, MIT Press, 2016.

[38] S. Rugaber, K. Stirewalt, Model-driven reverse engineering, IEEE Software.

[39] J.-M. Favre, Foundations of model (driven) (reverse) engineering : Models – episode i: Stories of the fidus papyrus and of the solarus, in: Language Engineering for Model-Driven Software Development, Dagstuhl Seminar Proceedings (DagSemProc), 2005.

[40] C. Raibulet, F. A. Fontana, M. Zanoni, Model-driven reverse engineering approaches: A systematic literature review, IEEE Access.

[41] C. Van Rijsbergen, C. Van Rijsbergen, Information Retrieval, Butterworths, 1979.

[42] Y. R. Kirschner, M. Walter, F. Bossert, R. Heinrich, A. Koziolek, Automatic derivation of vulnerability models for software architectures, in: 2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C), IEEE.

[43] Y. R. Kirschner, F. Bossert, M. Gstür, A. Koziolek, Retriever repository (2023).
URL https://github.com/PalladioSimulator/Palladio-ReverseEngineering-Retriever

[44] Stack Overflow, Stack Overflow Developer Survey 2023 (2023).
URL https://survey.stackoverflow.co/2023

[45] fortiss, CCE Trends (2023).
URL https://cce.fortiss.org/trends

[46] G. Basili, V. R. Caldiera, H. D. Rombach, the goal question metric approach, Encyclopedia of software engineering.

[47] Y. R. Kirschner, J. Keim, N. Peter, A. Koziolek, Automated reverse engineering of the technology-induced software system structure, in: Software Architecture, 1st Edition, Lecture Notes in Computer Science, Springer Nature Switzerland.

[48] P. Runeson, M. Höst, Guidelines for conducting and reporting case study research in software engineering, Empirical Software Engineering.

[49] Y. Kirschner, G. Moritz, T. Sağlam, S. Weber, A. Koziolek, Dataset and replication package for the view-based retriever approach to reverse engineering software architecture models (2023). doi:10.5281/zenodo.10442265.

[50] J. Garcia, I. Ivkovic, N. Medvidovic, A comparative analysis of software architecture recovery techniques, in: ASE'13, IEEE, 2013.

[51] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, Y. Cai, Enhancing architectural recovery using concerns, in: ASE'11, IEEE, 2011.

[52] V. Tzerpos, R. C. Holt, Accd: an algorithm for comprehension-driven clustering, in: WCRE'00, IEEE, 2000.

[53] P. Andritsos, P. Tsaparas, R. J. Miller, K. C. Sevcik, Limbo: Scalable clustering of categorical data, in: EDBT'04, Springer, 2004.

[54] B. S. Mitchell, S. Mancoridis, On the automatic modularization of software systems using the bunch tool, IEEE Transactions on Software Engineering 32 (3).

[55] S. Jordan, L. Linsbauer, I. Schaefer, Autoarx: Digital twins of living architectures, in: European Conference on Software Architecture, Springer, 2022.

[56] M. A. Garzón, T. C. Lethbridge, H. I. Aljamaan, O. Badreddin, Reverse engineering of object-oriented code into umple using an incremental and rule-based approach., in: CASCON'14, IBM / ACM, 2014.

[57] Y. Cai, H. Wang, S. Wong, L. Wang, Leveraging design rules to improve software architecture recovery, in: 9 International ACM Sigsoft Conference on Quality of Software Architectures, QoSA '13, 2013.

[58] P. Klint, T. van der Storm, J. Vinju, Rascal: A domain specific language for source code analysis and manipulation, in: ICSME'09, IEEE, 2009.

[59] K. Krogmann, M. Kuperberg, R. Reussner, Using genetic search for reverse engineering of parametric behavior models for performance prediction, IEEE TSE.

[60] M. Langhammer, A. Shahbazian, N. Medvidovic, R. H. Reussner, Automated extraction of rich software models from limited system information, in: 2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA), IEEE, 2016.

[61] J. Walter, C. Stier, H. Koziolek, S. Kounev, An expandable extraction framework for architectural performance models, in: 8 ACM/SPEC on International Conference on Performance Engineering Companion, ACM, 2017.

[62] A. Van Hoorn, J. Waller, W. Hasselbring, Kieker: A framework for application performance monitoring and dynamic software analysis, in: 3 ACM/SPEC international conference on performance engineering, ACM, 2012.

[63] D. Monschein, M. Mazkatli, R. Heinrich, A. Koziolek, Enabling consistency between software artefacts for software adaption and evolution, in: 2021 IEEE 18th International Conference on Software Architecture (ICSA), IEEE, 2021.

[64] H. Brunelière, E. Burger, J. Cabot, M. Wimmer, A feature-based survey of model view approaches, in: 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS '18, 2018.

[65] J. Meier, C. Werner, H. Klare, C. Tunjic, U. Aßmann, C. Atkinson, E. Burger, R. Reussner, A. Winter, Classifying approaches for constructing single underlying models, in: Model-Driven Engineering and Software Development, Springer International Publishing, 2020.

[66] E. Burger, J. Henß, M. Küster, S. Kruse, L. Happe, View-Based Model-Driven Software Development with ModelJoin, Software & Systems Modeling.

[67] H. Brunelière, J. G. Perez, M. Wimmer, J. Cabot, EMF views: A view mechanism for integrating heterogeneous models, in: 34th International Conference on Conceptual Modeling (ER 2015), Springer International Publishing, 2015.

[68] C. Werner, M. Wimmer, U. Assmann, A generic language for query and viewtype generation by-example, in: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion, IEEE, 2019.

[69] J. Meier, R. Kateule, A. Winter, Operator-based viewpoint definition, in: 8th International Conference on Model-Driven Engineering and Software Development - MODELSWARD,, INSTICC, SciTePress, 2020.

[70] C. Atkinson, C. Tunjic, A deep view-point language for projective modeling, Information Systems.

[71] H. Klare, M. E. Kramer, M. Langhammer, D. Werle, E. Burger, R. Reussner, Enabling consistency in view-based system development — the vitruvius approach, Journal of Systems and Software.

[72] P. Stevens, Maintaining consistency in networks of models: bidirectional transformations in the large, Software and Systems Modeling.

[73] A. Cleve, E. Kindler, P. Stevens, V. Zaytsev, Multidirectional Transformations and Synchronisations (Dagstuhl Seminar 18491), Dagstuhl Reports.

[74] G. Bergmann, Controllable and decomposable multidirectional synchronizations, Software and Systems Modeling.

[75] G. Brunet, M. Chechik, S. Easterbrook, S. Nejati, N. Niu, M. Sabetzadeh, A manifesto for model merging, in: 2006 International Workshop on Global Integrated Model Management, GaMMa '06, ACM, 2006.

[76] H. Bruneliere, J. Cabot, G. Dupé, F. Madiot, Modisco: A model driven reverse engineering framework, Information and Software Technology 56 (8).

[77] Z. Mushtaq, G. Rasool, B. Shehzad, Multilingual source code analysis: A systematic literature review, IEEE Access.

[78] S. Lehnert, Q.-u.-a. Farooq, M. Riebisch, Rule-based impact analysis for heterogeneous software artifacts, in: 2013 17th European Conference on Software Maintenance and Reengineering, IEEE, 2013.

[79] G. Di Lucca, A. Fasolino, F. Pace, P. Tramontana, U. De Carlini, Ware: a tool for the reverse engineering of web applications, in: Sixth European Conference on Software Maintenance and Reengineering, IEEE, 2002.

[80] P. Tonella, F. Ricca, Dynamic model extraction and statistical analysis of web applications, in: Proceedings. Fourth International Workshop on Web Site Evolution, WSE-02, IEEE Comput. Soc, 2002.

[81] P. Fradet, D. Le Métayer, M. Périn, Consistency checking for multiple view software architectures, SIGSOFT Softw. Eng. Notes.

[82] S. Ducasse, T. Gîrba, O. Nierstrasz, Moose: An agile reengineering environment, in: 10th European Software Engineering Conference Jointly with 13th International Symposium on Foundations of Software Engineering, 2005.

[83] R. Müller, D. Mahler, M. Hunger, J. Nerche, M. Harrer, Towards an open source stack to create a unified data source for software analysis and visualization, in: VISSOFT'18, IEEE, 2018.