Speak to your Software Visualization—Exploring Component-based Software Architectures in Augmented Reality with a Conversational Interface

Peter Seipel, Adrian Stock, Sivasurya Santhanam, Artur Baranowski,
Nico Hochgeschwender, and Andreas Schreiber
German Aerospace Center (DLR)
Simulation and Software Technology
Cologne, Germany
forename.surname@dlr.de

***Abstract*—Exploring of software architectures with software visualization in Augmented Reality (AR) is possible with different interaction methods, such gesture, gaze, and speech. For interaction with speech (i.e., natural language), we present an architecture and an implementation of conversational interfaces for the Microsoft HoloLens device. We aim to remedy some peculiarities of AR devices, but also enhancing the exploration task at hand. To implement the conversational interface different natural language processing (NLP) components such as natural language generation and intent recognition are typically required. Our proposed architecture integrates conversational components with the AR-based software visualization. We describe its implementation based on different user utterances, where the system provides information about the to-be-explored component-based software architecture in the form of adjusted visualizations and speech-based results. We apply out tool to explore OSGi-based software architectures.**

***Index Terms*—software visualization, augmented reality, conversational interfaces, chatbots, natural language processing, graph databases, OSGi, software architecture**

I. Introduction

Exploring large-scale software projects with interrelationships among multiple software components is a challenging exercise. Due to its complexity, exploration activities such as navigating to certain components, identifying dependencies among components, assessing metrics of individual architectural elements, and so forth is a cumbersome task. Those activities need to be improved as they are relevant in many scenarios such as gaining a project overview for onboarding new developers, discussing architectural decisions and quality attributes with stakeholders, testing and debugging by developers. To improve the exploration task, we developed multiple approaches in 2D, 3D and Virtual Reality (VR) to visualize software architectures in our previous work [13], [9]. With the use of see-through visors, AR allows to exploit the real world for visualizing software systems effectively. This facilitates the inspection of the software architectures in a collaborative manner. As immersive augmented reality reduces the issue of occlusion and eases navigation [8], it aids the user during this process. To make interactive visualizations accessible real-world metaphors like cities or archipelagos are used in immersive AR for interactive visualization of abstract software entities [5].

Regardless of the visualization approach (such as 2D or 3D), one needs to semantically map software entities to visual metaphors. We visualize OSGi-based [7] software architectures, which are used in many large software systems in industry. The *OSGi Alliance*[1] specifies a component system for Java environments. These components and their resources are packaged in *Bundles*. They are dynamically connected via *Services*, which represent components' communication interface. For large-scale OSGi-based architecture the relationship between provided and required services is often not immediately obvious. Also, dependency relationships between bundles become increasingly opaque as a project grows. This is a motivation for our work to explore OSGi-based software architectures in immersive AR.

Using the Microsoft HoloLens, we use an *island metaphor* [12] for visualizing OSGi-based software architectures in AR [2], [1]. Here, the software structure is represented as an archipelago within a virtual water level (Fig. 1). The virtual water level can be placed somewhere in the room, for example on the floor, a desk, or any table (Fig. 2).

Different levels of granularity are mapped to the metaphor for visualization. For instance, bundles, packages and compilation units are represented as islands, regions and buildings. Using gestures, the archipelagos position, orientation and scale can be adjusted. Also, common software metrics are visualized. For instance, lines of code within a particular compilation unit are mapped to the building's height. In our previous works we have considered exploration just with navigation—however exploration could be enhanced by utilizing the additional input modalities provided by the Microsoft HoleLens, such as voice and gesture control. Voice and gesture control could pave the way for advanced exploration activities such as simultaneously selecting architectural elements via gesture and requesting additional information about this element via speech.

[1]https://www.osgi.org/

978-1-7281-4939-4/19/$31.00 ©2019 IEEE
DOI 10.1109/VISSOFT.2019.00017

Fig. 1: Visualizing software architectures in Augmented Reality with the Microsoft HoloLens using the island metaphor.

II. Motivation and Use Case

In our previous works for VR [9] and AR [2] we use gesture control and hand controller to navigate to individual islands, regions or buildings which can be selected for detailed inspection and displaying service and dependency relationships. However, we observed that relying solely on gesture control makes exploration limited to navigation. For example, searching for a particular bundle by dragging the archipelago around can be inefficient, when instead a query could be issued for finding the island. Likewise, making queries via text input on a virtual keyboard is cumbersome due to limited gesture recognition capabilities. Similarly, searching for components using simple keyword-based approach with voice control is also inefficient as users need to remember lots of keywords for each functionality. This problem of paraphrasing user input was already addressed in several works and resolved by using embedding-based techniques [14], [4].

Fig. 2: Placement of the software island on a table.

We propose the use of natural language understanding techniques to recognize user's intent despite the variations in the utterances. To this end, intent recognition is employed to classify utterances based on their semantic content and similarity. This unburdens users from learning and remembering keywords. Also, conversational interfaces can act more natural in so far as the interface serves as a virtual assistant, providing extended feedback to the user. To capitalize on those benefits, we describe an architecture for integrating a conversational interface into an AR device such as the Microsoft HoloLens (Sec. IV). The proposed system also considers the contextual information of the visualization (e.g., which bundles are in view, etc.) to make exploration tasks such as searching implementable.

Our main *use case* so far is to view, explore, and explain software architecture collaboratively. Two persons equipped with HoloLens devices view the software visualization in the same room, where the visualization is placed on a desk or any table. A specific use case is, that an experienced developer explains the architecture to a new team member. Our goal is, that the conversational interface based on utterances “automatically” follows the conversation and adapts the visualization accordingly.

Using AR instead of VR for the described use case might foster the adaption of our visualization at workplaces more easily, which has to be verified in future user studies—for example, by comparing VR and VR for this use case.

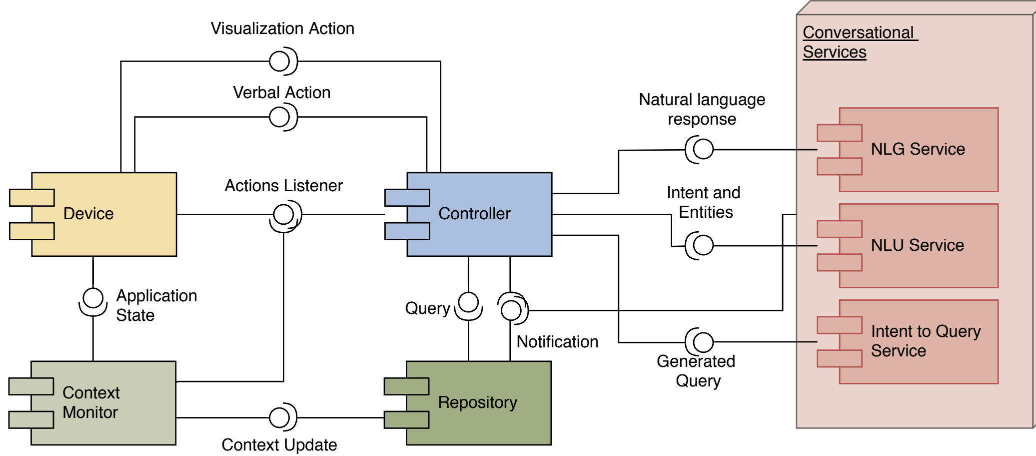

Fig. 3: Component diagram of conversational interface integration with the software exploration system.

III. Architecture of the Conversational Interface in Software Exploration

We propose an architecture integrating different input and output modalities such as gesture, gaze, and speech. The general idea of the architecture is to provide a template solution for developing software exploration applications with AR/VR devices. The proposed architecture (Fig. 3) consists of several components that provide and require interfaces. By following the Model-View-Controller (MVC) design pattern, *Repository*, *Device*, and *Controller* function as Model, View, and Controller. The components are:

a) Device: The device works as the user's interface to the system, thus taking care of the user's input. The system is agnostic towards the way it presents the information. It is not restricted to use an AR device as in our use case (Sec. IV) but could even be applied to visualizations in 2D.

b) Context Monitor: The *Context Monitor* observes user-initiated actions the user executes, including but not limited to both gesture and speech. Additionally, it keeps track of the current focus of the visualization by observing the *Application State*. Based on this, the current context is created. Every time the context changes, the *Context Monitor* writes the new context into the *Repository*. Using the context, the system can execute actions based on the current scenario.

c) Controller: The *Controller* is the central core of the system connecting and regulating various services, *Repository* and the *Device*. Based on the inputs from the *Device*, the *Controller* orchestrates the services to be called. We designed the *Controller* with modularity in mind, so that each module is independently integrated based on their functionality. The modular architecture of the *Controller* allows to implement different dialogue control strategies based on rules, machine learning, and so forth [6].

d) Repository: The *Repository* persistently stores the software architecture to be explored and contextual information about current and past visualizations of the architecture. With appropriate queries, insightful information can be inferred from the *Repository*.

e) Conversational Services: The *Conversational Services* consist of the components Natural Language Understanding (NLU), Intent to Query, and Natural Language Generation (NLG). The *Controller* passes the user's natural language input along with the context to the NLU unit. The NLU service recognizes both the intents and entities and returns back the results to the *Controller*. The *Intent to Query* service transforms the intents and entities to a query. These queries are sent to the *Controller*, which then requests the *Repository* for a response. To have an engaging conversation, the dialog system should be able to respond to and ask the user in a human readable format. To achieve that, the responses from the *Repository* are used as seeds to build natural language sentences using NLG service.

IV. Implementation

In this section, we describe the implementation of our proposed architecture for a use case where a user demands a search query with voice input. We first describe the technologies that we used to implement the scenario. We present a detailed description of the implementation about the data flow among the components in the following paragraphs.

A. Configuration

We use the *Microsoft HoloLens*[2] in our work. Being an IO device, it includes services such as speech-to-text, text-to-speech, gesture control, and speakers. The HoloLens is just one of many VR/AR/MR devices; therefore our architecture is not restricted to this specific device. It is also possible to integrate our proposed system into other VR environments. When integrated with the conversational interfaces, the features of the HoloLens could be highly exploited to better help exploration of architectures.

[2]https://www.microsoft.com/en-us/hololens

The *Controller* includes an implementation of *Rasa core* [3] for the dialog management system. We use the graph database NEO4J in the repository, as graph databases are suitable choices for semantic queries, due to its nature of interconnected data. To convert the source data into a graph in NEO4J, we use the Open Source tool JQASSISTANT [10], [11]. The *NLU Service* in the *Conversational Services* uses RASA NLU [3] to detect intents and entities. The *Intent-to-Query* service builds CYPHER queries with respect to intents and entities.

B. Use case

The user either uses a hand-gesture or a speech input to communicate with the system. In either case, the following sequence takes place. In this example we will focus on a speech input. Assuming the user has already navigated to the bundle named "core comp" and selected this bundle using the tap-gesture. Now the user says "find the class with the highest number of methods inside this bundle". The *Device* will convert this audio input into string using the *speech-to-text service* and provides an event. The *Controller* gets notified by this *Event*. An *Event* contains all the data that is needed for further processing (e.g., the user utterance as a string or the information that a double-tap-gesture is detected). Simultaneously, if there is any change in the context, the *Context Monitor* recognizes the *Application State* and updates the context. It then writes the new context into the *Repository*. Due to the new entry the *Repository* notifies the *Controller*. This ensures the *Repository* to have updated information about the current state.

Once the *Controller* receives the input, the *Controller* sends the user's utterance as a string to the *NLU Service*, which responds with corresponding *Intent and Entities* to the *controller*. As RASA NLU is trained using machine learning methods for identifying *Intents and Entities*, sentences with variations in words but having semantic equivalence will be identified as having same intent. In our example, the detected Intent is "select_class_with_most_methods" and no entities are detected. The *Controller* requests the *Repository* for updated context. In our case, the updated context was already written into the *Repository* when the user made the selection via tap-gesture as shown below.


```
{
 "focused_object" : null,
 "focused_object_type" : null,
 "selected_object" : "Core Comp",
 "selected_object_type" : "bundle"
}
```


The *Controller* passes the Intents, entities and the context to the *NLU Service*. Depending upon the intent, The *NLU Service* decides whether the context to be used as entity. Contexts are only transformed into entities when needed. Table I shows some variations in utterances and their corresponding intents and entities along with possible contexts. Once, the intents and entities are resolved, they have to be converted to a query. To transform *Intent and Entities* to a graph query, the system uses the *Intent to Query service*. Based on a template-based approach, a query template for the corresponding intent is selected and entities are filled. For the example at hand, the generated CYPHER query is:

```
MATCH
(b:Bundle{name:'Core_Comp'})-[]->(c:Class),
(c)-[d:DECLARES]->(m:Method) RETURN c,
COUNT(m) ORDER BY COUNT(m) DESC LIMIT 1
```

Next the *Controller* uses the generated graph query, which represents the user's command, to obtain the response from the *Repository*. The repository with the NEO4J graph database provides a JSON response to the concerned CYPHER query as shown. In our case, the name of the bundle is "Component-ContextImpl" it possesses 54 methods.


```
{
 {"srcFileName":"ComponentContextImpl.
    java",
  "fileName": "/de/rce/core/component/
     ComponentContextImpl.class",
  "fqn": "de.rce.core.component.
     ComponentContextImpl",
  "visibility": "public",
  "name": "ComponentContextImpl"
 },54
}
```


Apart from the name of the bundle and its methods, it also provides lots of information and they are structured data. The user expects a tailored natural language response to the utterance. The controller selects the required data (e.g, type, name and number of methods) and sends it to the *NLG Service*, which creates a natural language sentence based on the received information. "The class with the highest number of methods is ComponentContextImpl with 54 methods" is the sentence generated for the given data. Both, the JSON response from the graph database and the natural language sentence, are sent to the device by the *Controller*. The JSON is parsed on the device and the extracted information is displayed on a virtual info panel. Depending on the given data, the panel shows the list of entries or the requested summary. In addition to this info panel, the visualization re-formats and focuses on the biggest class as requested. Concurrently, the device transforms the sentence generated by the *NLG Service* to an audio stream by using the *text-to-speech API*. This is played while the visualization update happens.

The same procedure works for other utterances as well. Table I contains some examples. The table shows the Intents and Entities that the NLU service extracts from the utterance and the current context. The depicted examples form an instance of a sequence of different utterances. When the user demands "Please show me bundle core.auth," the NLU service identifies the Intent "select_component" and the Entities "bundle" and "core.auth." The focus of the device then changes so it shows the desired bundle. At this point,

TABLE I: Examples of Utterances and their Intents and Entities Parsed.

#	Utterance	Intent	Entity	Extracted information from current context
1	*"Please show me bundle core.auth, select core.auth, ..."*	select_ component	bundle, core.auth	None
2	*"Tell me more about this bundle, what is this bundle about?, ..."*	summarize_ information	bundle	core.auth
3	*"Select the largest class of this bundle, show the largest class, ..."*	select_ component	class, biggest	core.auth
4	*"How do I select an island?, ..."*	explain_ usage	None	AuthService Impl.java
5	*"How many packages are inside this bundle?, ..."*	count_ component	bundle	AuthService Impl.java
6	*"How many classes are inside this bundle?, ..."*	count_ component	class, bundle	AuthService Impl.java
7	*"Which is the biggest package of this bundle?, ..."*	select_ component	biggest, package, bundle	AuthService Impl.java
8	*"navigate to the smallest bundle?, ..."*	select_ component	smallest, bundle	de/rce/core/ component/

the current context provides no information, as this was the 1st utterance. When the user asks for more information about the bundle, the system knows that the current context is the bundle "core.auth" as it was the topic of the previous utterance. Thus, it can provide the information about the bundle even though its name wasn't mentioned in the utterance. The 3rd utterance combines both the current context and the entities to process the user's input. The 4th example is a simple question about how to use the ISLANDVIZ software. The context here is "`AuthServiceImpl.java`," which was the result of the previous processing of the user's utterance.

C. Status and Availability

Our implementation ISLANDVIZ FOR HOLOLENS is available as Open Source under an Apache 2.0 license [1]. The NLU implementation was started as a separate internal project and is now being integrated in the ISLANDVIZ FOR HOLOLENS software.

V. CONCLUSION AND FUTURE WORK

We proposed to adopt conversational interfaces for enhancing users to explore OSGi-based software architectures to be visualized in immersive AR. By integrating intent-based NLU components into the overall architecture, we enable users to search seamlessly in the natural language. Thus, empowering the user to make use of both gesture and speech actions simultaneously for the exploration. This paves the way to implement virtual assistants which are capable to hold engaging conversations about the exploration task at hand.

In our future work we will conduct user studies to investigate whether or not users are enticed by the conversational technology to search through the architecture in a more efficient manner. To do so, we will store and analyze contextual information such as the history of all visited architectural elements. In addition, we plan to use this information to guide the exploration process to those architectural elements which were not yet visited and observed. General future work include adaption of the ISLANDVIZ visualization to other component models than OSGi.

REFERENCES

[1] A. Baranowski and P. Seipel. DLR-SC/holo-island-viz: VRST2018 demo version v0.1, Nov. 2018.

[2] A. Baranowski, P. Seipel, and A. Schreiber. Visualizing and exploring osgi-based software architectures in augmented reality. In *Proceedings of the 24th ACM Symposium on Virtual Reality Software and Technology*, VRST '18, pages 62:1–62:2, New York, NY, USA, 2018. ACM.

[3] T. Bocklisch, J. Faulker, N. Pawlowski, and A. Nichol. Rasa: Open source language understanding and dialogue management. *arXiv preprint arXiv:1712.05181*, 2017.

[4] A. Di Prospero, N. Norouzi, M. Fokaefs, and M. Litoiu. Chatbots as assistants: An architectural framework. In *Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering*, CASCON '17, pages 76–86, Riverton, NJ, USA, 2017. IBM Corp.

[5] S. Diehl. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer-Verlag, Berlin, Heidelberg, 2007.

[6] B. Dumas, D. Lalanne, and S. L. Oviatt. Multimodal interfaces: A survey of principles, models and frameworks. In *Human Machine Interaction*, 2009.

[7] J. McAffer, P. VanderLei, and S. Archer. *OSGi and Equinox: Creating Highly Modular Java Systems*. Eclipse Series. Addison-Wesley, Upper Saddle River, NJ, 2010.

[8] L. Merino, A. Bergel, and O. Nierstrasz. Overcoming issues of 3d software visualization through immersive augmented reality. In *2018 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 54–64. IEEE, 2018.

[9] M. Misiak, D. Seider, S. Zur, A. Fuhrmann, and A. Schreiber. Immersive exploration of osgi-based software systems in virtual reality. In *2018 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*, volume 00, pages 1–2, March 2018.

[10] R. Müller, D. Mahler, M. Hunger, J. Nerche, and M. Harrer. Towards an open source stack to create a unified data source for software analysis and visualization. In *2018 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 107–111, Sep. 2018.

[11] L. Nafeie and A. Schreiber. Visualization of software components and dependency graphs in virtual reality. In *Proceedings of the 24th ACM Symposium on Virtual Reality Software and Technology*, VRST '18, pages 133:1–133:2, New York, NY, USA, 2018. ACM.

[12] A. Schreiber and M. Misiak. Visualizing software architectures in virtual reality with an island metaphor. In J. Y. Chen and G. Fragomeni, editors, *Virtual, Augmented and Mixed Reality: Interaction, Navigation, Visualization, Embodiment, and Simulation*, pages 168–182, Cham, 2018. Springer International Publishing.

[13] D. Seider, A. Schreiber, T. Marquardt, and M. Brüggemann. Visualizing modules and dependencies of osgi-based applications. In *2016 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 96–100, Oct 2016.

[14] R. Socher, E. H Huang, J. Pennington, A. Y Ng, and C. Manning. Dynamic pooling and unfolding recursive autoencoders for paraphrase detection. *Advances in Neural Information Processing Systems*, 24, 01 2011.