Extracting API Structures from Documentation to Create Virtual Knowledge Graphs

Maximilian Weigand[1][a], Felix Gehlhoff[1][b] and Alexander Fay[2][c]
[1]*Institute of Automation Technology, Helmut-Schmidt-University / University of the Federal Armed Forces Hamburg, Germany*
[2]*Chair of Automation Technology, Ruhr University Bochum, Germany*

Keywords: Application Programming Interface (API), Virtual Knowledge Graph, Ontology, Software, Documentation.

Abstract: Semantic Web technologies and standards have emerged as effective solutions for data exchange, also in engineering contexts. They provide a standardized way to exchange data between different software and facilitate interoperability. Within this work, we introduce a workflow to systematically analyze the structure of application programming interfaces (APIs) of software, enabling the efficient transformation of information available from the API into information models that are structured according to Semantic Web standards. Our goal is to create a reusable interface for engineering software on top of its API. The approach leverages shared concepts between object-oriented programming and knowledge graphs to abstract components of the API into a knowledge graph. The workflow allows to selectively extract relevant API components and automates the generation of necessary code. To demonstrate the approach, we created an application that implements the workflow and use it for a Java-based API for a modeling software, showcasing the reduction of manual effort.

1 INTRODUCTION

In modern engineering, engineers rely on a wide range of software applications. However, data exchange between these applications often poses a challenge due to limited interoperability.

Many of these software applications offer application programming interfaces (APIs), which make the software more open by providing a systematic and versatile means to extract information (Fay et al., 2013). However, implementing such functionality is often time-consuming and tends to be a specialized, non-reusable solution.

Ontologies and knowledge graphs (KGs), originating from the Semantic Web, have emerged as effective means for accurate information modeling and efficient data exchange in engineering fields, e.g. automation engineering (Dotoli et al., 2018; Ekaputra et al., 2017; Sabou et al., 2020). Their standardized formats make them ideal for defining and exchanging technical information.

In a previous publication (Weigand and Fay, 2022), we addressed the challenge of data exchange and interoperability between various software applications in engineering. We proposed an approach leveraging Semantic Web technologies to enhance data exchange by abstracting data available from the API of a software into a KG.

In recent years, the concept of Digital Twins (DTs) has gained significant attention in engineering. A DT is essentially a digital representation of a physical asset. Ontologies and KGs have been discussed as potential solutions for DTs, as one of the main challenges remains creating interoperable and semantically unambiguous DTs from various data sources (Vogel-Heuser et al., 2021). Our approach can therefore support the creation of DTs in engineering by providing a structured method for extracting and representing data from engineering software in a standardized and reusable way.

In our previous publication (Weigand and Fay, 2022), the interface on top APIs of engineering software was described generically, and was adapted for an exemplary software, to demonstrate its functionality. The adaption requires tasks like specifying a set of relevant components of the API, i.e. components that represent information that should be abstracted into the KG. As the adaption proved to be quite labor-

[a] https://orcid.org/0009-0000-1602-4873
[b] https://orcid.org/0000-0002-8383-5323
[c] https://orcid.org/0000-0002-1922-654X

Weigand, M., Gehlhoff, F. and Fay, A.
Extracting API Structures from Documentation to Create Virtual Knowledge Graphs.
DOI: 10.5220/0013083100003838
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 16th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K 2024) - Volume 2: KEOD*, pages 287-294
ISBN: 978-989-758-716-0; ISSN: 2184-3228
Proceedings Copyright © 2024 by SCITEPRESS – Science and Technology Publications, Lda.

intensive, we will show within this publication how this process can be structurized into a workflow and can be implemented in a supported manner that is automated as much as possible.

In Section 2, we will evaluate approaches that utilize the shared concepts of APIs and KGs by integrating them. Section 3 will introduce requirements for the proposed workflow, which will be presented in detail in Section 4. Section 5 will demonstrate several steps of the workflow through an implementation, which will be showcased by using it to analyze the API of an exemplary systems engineering software. Finally, we will summarize insights from the demonstrated use case in Section 6 and assess it.

2 FUNDAMENTALS AND RELATED WORK

2.1 Fundamentals

An API generally serves as a mechanism for a client software component to interact with a supplier software component by offering key functions of the supplier software while concealing their underlying implementation details (Reddy, 2011). In the simplest form, this may be providing a subset of the supplier software's source code, typically in the form of a library, and is therefore often called a *library API*. The term *API* is currently more commonly used to refer to *web APIs*. In the context of this publication, the term *API* specifically refers to library APIs of engineering software, such as systems engineering or mechanical design software. These software often provide APIs to enable developers to extend the software's functionality. Our focus is on APIs within the object-oriented programming (OOP) paradigm, as it is the prevailing approach in software development.

A KG is a graph-based data structure intended to represent real-world knowledge by representing entities of that knowledge as nodes and relations in between entities as edges (Hogan et al., 2021). The Resource Description Framework (RDF) (Cyganiak et al., 2014) is a standard model for representing data on the Semnatic Web using subject-predicate-object triples. RDF Schema (RDFS) (Brickley and Guha, 2014) extends RDF by providing a basic vocabulary for describing relationships and hierarchies, while the Web Ontology Language (OWL) (Hitzler et al., 2012) builds on RDF and RDFS to enable more complex semantic relationships within the knowledge. Typically, a KG is an RDF graph using RDFS or OWL vocabulary. Within this publication, the term *KG* specifically refers to RDF graphs using RDFS vocabulary.

2.2 Shared Concepts of Knowledge Graphs and Object-Oriented Data

KGs and data models within the OOP paradigm share several foundational concepts, primarily focused on representing information. Examples are classes in OOP and RDFS (`rdfs:Class`), objects in OOP and resources in RDFS (`rdfs:Resource`) or methods in OOP and properties in RDF (`rdf:Property`).

Within a previous publication (Weigand and Fay, 2022), we created an interface that utilizes these shared concepts to abstract the data available through an API of a software as a KG. The KG resulting from this abstraction retains the advantages of KGs, e.g., it can be queried using SPARQL (Harris and Seaborne, 2013), a standardized query language to retrieve information from KGs systematically. The meta data, e.g. the class structure of the API, is also contained in the KG, making it possible to query what data is available from the API. In our approach, the KG is not materialized, i.e. access to the API data and the abstraction into a KG occur only at the time of the query, making it a virtual KG (VKG). In summary, the VKG proposed in our previous publication abstracts:

- Classes of the API are abstracted as `rdfs:Class`
- Class hierarchies are abstracted as `rdfs:subClassOf` properties
- The definition of public methods with no arguments (*get-methods*) are abstracted as `rdf:Property`, including associated `rdfs:domain` and `rdfs:range` properties
- Instances of classes are abstracted as `rdfs:Resource`, including a `rdf:type` property to the associated class and superclasses
- Instances that are returned if get-methods of other instances are executed are abstracted as a `rdf:Property` to a `rdfs:Resource` or to literals

In the previous publication, the concept of the VKG abstraction was explained, and we introduced a generic version of the VKG interface. This generic version includes key components such as the logic to execute a query over the VKG and is intended to be adapted to a specific software. While some adaptions must be done manually, others can be largely automated, though they were implemented manually in the initial demonstration in our previous publication. For instance, as shown earlier, each class of the API is abstracted as a class in the KG. Our goal within this work is to develop a structured approach to automate the extraction of such information from the API to reduce manual implementation effort.

2.3 Related Work

The shared concepts of OOP and KGs have been leveraged by other approaches, which will be summarized in this section.

Object Triple Mapping (OTM) libraries integrate KGs into the data models of OOP applications. There are very mature OTM solutions, such as *Alibaba* which offers a range of features and good performance (Ledvinka and Křemen, 2020). To use OTM libraries, a mapping between the OOP data model and the KG has to be defined within the source code of the application, for example by using code annotations. It then can be used to store the application data in the form of a KG in a triple store. This allows the application to store data persistently, while also enabling other applications or end-users to access it independently. OTMs are deeply integrated into the application and must therefore be implemented during the development of the application.

If parts of the source code are intended for publication as an API, the code is typically documented. A popular built-in tool of Java Development Kits is *Javadoc*. Javadoc converts parts of the code including information given in comments into documentation, typically in HTML format, though the format can be chosen by selecting a specific module called *doclet*. Similar tools exist for other OOP languages, as well as multi-language tools like *Doxygen*[1].

The doclet *ontlet* leverages this principle to extract the semantics of a Java library as an OWL ontology. This ontology facilitates understanding the semantic similarities between different libraries, aiding in code migration efforts. While the approach does not require modifying the source code, it does necessitate the source code to be available. (Ancona et al., 2012)

An alternative approach to create an ontology for a Java project is parsing it to generate an Abstract Syntax Tree (AST) and then converting the AST into RDF triples (Atzeni and Atzori, 2017). Unlike the previous approach, this approach can handle Java bytecode, meaning it accepts compiled code as input.

If an API to extend a software exists, the API documentation is usually available. Although API documentation is primarily created to be human-readable, it is machine-readable due to its standardized structure. Consequently, a viable solution is using a mapping language like the RDF Mapping Language (RML) (De Meester et al., 2024), which maps various structured data sources (like CSV or JSON) into RDF formatted data. Using this rule-based approach, the complete API structure can be mapped.

[1]www.doxygen.nl

3 REQUIREMENTS

The following requirements outline considerations and constraints for the steps of a structured approach to support and automate creating a VKG interface.

R1 (Brownfield Development Context): As outlined in the state of the art, various OTM approaches exist that enable mapping an OOP data model within the source code of an API to a KG. These approaches are well-established, having been the subject of extensive previous research, and there are multiple mature implementations available. However, these existing approaches are generally designed for greenfield software development processes, where the source code is under development, i.e. it is accessible and modifiable. In contrast, the proposed approach (Weigand and Fay, 2022), is tailored for use with an existing library API, placing it in a brownfield software development context. In this context, the source code is not accessible and cannot be modified. Only a subset of components, specifically the API, is usable, but also not accessible as source code. It is however documented, for example with tools like Javadoc. R1 restricts the input of the developed workflow to be a structured documentation of the API.

R2 (Selective Extraction of Relevant Components): As the API documentation is a structured data source, it can be transformed fully automatically using a rule-based approach. For example, by mapping each class of the API to an `rdfs:Class`. However, given that the API likely includes an extensive amount of components, including some that represent irrelevant information, it is crucial to focus on extracting only specific components that are relevant for exchange via the VKG interface. In the context of engineering, e.g. mechanical design, a relevant class of the API might be one that represents geometrical features of a part, while an irrelevant class might be one that handles notifications of the user interface of the software. As the relevance of API components cannot be determined automatically, the selection of relevant API components must be carried out manually. However, selection steps in the workflow should be supported and facilitated as much as possible.

R3 (Automation): In contrast to the steps that must be performed manually due to R2, the remainder of the extraction process should be automated to the greatest extent possible. Automation serves two key purposes: first, to reduce implementation effort, such as automatically establishing subclass relationships when the developer adds a class and its subclass to the set of relevant classes; and second, to mitigate potential errors, such as verifying that when a developer adds a method, the class returned by the method is within the set of relevant classes.

R4 (Usability): During the manual or guided steps of the workflow, the developer should be able to work within a familiar environment. This necessitates reusing the graphical representation of the documentation, with minimal modifications to its appearance. Modifications should only introduce the required functions, such as the ability to add a class to the set of relevant classes.

4 DEVELOPED WORKFLOW

The workflow described within this section is a systematic approach to develop a VKG interface for an API. Its steps are displayed in Figure 1 using BPMN 2.0 (Object Management Group, 2011) and will be explained in detail subsequently.

S1 (Analyze Class Structure): In this initial step, the complete class structure of the API is analyzed. The input for this step is the structured documentation of the API (R1). Since it is structured, this step can be fully automated and is modeled as a BPMN `Service Task`. A comprehensive analysis of the inheritance structure of all classes described in the API documentation is performed. The output of this step is a list of all classes, each with references to its superclasses. Inheritance relations are crucial information, are part of the VKG and will be introduced in S2.2.

S2 (Add Components from API Documentation): In this step, the developer will be assisted in selecting relevant classes and methods of the API. The entire step is a BPMN `Loop Sub-Process`, meaning the contained sub-steps (S2.1 to S2.4) can be repeated multiple times as needed.

S2.1 (Select Relevant Classes): In this step, the developer selects relevant API classes from the entire class structure. The relevance of a class is determined based on whether it describes information that should be exchangeable via the new VKG interface. As detailed in R2, the relevance of a class must be determined by the developer, therefore the developer is assisted by a tool during this step, which is modeled as a BPMN `User Task`. According to R4, the tool reuses the graphical representation of the API documentation, e.g. the HTML Javadoc of a Java API. Only minimal modifications are made to introduce controls for adding classes to the set of relevant classes.

S2.2 (Add Superclass Relation): In this step, superclass relations are automatically added. A superclass relationship originates from one class and points to its superclass. Subclass relationships are not explicitly extracted, as they can be inferred from existing superclass relations. Superclass relations are added when a class is added and a superclass or subclass of that class has already been added. If the relationship spans multiple classes, and the intermediate classes are not added, the relationship is created directly between the added classes. These indirect superclass relationships are corrected if an intermediate class is added later. The entire class hierarchy, extracted in Step 1, serves as the input for this process, which is executed each time a class is added. Consequently, this step is fully automatable (R3) and is modeled as a BPMN `Service Task`.

S2.3 (Select Relevant Methods): Similar to classes in S2.1, public methods of classes are now selected and added by the developer. The selection is restricted to methods without arguments as they can be mapped directly to an RDF triple. A triple represents a directed relationship between two entities and is the fundamental data model of an RDF graph. If a method has arguments, it would imply a parametrization of the relationship, which cannot easily be represented in RDF. Therefore, methods with arguments cannot be selected. As with S2.1 and in accordance with R2, the developer must determine which methods represent relevant relationships. Therefore, this step is also modeled as a BPMN `User Task`. According to R4, the functionality for adding a method must also be integrated into the existing graphical representation of the API documentation.

S2.4 (Add Missing Classes): If the class of the instance that the method added in the previous step returns is not within the set of selected classes, it will be added automatically in this step. This step is based on information available from previous steps and will be executed fully automatically each time a method is added (R3). Consequently, this step is modeled as a BPMN `Service Task`.

S3 (TBox Generation): Once S2 has been completed, the TBox of the VKG is fully defined. Although the TBox can later be queried via the VKG interface, an early export at this stage is possible and can be performed optionally. The TBox can be exported in standard formats such as RDF/XML. The export process relies entirely on the information defined in the previous steps. While the developer decides whether to initiate the export, the remainder of this step is executed automatically. Consequently, this step is modeled as a BPMN `Service Task`.

S4 (Code Generation): In this step, parts of the code for the VKG interface are generated. The VKG interface operates using two lists. One list includes all classes, which is primarily created in S2.1. Each class specifies its direct superclasses as established in S2.2. The other list contains methods, as specified in S2.3. Each method indicates the class it originates from and

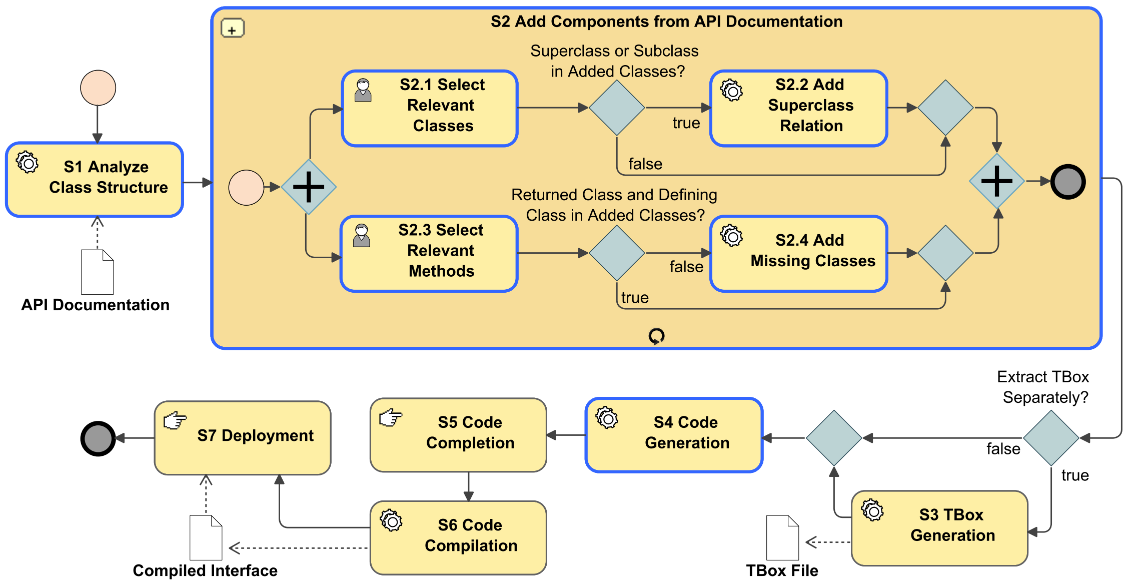

S2 Add Components from API Documentation
Superclass or Subclass in Added Classes?
S2.1 Select Relevant Classes
S2.2 Add Superclass Relation
true
false
Returned Class and Defining Class in Added Classes?
S2.3 Select Relevant Methods
S2.4 Add Missing Classes
false
true
S1 Analyze Class Structure
API Documentation
Extract TBox Separately?
false
true
S3 TBox Generation
TBox File
S4 Code Generation
S5 Code Completion
S6 Code Compilation
S7 Deployment
Compiled Interface

Figure 1: Steps of the developed workflow.

the class or the type of literal it returns. This information is transformed into a specific code format, as defined by the implementation of the generic version of the VKG interface, which is publicly available on GitHub[2]. Since the required information is provided by the previous steps and the existing code, this step is fully automated and is therefore modeled as a BPMN `Service Task`.

S5 (Code Completion): In this step, the developer completes parts of the code that cannot be automatically generated. Completion involves creating implementations of abstract classes defined by the generic version of the VKG interface. The required completions, such as the functionality to identify the class of an instance, are therefore specified by the generic version of the VKG interface. However, the implementation of these completions is highly dependent on the specific software being used. As a result, this step cannot be automated or supported by a tool and relies entirely on the developer's input. Therefore, it is modeled as a BPMN `Manual Task`.

S6 (Code Compilation): The code now consists of the generic version of the VKG interface, the automatically generated code from S4, and the manually added code from S5. The compilation is performed automatically by a compiler within this step, which is modeled as a BPMN `Service Task`.

S7 (Deployment): This final step involves deploying the compiled VKG interface. This may include tasks such as copying the VKG interface to a dedicated add-on folder or creating additional descriptive files that enable the software to recognize the VKG interface as an add-on. The specific actions required are highly dependent on the software in question. It is therefore modelled as a BPMN `Manual Task`.

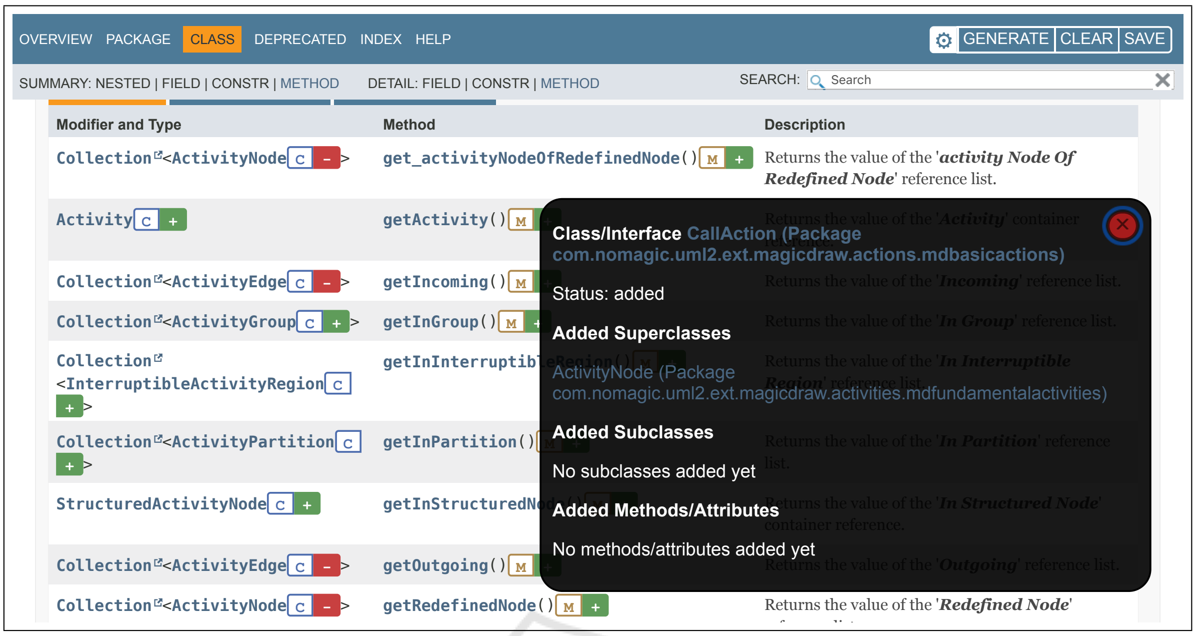
5 IMPLEMENTATION

To demonstrate our approach, we implemented a tool that automates and supports several steps of the developed workflow. This includes steps S1, S2, and S4, as well as all sub-steps of step S2 (S2.1 to S2.4). Step S3 was omitted as it is optional. Step S5 was omitted because it involves manual input and is intended to be completed by the developer working directly on the source code of the VKG interface. Step S6, which is fully automated and handled by a compiler, was also excluded, as we did not integrate the compiler into our tool. Step S7 was also omitted because it is a manual task. Blue borders in Figure 1 indicate the implemented steps, while grey borders represent the steps that were not implemented.

The tool is built around the HTML API documentation of a Java API, which was generated by Javadoc. Our implementation utilizes NestJS[3], a Node.js[4] framework typically used to build server-side applications, particularly for the backends of web applications, using TypeScript. NestJS can also be used to serve HTML documents as a frontend, in our case the Javadoc HTML documents of the API documentation. Before serving each document, the tool in-

[2]github.com/mxweigand/vmax_core

[3]docs.nestjs.com
[4]nodejs.org


OVERVIEW
PACKAGE
CLASS
DEPRECATED
INDEX
HELP
GENERATE
CLEAR
SAVE
SUMMARY: NESTED | FIELD | CONSTR | METHOD
DETAIL: FIELD | CONSTR | METHOD
SEARCH:
Search

Modifier and Type	Method	Description
Collection<ActivityNode C –>	get_activityNodeOfRedefinedNode() M +	Returns the value of the '***activity Node Of Redefined Node***' reference list.
Activity C +	getActivity() M +	Returns the value of the '***Activity***' container reference.
Collection<ActivityEdge C –>	getIncoming() M +	Returns the value of the '***Incoming***' reference list.
Collection<ActivityGroup C +>	getInGroup() M +	Returns the value of the '***In Group***' reference list.
Collection<InterruptibleActivityRegion C +>	getInInterruptibleRegion() M +	Returns the value of the '***In Interruptible Region***' reference list.
Collection<ActivityPartition C +>	getInPartition() M +	Returns the value of the '***In Partition***' reference list.
StructuredActivityNode C +	getInStructuredNode() M +	Returns the value of the '***In Structured Node***' container reference.
Collection<ActivityEdge C –>	getOutgoing() M +	Returns the value of the '***Outgoing***' reference list.
Collection<ActivityNode C –>	getRedefinedNode() M +	Returns the value of the '***Redefined Node***'

Class/Interface CallAction (Package com.nomagic.uml2.ext.magicdraw.actions.mdbasicactions)

Status: added

Added Superclasses

ActivityNode (Package com.nomagic.uml2.ext.magicdraw.activities.mdfundamentalactivities)

Added Subclasses

No subclasses added yet

Added Methods/Attributes

No methods/attributes added yet

Figure 2: Method section of the Javadoc API documentation of a class with additional control elements.

tercepts and modifies it by injecting additional HTML components and JavaScript functions. These modifications enhance the document with interactive elements needed for the workflow. After these changes, the document is sent to the web browser, which displays it with the new components integrated. In addition to the frontend, which serves the modified HTML documents, we also set up a backend. The backend handles tasks such as executing S1, which involves analyzing the entire API documentation, or, when the developer selects a class in S2.1, adding the class to a class list and automatically executing subsequent steps of the workflow, such as S2.2.

For the demonstration, we used an example software for which we had already developed and tested the VKG interface in a previous publication (Weigand and Fay, 2022). In that earlier work, the interface was created manually, which proved to be a time-consuming process. The software used in the prior study was Cameo Systems Modeler by Dassault Systèmes. In this demonstration, we utilized a very similar software version: Magic Systems of Systems Architect (MSOSA). MSOSA is a multi-purpose modeling software that supports various modeling languages, including SysML. This software provides a suitable environment to showcase our tool's ability to streamline the VKG interface creation process, improving efficiency compared to the fully manual approach previously employed. MSOSA features a Java API that allows for modification of the software's functionality. This API can also be leveraged for our purpose: extracting engineering information.

The API is documented using Javadoc, which forms the basis for our tool. We developed a prototypical tool[5] that implements the previously outlined steps (S1, S2 and S4), using the Javadoc API documentation of MSOSA to automate parts of the process and simplify the extraction of relevant information.

Pages of Javadoc documentation typically display information about a particular class, including its public methods. Figure 2 shows an exemplary page of the API documentation of MSOSA which was modified by our tool. The original API documentation is publicly available online[6]. At the top right of the page, within the navigation bar, general tools are available, e.g. to generate code for the interface based on the currently added API components (*GENERATE*). After each link that leads to a page detailing another class, two buttons are added. The first button, marked with *C*, indicates that the link directs to a class. Clicking this button opens an informational dialog, as illustrated on the right side of Figure 2. The second button (+ or -) allows developers to add or remove the class from the list of relevant classes. After each link leading to a method of the class, a similar button, marked with *M* is added. On the right side of Figure 2 the informational dialog that appears when a developer clicks on the class information button is shown. It shows whether the class has been added to the list, along with all subclasses and superclasses, including implicit ones. Additionally, the dialog displays all

[5]github.com/mxweigand/vmax_api_doc_browser
[6]jdocs.nomagic.com/2024x

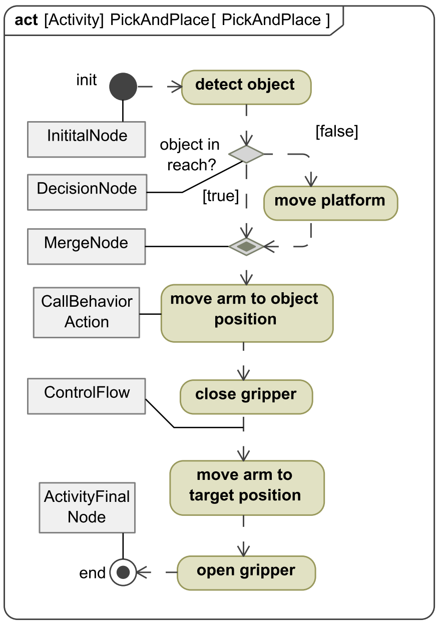


Figure 3: Example robotic process modeled in a SysML Activity Diagram (element types indicated in grey boxes).

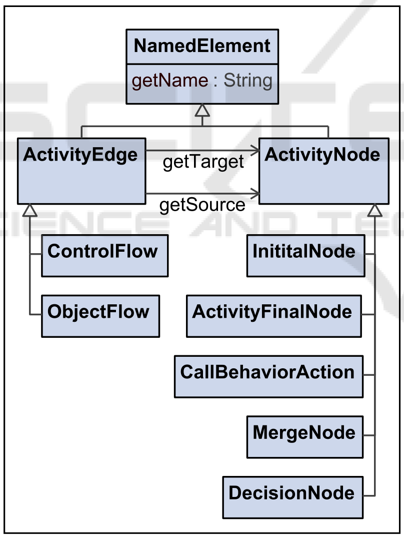


Figure 4: Extracted API structure.

methods of the class that have been added, along with their return types. Links to the corresponding methods or classes are provided, allowing easy navigation within the Javadoc documentation.

To demonstrate the effectiveness of the tool, we selected a specific set of classes and methods using the tool and then generated the corresponding code for the VKG interface. In our previous publication (Weigand and Fay, 2022), we introduced a generic version of the VKG interface, which is available on

```
SELECT ?name WHERE {
  ?action a ex:CallBehaviorAction .
  ?action ex:getName ?name .
}

RESULTS
 -------------------------------------
 | name                              |
 =====================================
 | "move arm to target position"     |
 | "move platform"                   |
 | "move arm to object position"     |
 | "detect object"                   |
 | "open gripper"                    |
 | "close gripper"                   |
 -------------------------------------
```

Listing 1: Exemplary SPARQL query to extract names of CallBehaviorActions. Prefixes were omitted to save space.

GitHub[7]. This generic version includes key components such as the logic to execute a query over the VKG and is intended to be adapted to a specific software. The adaption to MSOSA is also available on GitHub[8]. As an example, we used a SysML Activity Diagram created in MSOSA, shown in Figure 3. The diagram shows a simple pick-and-place process for a robotic manipulator on a movable platform. The objective was to extract all information represented by the diagram elements via the VKG interface. Information contained in the description of this pick-and-place process might be relevant in further engineering phases, such as programming the robotic manipulator that executes the shown process. The resulting UML class diagram, shown in Figure 4, illustrates the TBox of the VKG. In Figure 3, the diagram element types are also indicated using grey boxes. The TBox of the VKG was ultimately created by selecting the 10 classes and 3 methods shown in Figure 4 by browsing the Javadoc with the developed tool and adding classes and methods, which was carried out within a few minutes. The generated code amounted to 6 lines of code per class and 7 lines of code per method, i.e. a total of 81 lines of code. Subsequently, additional steps were taken to complete and compile the entire VKG interface. A sample SPARQL query (see Listing 1) was created. The query retrieves the names (`?name`) of all resources that are of type `ex:CallBehaviorAction` and have a property `ex:getName`. As shown in Figure 4, the `getName` method of the API is defined by the class `NamedElement`. The class `CallBehaviorAction` of the API is a subclass of this class and inherits the method. The results of the query, also shown

[7]github.com/mxweigand/vmax_core
[8]github.com/mxweigand/vmax_plugin_msosa

in Listing 1, correctly included the names of all 6 `CallBehaviorAction` instances shown in Figure 3. Instances of other subclasses, e.g. the `InitialNode` named *init*, were not returned by the query.

6 CONCLUSION AND FUTURE WORK

The objective of this publication was to develop a workflow to automate and support the creation of a VKG interface for the API of an engineering software. Several steps of the developed workflow were implemented in a tool that significantly simplified the implementation effort for the VKG interface. The process was streamlined by displaying the documentation as true to the original as possible, with the additional required features added in a minimalist and intuitive way. The navigational functions of the Javadoc were preserved, ensuring a familiar environment for the developer. Automated steps reduced manual effort while minimizing errors, ensuring an efficient and accurate workflow for the VKG interface creation. Several steps not covered in the tool's implementation but included in the developed workflow were carried out manually. As discussed earlier, these steps are not automated due to their complexity and the need for customization. However, creating guidelines and defining requirements for these manual steps would still be beneficial. Such guidelines would assist in checking prerequisites ahead of time to determine whether the workflow is applicable to a particular API. They would also help streamline the manual steps, making them more efficient, while reducing the likelihood of errors during the process. This will be a focus of future research aimed at enhancing the workflow.

ACKNOWLEDGEMENTS

This research is part of the project iMOD which is funded by dtec.bw – Digitalization and Technology Research Center of the Bundeswehr. dtec.bw is funded by the European Union – NextGenerationEU.

REFERENCES

Ancona, D., Mascardi, V., and Pavarino, O. (2012). Ontology-based documentation extraction for semi-automatic migration of Java code. In *The 27th Annual ACM Symposium on Applied Computing*.

Atzeni, M. and Atzori, M. (2017). CodeOntology: RDF-ization of source code. In *The Semantic Web – ISWC 2017*.

Brickley, D. and Guha, R. (2014). RDF Schema 1.1. W3C Recommendation. *http://www.w3.org/TR/rdf11-schema/*.

Cyganiak, R., Wood, D., and Lanthaler, M. (2014). RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation. *http://www.w3.org/TR/rdf-concepts/*.

De Meester, B., Heyvaert, P., and Delva, T. (2024). RDF Mapping Language (RML) Unofficial Draft 20 June 2024. *https://rml.io/specs/rml/*.

Dotoli, M., Fay, A., Miśkowicz, M., and Seatzu, C. (2018). An overview of current technologies and emerging trends in factory automation. *International Journal of Production Research*, 57(15-16).

Ekaputra, F. J., Sabou, M., Serral, E., Kiesling, E., and Biffl, S. (2017). Ontology-Based Data Integration in Multi-Disciplinary Engineering Environments: A Review. *Open Journal of Information Systems*, 4(1).

Fay, A., Biffl, S., Winkler, D., Drath, R., and Barth, M. (2013). A method to evaluate the openness of automation tools for increased interoperability. In *IECON 2013 - 39th Annual Conference of the IEEE Industrial Electronics Society*.

Harris, S. and Seaborne, A. (2013). SPARQL 1.1 Query Language. W3C Recommendation. *https://www.w3.org/TR/sparql11-query/*.

Hitzler, P., Krötzsch, M., Parsia, B., Patel-Schneider, P. F., and Rudolph, S. (2012). OWL 2 Web Ontology Language Primer (Second Edition). W3C Recommendation. *https://www.w3.org/TR/owl2-primer/*.

Hogan, A., Blomqvist, E., Cochez, M., d'Amato, C., Melo, G. D., Gutierrez, C., Kirrane, S., Gayo, J. E. L., Navigli, R., Neumaier, S., et al. (2021). Knowledge graphs. *ACM Computing Surveys (Csur)*, 54(4).

Ledvinka, M. and Křemen, P. (2020). A comparison of object-triple mapping libraries. *Semantic Web*, 11(3).

Object Management Group (2011). Business Process Model and Notation (BPMN) Version 2.0. *https://www.omg.org/spec/BPMN/2.0/PDF*.

Reddy, M. (2011). *API Design for C++*. Morgan Kaufmann.

Sabou, M., Biffl, S., Einfalt, A., Krammer, L., Kastner, W., and Ekaputra, F. J. (2020). Semantics for Cyber-Physical Systems: A Cross-Domain Perspective. *Semantic Web*, 11(1).

Vogel-Heuser, B., Ocker, F., Weiß, I., Mieth, R., and Mann, F. (2021). Potential for combining semantics and data analysis in the context of digital twins. *Phil. Trans. R. Soc. A*, 379:20200368.

Weigand, M. and Fay, A. (2022). Creating Virtual Knowledge Graphs from Software-Internal Data. In *IECON 2022 - 48th Annual Conference of the IEEE Industrial Electronics Society*.