Master Thesis Proposal

Thesis Title (WIP!): Inspecting Java Program States with Semantic Web Technologies

Anton W. Haubner

2021-10-21

TECHNISCHE
UNIVERSITÄT
DARMSTADT

1 Motivation

When implementing a concept from an application domain in a mainstream programming language, its implementation may often differ strongly from its original design. This can be due to optimizations for efficiency because the implementation language lacks the necessary expressiveness or due to other reasons.

For example, Simple Graphs are an abstract structure frequently used to model and solve problems from many application domains. E.g. they can be used to model computer networks, molecules from chemistry, etc. They consist of vertices that are connected via edges. However, to increase space-efficiency or performance of certain computations, their representation in computer programs usually does not reflect this concept directly. Instead, they can for example be implemented as adjacency matrices where the indices of a matrix model the vertices, while the matrix values indicate the presence of an edge. This representation may be preferred since it is space-efficient for non-sparse graphs, it can look up in constant time whether two vertices are adjacent, and one can directly apply methods of algebraic graph theory to it.

As such an implementation can differ so strongly from the original concept, this complicates traditional debugging methods. For instance, reviewing the adjacency matrix of a large graph in the value inspector of a debugger does not reveal much insight into the structure. Furthermore, optimized implementations like the adjacency matrix can often syntactically represent structures that are invalid in the application domain. Here, self-loops that connect vertices to themselves are not allowed for Simple Graphs but can be represented by adjacency matrices with entries on the diagonal. Detecting such faulty structures during debugging is tedious or requires writing custom code for analysis.

This problem intensifies if we move to even richer application domains. For illustration, more complex types of graphs can be used to model molecules, and graph rewriting systems can be utilized to explore their reactions [16, 10]. Debugging if and when a rewriting system implementation produces invalid or unlikely molecules becomes tedious quickly due to the reasons listed above.

Instead, we can imagine a debugger that understands the application domain. That is, it understands that atoms are modeled as vertices in a molecule graph and that they have bond relations which each other. It also understands how these vertices are implemented in the terms of the underlying programming language and what objects represent them in a program's memory.

A debugger with this understanding of the application domain would allow us to perform queries about complex concepts on the state of a program. E.g. we could query "for all carbon atoms with more than four bonds" which is a highly unlikely reaction result and could indicate a bug in the graph rewriting system. The debugger then could supply us with all adjacency matrix indices corresponding to such atoms.

Semantic Debugging In [8], Kamburjan et al. explore such ideas. They apply existing technologies of the Semantic Web [3] to extract semantic knowledge databases from program states. They formalized a minimal object-oriented language, *SMOL*, and extract knowledge in the well-supported RDF graph format. This enables the application of semantic technology for analysis and debugging of program states and even verification of programs.

For example, Kamburjan et al. introduce the idea of *Semantic Debugging*. That is, a debugger that allows investigating program states without having to manually inspect the call stack and heap memory. Instead, the user can perform queries across the whole state using the well-established SPARQL query language. Furthermore, users may use the OWL language to define their own ontologies for their application domain. This permits them to query the program state using domain knowledge.

Kamburjan et al. also demonstrate other uses of the extracted RDF knowledge base. E.g. invariants that reference concepts from the application domain which can be proven using facts from the knowledge base. They also demonstrate, how a knowledge base may be queried at runtime to give programming languages reflection capabilities on the application domain level.

1.1 Thesis Topic

This thesis wants to lay the foundations for the application of Semantic Web technologies in a mainstream programming language: Java. Thus, the extraction of RDF knowledge graphs shall be implemented for Java. Moreover, their usefulness shall be demonstrated by realizing a semantic debugger for the language.

Kamburjan et al. have shown the benefits of such a debugger for the model language SMOL. However, semantic debugging does introduce additional complexity to the debugging process: For one, the user must be knowledgeable in the use of semantic technologies like RDF, SPARQL, etc. They must be able to comprehend the programming language domain model that gives semantics to the RDF graph and apply it to formalize their own application domain. Also, the construction of knowledge bases can be computation-intensive for rich program states and may introduce delays to the debugging process [8].

Thus, this thesis also explores whether the semantic debugging approach is still viable for the more complex Java language. Additionally, since Java is widely used in practice, the thesis will make the benefits of semantic debugging available for the development of real-world applications. Furthermore, it will open up Java to further application of semantic technologies.

In the remainder of this section, we will give a *brief* overview of the contents of the thesis. Section 2 will aggregate the contents into work packages and give more detailed insights into possible tasks and challenges. It also features an illustration of a typical semantic debugging workflow and the involved implementation components. Section 3 defines the time constraints for the implementation of this thesis and gives a preliminary schedule for the work packages.

1.2 Thesis Content Overview

One of the first challenges posed by this thesis is for the student to familiarize themselves with the technologies of the Semantic Web and their application.

Next, a suitable model for the *internal domain* of execution states of Java programs must be developed. It provides the necessary semantics for any RDF graph to be extracted from a Java state. As Java is a rich language, this is expected to be one of the more work-intensive contributions of the thesis.

Furthermore, an interface to the Java Virtual Machine must be implemented which permits access to the state of a (paused) Java program.

A mapping algorithm needs to be realized which extracts an RDF graph as a knowledge base from such a state. Then, the debugger must implement an *answering engine* that accepts SPARQL queries for that knowledge base and returns those nodes of the RDF graph that satisfy the query. The user should also be able to supply a custom application domain to the debugger relying on the definitions of the internal domain (e.g. as a file in OWL/RTF format). Accordingly, the user should be able to reference concepts from their application domain in their query. Optionally, support for validating SHACL shapes on the graph may also be implemented as it simplifies the detection of faulty structures.

To deal with the Semantic Web data formats (RTF etc.) and to perform queries on the knowledge base, existing tools like Apache Jena [6] should be utilized.

Finally, to facilitate the practical application of the debugger, it shall be incorporated as a plugin into an Integrated Development Environment (IDE) for Java, e.g. IntelliJ IDEA [13]. This plugin shall provide a *Read-Evaluate-Print-Loop (REPL)* that permits users to enter queries for the answering engine. For the RDF nodes resulting from a query, it must be able to reverse the RDF graph mapping and display the corresponding Java objects to the user in a traditional value inspector.

The usage of the implemented state mapper and debugger shall be demonstrated in a case study. Furthermore, their performance on program states of varying complexity and for different semantic tasks must be evaluated.

2 Work Packages

From now on, we may refer to the state mapper and semantic debugger, as well as other implementation components of this thesis as “thesis tooling”.

In this section, we give a detailed description of all work packages that constitute this thesis. Even though the work packages mostly only describe implementation tasks, they also always require textual documentation of the implementation work.

Figure 1 illustrates the role of the work packages in a typical workflow of the semantic debugger that is to be implemented.

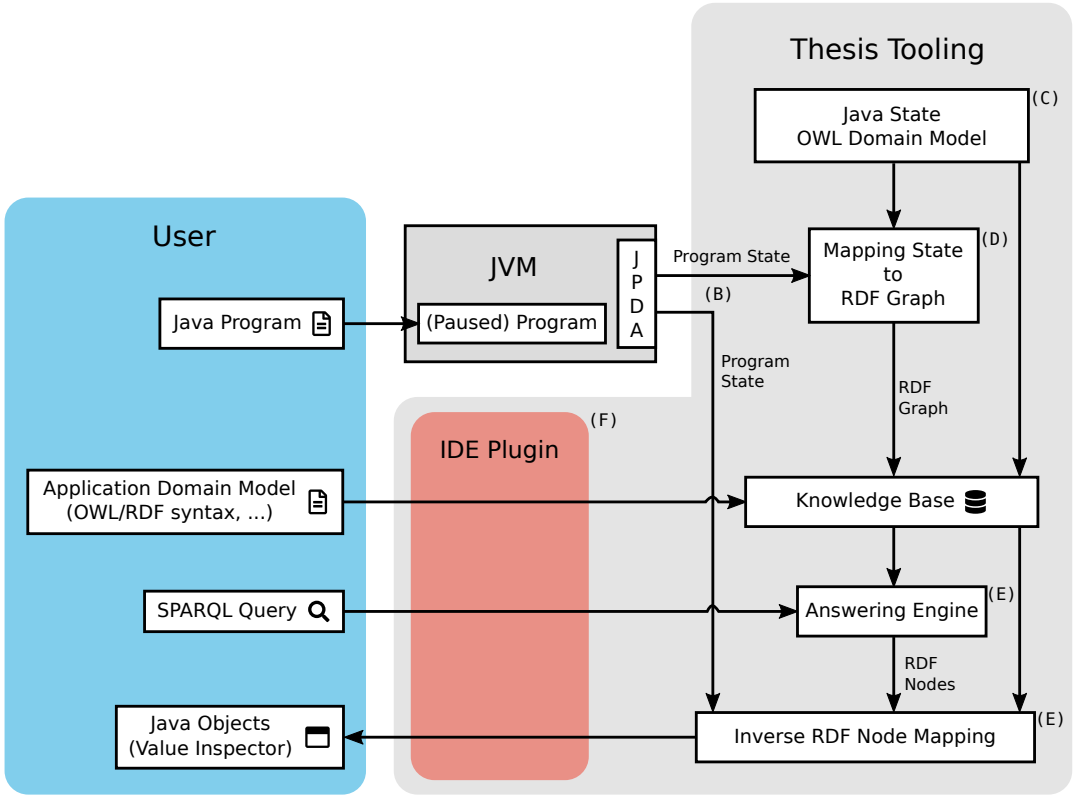


Figure 1: Informal illustration of a typical debugging workflow showcasing the different components of the tooling that is to be developed. Where applicable, the identifying letter of a work package has been annotated to those components it is responsible for. Icons by Font Awesome [5], licensed under CC BY 4.0 license [4].

List of packages

A: Familiarization with Semantic Web Technologies The thesis project will need to integrate many established Semantic Web technologies.

This includes…

The *Resource Description Framework (RDF)* A language for describing structured information as triples. These triples relate subjects and objects via a predicate and can form an *RDF graph*.

As one part of the thesis, JVM program states will need to be mapped to RDF graphs. This way, the program state can be processed by other Semantic Web technologies.

The *SPARQL Protocol And RDF Query Language* A language which allows to query RDF graphs. In particular the thesis aims to allow users to inspect JVM states during debugging by issueing SPARQL queries.

The *Shapes Constraint Language (SHACL)* SHACL shapes allow to formulate constraints on RDF graphs. For example that certain resource types must have certain properties, the types of those properties, range constraints on the values of these properties etc. Existing Semantic Web Frameworks like Apache Jena allow to automatically validate RDF data against those constraints. SHACL Shapes have a potential use in a debugging scenario as they permit developers to verify that program states do not violate their expected format or to detect such violations.

Thus, SHACL support may be implemented in addition to SPARQL query support.

The *Web Ontology Language OWL* With OWL, *ontologies* can be formalized. That is, a description of a subject area, its concepts, their properties and relations.

For example, as part of the thesis, an OWL domain model for Java program execution states will be developed. It shall describe the different entities and ideas of an (executed) Java program (classes, objects, methods, …) and their relationships (e.g. objects are instances of classes). Such an OWL model is needed to give RDF graphs of programs semantics in the Java domain. This is similar to the OWL model for the SMOL language of [8, Fig. 2].

Furthermore, user-provided OWL classes can further infer application domain concepts from the Java OWL model. I.e. users may define how classes and objects form a Simple Graph, its vertices and edges.

The *Apache Jena Java Framework for Semantic Web Applications* Apache Jena allows to leverage the aforementioned Semantic Web technologies (reading / writing RDF graphs, performing SPARQL queries, etc.).

Usage of Apache Jena will accelerate this project as Semantic Web technologies do not need to be re-implemented.

(...potentially more relevant technologies...)

The Semantic Web and its technologies are a complex and rich topic. Thus, getting acquainted with these technologies and becoming able to use them is a critical component of this thesis.

This work package includes:

- Getting familiar with the above Semantic Web technologies. That is, reading relevant introductory literature and experimenting with tooling.
- Summarizing and explaining those Semantic Web concepts that are vital for understanding the thesis to the reader.

B: Pausing Java program execution & Accessing JVM State A core goal of the thesis is to debug Java programs. For this, the developed tool must be able to halt the execution of a Java program and be able to inspect its state.

Via the *Java Debugger Architecture (JPDA)* [11], and in particular its *Java Debug Interface (JDI)* component one can

- set breakpoints
- step through a program
- inspect the program state
- …

For example, the VirtualMachine interface [12] of the JDI can list all ReferenceTypes, e.g. all classes, their fields and values etc. It also permits access to all threads, their StackFrames, variables and values.

This work package includes:

- Getting familiar with the JPDA
- Implementing a proof-of-concept program capable of pausing other Java programs and extracting the most vital JVM state information (e.g. classes, methods, objects, …)

C: An OWL model for Java program states As mentioned in work package (A) and before, an RDF graph must be extracted from a JVM state as provided by package (B).

However, to be able to meaningfully construct such a graph, it must be given semantics in the domain of execution states of Java programs.

Thus, a suitable OWL model of this domain must first be developed. There already exists work on ontologies for the knowledge domain of the Java Programming Language, e.g. [9, 1]. This work may be adapted to develop a new ontology suited for this thesis. A particular challenge is to develop the ontology such that it not only describes static source code, but also captures concepts of a Java program paused during execution. On the other hand, Java is a very rich language and describing every one of its features would not fit into the time constraints of this thesis.

Thus, the developed OWL model should be restricted to a core set of features suitable to implementing the most basic semantic debugging features.

This work package includes:

- Researching existing ontologies for the knowledge domain of the Java language. This includes identification of the methodology by which ontologies for object-oriented programming languages are created in general.
- Determining the subset of core domain concepts relevant to this thesis.
- Developing an ontology encompassing these core concepts.
- Formalizing it in OWL.

D: Mapping Java program states to RDF knowledge graphs As outlined before, to apply any sort of Semantic Web tooling to the the state of a Java program, it must first be converted to an RDF graph, a format commonly understood by such tooling.

More specifically, a map must be developed which maps Java program states to an *ABox*. An *ABox* is a notion from *description logics* on which OWL is built. It is a collection of assertions describing to what concepts a certain object belongs and what roles and relationships they assume within a domain.

It may for example describe, that a certain object belongs to a certain class. And it may say that that class has a specific field etc. Thus, the output of this mapping relies on the OWL domain model developed in work package (C).

On the other hand, to properly define such a mapping, a formalization of the Java states retrieved through work package (B) may also be needed. There exist (partial) formalizations of the Java language, some of which explicitly model intermediate program states (e.g. [15, 2]). Some of these state models may suitable to be adapted for this thesis.

Ultimately, a model of Java states must be worked out which...

- ...can preserve the high-level view of the program state provided by the JDI (see work package (B)). That is, a state model that is aware of stack frames, classes, fields etc. is more useful for constructing an RDF graph than a store of memory addresses and primitive values.
- ...restricts itself to core concepts of Java. Capturing all Java features is outside the scope of this thesis.

This work package includes:

- Research existing models of Java program states.
- Evaluate which features of such a model are essential to implement a mapping as described.
- Develop a suitable model of Java program states. Implement subroutines to automatically extract instances of this model from the JDI.
- Implement the mapping from program states to *ABox*s

E: Query Answering Engine A central feature of the debugger developed in this thesis is the ability to perform queries on the program state using SPARQL. Moreover, these queries should be able to ask about concepts from the Java ontology of work package (C), as well as ask about concepts from a user-provided application domain.

To realize this feature, an *answering engine* must be implemented. An answering engine is a function which takes a knowledge base as well as a query as input, and yields all objects and entities in the knowledge base which satisfy the query.

In the context of this thesis, the knowledge base consists of up to three components:

- The set of OWL axioms defining the Java state ontology (work package (C)).
- Optionally, a set of user-provided OWL axioms describing the application domain.
- The RDF graph resulting form work package (D)

The query shall be provided in the SPARQL language. This allows us to utilize the existing ARQ query engine [7] of Apache Jena which can answer the SPARQL query given the above input information.

The query engine will return nodes from the RDF graph satisfying the query. However, just the RDF representation of those nodes may not be sufficient to effectively debug an application since the user may need to inspect the corresponding Java objects or constructs from which the RDF nodes have been created. Hence, part of this work package is also to extend the mapping of package D to be (partially) reversible. That is, it should be at least possible for RDF nodes representing Java objects to retrieve those objects from the JVM state.

This work package includes:

- Combine the results of the previous work packages and Apache Jena to implement an answering engine as described.
- Extend the mapping of work package (D) to be (partially) reversible as described.

F: IDE Integration Integrated Development Environments (IDEs) for Java like *IntelliJ IDEA* [13] already include “traditional debuggers” which allow to directly inspect objects and values in a paused program state.

This thesis aims to provide a new debugger for a mainstream programming language. In this spirit, it should also integrate well with mainstream tooling of that language and be readily available for use in common IDEs.

For this work package, the thesis tooling as described by the previous work packages shall be integrated into an IDE as a plugin. More specifically, the plugin to be developed shall integrate a *Read-Evaluate-Print-Loop (REPL)* into the UI of the IDE. Also, it should allow users to supply their own application domain definitions, e.g. as files defining an ontology in RDF/OWL syntax.

The REPL will be available during debugging when execution is paused via breakpoint etc. It shall allow the user to input a SPARQL query. If the user inputs a query, the current state of the paused program shall be mapped to an RDF graph using the results from package (D). Then, the query and graph are to be processed by the answering engine of work package (E). Using the reverse mapping of work package (E), all Java objects corresponding to the results of the answering engine shall be retrieved.

These objects shall then be displayed as a manually inspectable tree view, similar to those known from the value inspectors of traditional debuggers.

Optionally, a visualization or inspection tool for the RDF graph itself may also be implemented.

Note, that the integration of the plugin with the existing debugging system of an IDE might pose a particular technical challenge since it may not be possible to attach multiple JPDA debuggers to the same Java process. That is, the debugger built into the IDE and the debugger from work package (B).

If it is not possible to attach multiple debuggers, there are two possible ways to resolve this problem:

1. Either the plugin must not be integrated with the IDE debugger and act as a completely separate feature. This, however, may lessen the user-experience since users may not be able to seamlessly switch between traditional and semantic debugging without restarting the program to be debugged.
2. Or, the implementation of work package (B) must be rewritten not to directly leverage the JPDA but to extract all necessary information from the IDE debugger.

   This approach comes with two disadvantages. For one, the thesis tooling will then be tightly coupled to the API of a particular IDE. Secondly, the interface of the built-in IDE debugger may not be well-documented. For instance, the documentation of the *IntelliJ Platform Plugin SDK* [14] does not directly document the debugger. There is only a link to the relevant source code (XDebuggerManager) in the list of extension points.

The drawbacks of either approach must be evaluated and one of them be implemented.

This work package includes:

- Learn about the plugin API of a popular Java IDE, e.g. IntelliJ Idea.
- Implement the UI as described.
- Evaluate the possible means of integrating the semantic debugger with the existing debugger interface of the IDE as discussed above.
- Integrate the results of the previous work packages into the plugin as described.

G: Case Study – Debugging Example An illustrative example shall be worked out that demonstrates the capabilities of the tooling that is developed in this thesis.

That is, an easy to understand but not entirely trivial Java application must be written. Ideally, the implementation should slightly differ from the application domain and allow to represent invalid states.

For example, Kamburjan et al. provide an implementation of the *2-3 tree* data structure in [8] for the SMOL language. It syntactically permits the instantiation of invalid tree nodes. This is due to efficiency optimizations of the implementation. Kamburjan et al. then argue that their implementation of semantic debugging tooling for SMOL can help to debug such violations.

Similarly, the example application to be developed for this work package should also be faulty and produce a state violating the concepts of the application domain. Then, in a case study, it shall be demonstrated how the tooling from the previous work packages can be applied to fix the fault (e.g. querying the RDF graph of a faulty state etc.).

This work package includes:

- Implement a suitable example of a faulty Java application as described.
- Document the debugging process in a case study.

H: Case Study – Performance An example application shall be implemented which features a data structure of scalable size n. If the application from work package (G) already includes such a data structure, it may be reused.

Then, a case study evaluating the performance of the thesis tooling shall be conducted for increasing n. Here, multiple different metrics may be relevant to assess the performance:

- Execution time and memory requirements for computing the RDF graph from a Java state.
- Execution time and memory requirements for running SPARQL queries of differing complexity on the RDF graphs.
- Execution time and memory requirements for computing increasingly complex inferences via the Apache Jena reasoner on the generated RDF graphs.
- …

This work package includes:

- A Java implementation of a data structure scalable in size as described. (If it can not be adapted from work package (G).)
- Developing a suitable formalization of the application domain.
- Developing suitable SPARQL queries and inferences for the Jena reasoner for a performance evaluation.
- Conducting an performance evaluation as described.

3 Time Schedule

3.1 Available Time

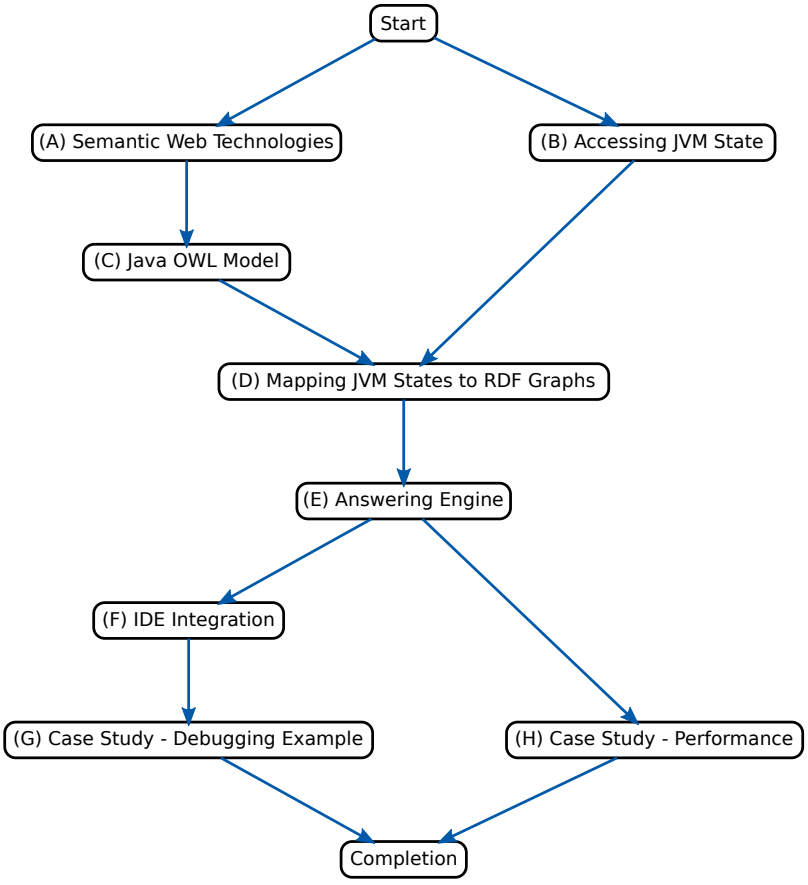
The M. Sc. Computer Science course at TU darmstadt allocates 30 credit points for the master thesis and it must be written within 26 weeks. At TU Darmstadt, a credit point translates to roughly 30 hours of work.

Hence, this thesis must be realized within $30\text{CP} \cdot 30\frac{\text{h}}{\text{CP}} = 900\text{h}$ of work.

Therefore, when assuming an even distribution of the work, this corresponds to about $900\text{h}/26\text{w} \approx 34.6$ hours of work per week.

3.2 Dependencies

The work packages of section 2 partially depend on each other. These dependencies also dictate the order in which they can be implemented. The following diagram depicts those dependencies:



Based on this system of dependencies, the following order has been decided for implementing the work packages:

1. (A) Semantic Web Technologies
2. (B) Accessing JVM State
3. (C) Java OWL Model
4. (D) Mapping JVM States to RDF Graphs
5. (E) Answering Engine
6. (F) IDE Integration
7. (G) Case Study – Debugging Example

8. (H) Case Study – Performance

Even though (C) does not necessarily depend on (B), (B) should be implemented beforehand: The knowledge about the JVM state data accessible through the JPDA might be useful for developing an appropriate Java State OWL model.

Since the example of case study (G) may be reusable for the case study (H), (G) should also be realized before (H).

3.3 Schedule

Available time: 26 weeks. The following is an approximate schedule for implementing the work packages:

Work Package	Alotted Time	Due Date
(A) Semantic Web Technologies	2 weeks	2021-11-14
(B) Accessing JVM State	2 weeks	2021-11-28
(C) Java OWL Model	5 weeks	2022-01-02
(D) Mapping JVM States to RDF Graphs	5 weeks	2022-02-06
(E) Answering Engine	4 weeks	2022-03-06
(F) IDE Integration	2 weeks	2022-03-20
(G) Case Study – Debugging Example	2 weeks	2022-04-03
(H) Case Study – Performance	2 weeks	2022-04-17
Thesis Finalization	2 weeks	2022-05-01

Parallel to the master thesis, some additional workloads have to be taken into account: A seminar course, an intermediate presentation on the thesis results, and one exam (no course attached):

- Seminar Presentation: Likely around 1st December. This is around 4 weeks after the start of the thesis.
- Intermediate Thesis Presentation. After around half of the alotted time. Therefore around 13 weeks after the start of the thesis which is at the start of January.
- Exam: Likely around mid-February which is around 15 weeks after the start of the thesis.
- Seminar Paper Submission: Around 20th February. This is around 16 weeks after the start of the thesis.

With this additional workload in mind, the time allotted has been slightly increased for those work packages to be implemented at the same time as the above dates.

References

[1] Mattia Atzeni and Maurizio Atzori. "CodeOntology: RDF-ization of Source Code". In: *The Semantic Web - ISWC 2017 - 16th International Semantic Web Conference, Vienna, Austria, October 21-25, 2017, Proceedings, Part II*. Ed. by Claudia d'Amato et al. Vol. 10588. Lecture Notes in Computer Science. Springer, 2017, pp. 20–28. DOI: 10.1007/978-3-319-68204-4_2. URL: https://doi.org/10.1007/978-3-319-68204-4%5C_2.

[2] Denis Bogdanas and Grigore Rosu. "K-Java: A Complete Semantics of Java". In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. Ed. by Sriram K. Rajamani and David Walker. ACM, 2015, pp. 445–456. DOI: 10.1145/2676726.2676982. URL: https://doi.org/10.1145/2676726.2676982.

[3] World Wide Web Consortium. *Building the Web of Data*. 2013. URL: https://www.w3.org/2013/data/ (visited on 10/20/2021).

[4] Font Awesome Contributors. *Font Awesome Free License*. 2021. URL: https://raw.githubusercontent.com/FortAwesome/Font-Awesome/master/LICENSE.txt (visited on 10/20/2021).

[5] Inc. Fonticons. *Font Awesome*. 2021. URL: https://fontawesome.com/ (visited on 10/20/2021).

[6] The Apache Software Foundation. *Apache Jena – A free and open source Java framework for building Semantic Web and Linked Data applications.* 2021. URL: https://jena.apache.org/ (visited on 10/20/2021).

[7] The Apache Software Foundation. *ARQ - A SPARQL Processor for Jena*. 2021. URL: https://jena.apache.org/documentation/query/index.html (visited on 10/20/2021).

[8] Eduard Kamburjan et al. “Programming and Debugging with Semantically Lifted States”. In: *The Semantic Web - 18th International Conference, ESWC 2021, Virtual Event, June 6-10, 2021, Proceedings*. Ed. by Ruben Verborgh et al. Vol. 12731. Lecture Notes in Computer Science. Springer, 2021, pp. 126–142. DOI: 10.1007/978-3-030-77385-4_8. URL: https://doi.org/10.1007/978-3-030-77385-4%5C_8.

[9] Aggeliki Kouneli et al. “Modeling the Knowledge Domain of the Java Programming Language as an Ontology”. In: *Advances in Web-Based Learning - ICWL 2012 - 11th International Conference, Sinaia, Romania, September 2-4, 2012. Proceedings*. Ed. by Elvira Popescu et al. Vol. 7558. Lecture Notes in Computer Science. Springer, 2012, pp. 152–159. DOI: 10.1007/978-3-642-33642-3_16. URL: https://doi.org/10.1007/978-3-642-33642-3%5C_16.

[10] Martin Mann, Heinz Ekker, and Christoph Flamm. “The Graph Grammar Library - A Generic Framework for Chemical Graph Rewrite Systems”. In: *Theory and Practice of Model Transformations - 6th International Conference, ICMT@STAF 2013, Budapest, Hungary, June 18-19, 2013. Proceedings*. Ed. by Keith Duddy and Gerti Kappel. Vol. 7909. Lecture Notes in Computer Science. Springer, 2013, pp. 52–53. DOI: 10.1007/978-3-642-38883-5_5. URL: https://doi.org/10.1007/978-3-642-38883-5%5C_5.

[11] Oracle and its affiliates. *Java Platform Debugger Architecture (JPDA)*. 2021. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/jpda/ (visited on 10/20/2021).

[12] Oracle and its affiliates. *JDI Interface VirtualMachine*. 2021. URL: https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/com/sun/jdi/VirtualMachine.html (visited on 10/20/2021).

[13] JetBrains s.r.o. *IntelliJ IDEA - Capable and Ergonomic IDE for JVM*. 2021. URL: https://www.jetbrains.com/idea/ (visited on 10/20/2021).

[14] JetBrains s.r.o. *IntelliJ Platform SDK Documentation*. 2021. URL: https://plugins.jetbrains.com/docs/intellij/welcome.html (visited on 10/20/2021).

[15] Samuel da Silva Feitosa, Rodrigo Geraldo Ribeiro, and André Rauber Du Bois. “Formal Semantics for Java-like Languages and Research Opportunities”. In: *RITA* 25.3 (2018), pp. 62–74. DOI: 10.22456/2175-2745.80912. URL: https://doi.org/10.22456/2175-2745.80912.

[16] Maneesh K. Yadav, Brian P. Kelley, and Steven M. Silverman. “The Potential of a Chemical Graph Transformation System”. In: *Graph Transformations, Second International Conference, ICGT 2004, Rome, Italy, September 28 - October 2, 2004, Proceedings*. Ed. by Hartmut Ehrig et al. Vol. 3256. Lecture Notes in Computer Science. Springer, 2004, pp. 83–95. DOI: 10.1007/978-3-540-30203-2_8. URL: https://doi.org/10.1007/978-3-540-30203-2%5C_8.