Towards Generating Labeled Property Graphs for Comprehending C#-based Software Projects

David Heidrich
German Aerospace Center (DLR)
Weßling, Germany
david.heidrich@dlr.de

Andreas Schreiber
German Aerospace Center (DLR)
Cologne, Germany
andreas.schreiber@dlr.de

Sebastian Oberdörfer
University of Würzburg
Würzburg, Germany
sebastian.oberdoerfer@uni-wuerzburg.de

ABSTRACT

C# is the most widely used programming language among XR developers. However, only a limited number of graph-based data acquisition tools exist for C# software. XR development commonly relies on reusing existing software components to accelerate development. Graph-based visualization tools can facilitate this comprehension process, e.g., by providing an overview of relationships between components. This work describes a new tool called Src2Neo that generates labeled property graphs of C#-based software projects. The stored graph follows a simple C# naming scheme and — contrary to other solutions — maps each software entity to exactly one node. The resulting graph facilitates the comprehension process by providing an easy to read representation of software components. Additionally, the generated graphs can act as a data basis for more advanced software visualizations without the need for complex data requests.

CCS CONCEPTS

• **Human-centered computing** → *Visualization*; • **Computer systems organization** → *Real-time system architecture*; • **Information systems** → *Information retrieval.*

KEYWORDS

software visualization, software comprehension, labeled property graph, c#, game engines

ACM Reference Format:
David Heidrich, Andreas Schreiber, and Sebastian Oberdörfer. 2022. Towards Generating Labeled Property Graphs for Comprehending C#-based Software Projects. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3551349.3560513

1 INTRODUCTION

In the area of XR development and research, the large amount of technical software requirements — like low latencies [6], high frame rates [23], and hardware compatibilities — have led to a widespread adoption of 3D game engines. While these game engines, such as *Godot*, *CryEngine*, or *Unity*, provide a wide range of pre-existing features and systems, they also commonly provide native support for the programming language C#. As this makes C# the most widely used programming language among XR developers [19], even game engines without C# capabilities, like *Unreal Engine*, allow C# scripting through various community plugins [5, 11].

By not using a proprietary scripting language, C# game engines enable developers to reuse source code of existing C#-based software projects. This can reduce development overhead and allows for rapid prototyping. In some cases, like *Godot*, actually the whole C# source code of the game engine is open source [7]. This gives developers, researchers, and students the ability to modify all features and systems provided by the game engine. However, before developers can modify or reuse existing source code, they must first build a basic understanding of the overall software system and its components. Due to the abstract and complex nature of source code, this comprehension can quickly evolve into a mentally demanding and time-consuming activity. This is especially the case for larger software projects, where even professional developers invest more than 50% of their working time on software comprehension instead of writing or modifying source code [22].

To facilitate this comprehension process, we generate labeled property graphs of C#-based software projects. More specific, we present our work-in-progress tool Src2Neo that converts a srcML file to a graph stored in a Neo4j database. This work describes the structure of Src2Neo in detail and presents a simple graph database model for C# software structure. Finally, we discuss how our approach can facilitate software comprehension and can result in more advanced software visualizations.

2 RELATED WORK

As graphs are a fundamental data representation of software structures, graph-based visualization techniques are the most popular type of software visualization [18]. Typically, graphs consist of nodes and edges. In the context of software structures, nodes generally represent software elements, like namespaces or classes. Edges typically represent relationships between software elements, like a namespace CONTAINS a class or a method CALLS another method. *Labeled property graphs* are a certain type of graph that are used to model real-world entities and their relationships to nodes and edges [1, 16]. Here, all nodes and edges of a specific type have a shared label. For example, all nodes that represent a namespace receive a *Namespace* label. Additionally, nodes and edges can store additional data in form of properties. That way, a *Class* node could, e.g., contain information about its lines of code or code complexity.

As labeled property graphs can model many aspects of a software system, they are often proposed as a unified data source for

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10.
https://doi.org/10.1145/3551349.3560513

```
<unit revision="1.0.0" language="C#" filename="Triangle/Triangle/Meshing/Data/BadSubseg.cs" hash="4a5e7cbf33db4b506e254a9f20e6763d6b864ac7">
<namespace>namespace <name><name>TriangleNet</name><operator>.</operator><name>Meshing</name><operator>.</operator><name>Data</name></name>
<block>{
    <using>using <name>System</name>;</using>
    <using>using <name><name>TriangleNet</name><operator>.</operator><name>Geometry</name></name>;</using>
    <using>using <name><name>TriangleNet</name><operator>.</operator><name>Topology</name></name>;</using>

    <class>class <name>BadSubseg</name>
    <block>{
        <decl_stmt><decl><type><specifier>public</specifier> <name>Osub</name></type> <name>subseg</name></decl>;</decl_stmt> <comment type="line">// An encroached subsegment.
        </comment>
        <decl_stmt><decl><type><specifier>public</specifier> <name>Vertex</name></type> <name>org</name></decl>, <decl><type ref="prev"/><name>dest</name></decl>;</decl_stmt>
        <comment type="line">// Its two vertices.</comment>

        <function><type><specifier>public</specifier> <specifier>override</specifier> <name>int</name></type> <name>GetHashCode</name><parameter_list>()</parameter_list>
        <block>{<block_content>
```

Figure 1: Section from a srcML file built from a C#-based mesh generator library [10].

software analysis and visualization [14, 17, 20]. However, data acquisition can be difficult based on the programming language. Due to differences between languages, data acquisition tools are generally designed for one programming language only. Hence, different acquisition toolings exist for, e.g., C++ [3, 13] or Java [14]. Besides the "C# Plugin" [2] for the open-source software analysis tool jQAssistant [14], there are — to the best of our knowledge — currently no tools capable of mapping C#-based software projects to a labeled property graph.

Generating a labeled property graph requires a predefined meta model of the graph database. The C# Plugin for jQAssistant already introduced a meta model for C#-based software systems [2]. However, we chose against using this model as it is designed for software analysis tasks. Hence, resulting graphs are very complex and include many meta nodes, like *Member* or *File*. Additionally, there are no direct edges between, e.g., class and namespace nodes or method and class nodes. Hence, visualization tools require complex data queries to gain basic insights.

3 LABELED PROPERTY GRAPHS

For our graph database meta model, we chose a simple design solely based on the C# software structure (see Figure 2). It only includes the node labels *Namespace*, *Class*, *Interface*, *Enum*, *Method*, and *Field*. The model also includes the edge labels *CONTAINS*, *IMPORTS*, *IMPLEMENTS*, *INHERITS_FROM*, and *CALLS*. Additionally, nodes contain properties, like "Name", "File Path", and "Lines of Code". However, additional properties can easy be added to nodes.

3.1 Src2Neo

We use XML files generated with the open source tool srcML [4]. Among other languages, srcML parses C# source code to a XML file. This file contains all original information, including file structure, white spaces and comments. Inside the XML file, all syntax elements receive individual XML-tags (see Figure 1). For example, a <unit> tag represents a file and <namespace>, <class>, and <function> tags represent their C# counterparts. As all software components can easily be addressed, srcML is commonly used in software metrics extraction [15].

Our tool Src2Neo then converts a given srcML-generated XML file to a labeled property graph. First, it identifies all software components (i.e., namespaces, classes, interfaces, enums, methods, and fields). To navigate inside the XML file and to find specific XML

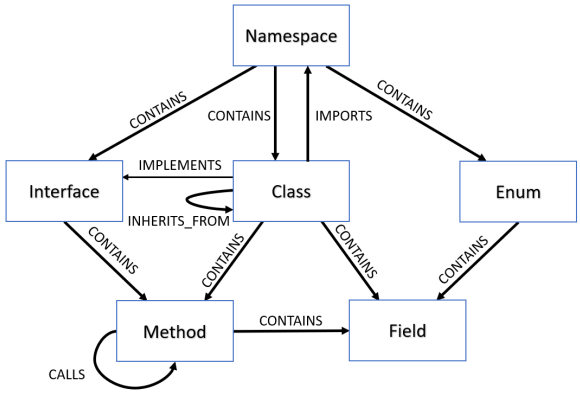

Figure 2: Meta model of the Src2Neo graph database.

nodes, we use XPath expressions. For example, we use the expression "./def:class" to find all class nodes inside the XML file or "./def:namespace" to get all namespaces. For each identified software component, we extract their software metrics.

After identifying all individual software components, we look for relationships between them. For example, to identify *Namespace CONTAINS Class* relationships, we use the XPath expression "./../../../def:namespace" on each class node in the XML. If a parent namespace is found, we store the relationship. Other examples are the *Class IMPORTS Namespace* relationships, where we look for the <using> tag with the XPath expression "./../def:using" on each class, or the *Method CALLS Method* relationship, where we look for the <call> tag inside a method and resolve the called name.

Finally, we write the data to a Neo4j graph database (see Figure 3). During the writing process, we go through all identified software components and store each component as an individual graph node. Then we add all software component relationships to the graph. All data is added via Cypher queries which are sent to the Neo4j server using the .Net Neo4j driver [8].

4 DISCUSSION

The generated nodes are a 1:1 representation of a C#-based software project. They do not include any meta nodes that require complicated data queries that can slow down the comprehension process. Hence, developers, researchers, and student can explore

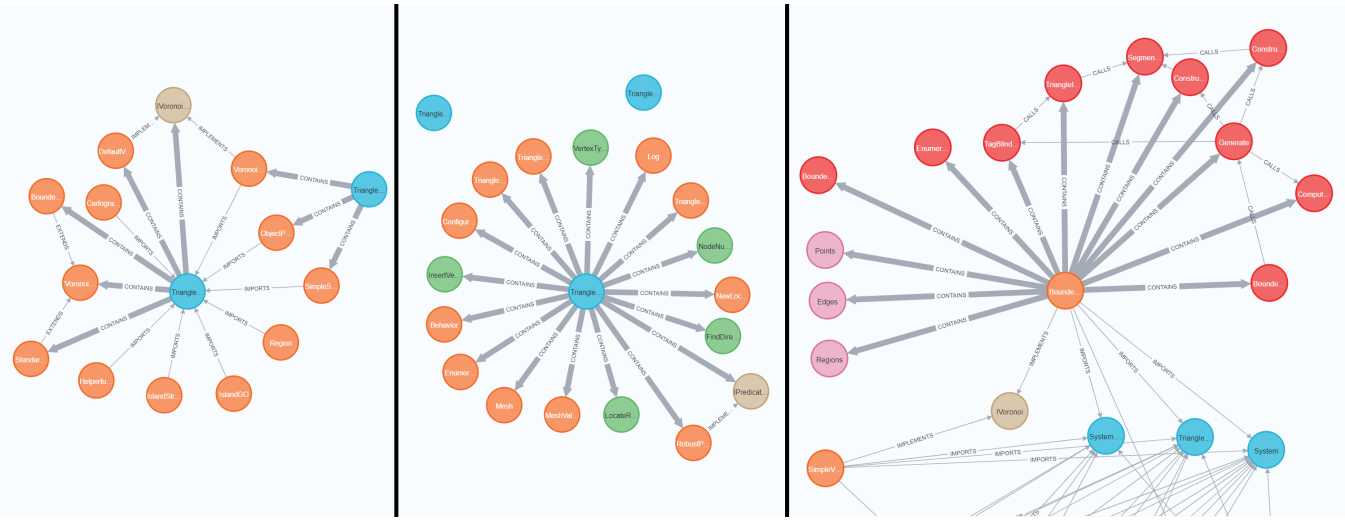

Figure 3: Different views of a generated labeled property graph visualized with the Neo4j Browser. Namespace nodes are blue, class nodes are orange, interface nodes are light-brown, enum nodes are green, method nodes are red, and field nodes are pink.

the software components without the need for specialized graph visualization tools. However, the relationship placement can be more challenging. To reduce complexity, our database meta model does not include all possible relationships. For example, we only use the IMPORTS relationship between classes and namespaces (as it is the case in the source code). But in some use cases, a representation with IMPORTS relationships between namespaces might be the more suitable solution. Hence, our graphs will not replace advanced software visualization tools with custom data queries.

Advanced software visualization tools could, however, benefit from using our generated graphs as a data source. On one hand, the easy graph layout could facilitate rapid software visualization prototyping (especially for unskilled software developers). On the other hand, metaphor-based software visualization tools, like IslandViz [12] or Code City [21], could map their metaphors to specific nodes and edges inside the graph database. This could reduce the complexity of such big open-source software visualization tools and make them more accessible.

5 FUTURE WORK

We plan to extend our meta model with more relationships, e.g., a TYPE_OF relationship between a field and a class. At the same time, we want users to be able to choose the level of detail of the generated graph. For example, users might want to combine methods and fields into a single *Class Member* node or do not want to include certain relationships, like *Method CALLS Method*, to keep the graph database more simple.

We are currently identifying possible user requirements following an user-centered design process. After this design process is finished, we plan to evaluate and quantify the benefits of Src2Neo and its generated labeled property graphs.

We also plan to support more C# components, like structs, records, generic classes, partial classes, or anonymous types. Additionally, we are thinking about adding specific support for C#-based game engine projects, like Godot or Unity projects, including meta files and resource files. Finally, as srcML is already capable of tagging C++ and Java projects, we want to add support for these programming languages in the future, too. Src2Neo is currently developed in C# and available on GitHub [9].

REFERENCES

[1] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter Boncz, George Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan Sequeda, et al. 2018. G-CORE: A core for future graph query languages. In *Proceedings of the 2018 International Conference on Management of Data*. 1421–1432.

[2] Stefan Bechert and Richard Müller. 2020. *jQAssistant C# Plugin.* https://github.com/softvis-research/jqa-csharp-plugin

[3] Andrea Biaggi, Francesca Arcelli Fontana, and Riccardo Roveda. 2018. An Architectural Smells Detection Tool for C and C++ Projects. In *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 417–420. https://doi.org/10.1109/SEAA.2018.00074

[4] Michael L Collard, Michael J Decker, and Jonathan I Maletic. 2011. Lightweight transformation and fact extraction with the srcML toolkit. In *2011 IEEE 11th international working conference on source code analysis and manipulation*. IEEE, 173–184.

[5] Stanislav Denisov, Josh Olson, Stan Prokop, Sean Devonport, and Victor Müller. 2022. *UnrealCLR.* https://github.com/nxrighthere/UnrealCLR

[6] Mohammed S Elbamby, Cristina Perfecto, Mehdi Bennis, and Klaus Doppler. 2018. Toward low-latency and ultra-reliable virtual reality. *IEEE Network* 32, 2 (2018), 78–84.

[7] Github. 2022. *Godot Engine.* https://github.com/godotengine/godot

[8] Github. 2022. *Neo4j .NET Driver.* https://github.com/neo4j/neo4j-dotnet-driver

[9] Github. 2022. *Src2Neo.* https://github.com/DLR-SC/src2neo

[10] Github. 2022. *Triangle.NET.* https://github.com/wo80/Triangle.NET

[11] Mikayla Hutchinson. 2019. *MonoUE.* https://mono-ue.github.io/

[12] Martin Misiak, Andreas Schreiber, Arnulph Fuhrmann, Sascha Zur, Doreen Seider, and Lisa Nafeie. 2018. IslandViz: A tool for visualizing modular software systems in virtual reality. In *2018 IEEE Working Conference on Software Visualization (VISSOFT)*. IEEE, 112–116.

[13] Johann Mortara, Philippe Collet, and Xhevahire Tërnava. 2020. Identifying and Mapping Implemented Variabilities in Java and C++ Systems using symfinder. In *Proceedings of the 24th ACM International Systems and Software Product Line Conference-Volume B*. 9–12.

[14] Richard Müller, Dirk Mahler, Michael Hunger, Jens Nerche, and Markus Harrer. 2018. 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)*. IEEE, 107–111.

[15] Roy Oberhauser. 2020. A Machine Learning Approach Towards Automatic Software Design Pattern Recognition Across Multiple Programming Languages *(Proceedings of the Fifteenth International Conference on Software Engineering Advances)*. IARIA, 27 – 32. https://nbn-resolving.org/urn:nbn:de:bsz:944-opus4-10255

[16] Ian Robinson, Jim Webber, and Emil Eifrem. 2015. *Graph databases: new opportunities for connected data.* " O'Reilly Media, Inc.".

[17] Aashik Sadar and Vinitha Panicker. 2015. DocTool-a tool for visualizing software projects using graph database. In *2015 Eighth International Conference on Contemporary Computing (IC3)*. IEEE, 439–442.

[18] Mojtaba Shahin, Peng Liang, and Muhammad Ali Babar. 2014. A systematic review of software architecture visualization techniques. *Journal of Systems and Software* 94 (2014), 161–185.

[19] SlashData. 2021. State of the Developer Nation 20th Edition.

[20] Lynn von Kurnatowski, David Heidrich, Nalin Güden, Andreas Schreiber, Hendrik Polzin, and Christian Stangl. 2021. Analysing and Visualizing large Aerospace Software Systems. In *ASCEND 2021*. 4082.

[21] Richard Wettel and Michele Lanza. 2008. Codecity: 3d visualization of large-scale software. In *Companion of the 30th international conference on Software engineering*. 921–922.

[22] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E Hassan, and Shanping Li. 2017. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering* 44, 10 (2017), 951–976.

[23] Jianghao Xiong, En-Lin Hsiang, Ziqian He, Tao Zhan, and Shin-Tson Wu. 2021. Augmented reality and virtual reality displays: emerging technologies and future perspectives. *Light: Science & Applications* 10, 1 (2021), 1–30.