Towards debugging facilities for graphical modeling languages in web-based modeling tools

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Business Informatics

eingereicht von

Hansjörg Eder, BSc
Matrikelnummer 01426985

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: O.Univ.Prof. Dipl.-Ing. Mag. Dr.techn. Gertrude Kappel
Mitwirkung: Dipl.-Ing. Dr.techn. Philip Langer

Wien, 10. Dezember 2020

Hansjörg Eder | Gertrude Kappel

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Towards debugging facilities for graphical modeling languages in web-based modeling tools

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Business Informatics

by

Hansjörg Eder, BSc
Registration Number 01426985

to the Faculty of Informatics
at the TU Wien

Advisor: O.Univ.Prof. Dipl.-Ing. Mag. Dr.techn. Gertrude Kappel
Assistance: Dipl.-Ing. Dr.techn. Philip Langer

Vienna, 10th December, 2020

Hansjörg Eder   Gertrude Kappel

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Erklärung zur Verfassung der Arbeit

Hansjörg Eder, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 10. Dezember 2020

Hansjörg Eder

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Acknowledgements

First of all, I would like to thank my family, without whose motivating and financial support this study would not have been possible.

Moreover, I would like to thank my fellow students with whom I worked for hours on exercises and also spent many fun evenings.

I would also like to thank my friends, who have been with me for years. Although I had little time for activities, I could always rely on their support.

I would like to thank O.Univ.Prof. Dr. Gertrude Kappel and Dr. Tanja Mayerhofer, who sparked my interest in Model Engineering during my Master's studies.

I would especially like to thank Dr. Philip Langer, without whose constant support and valuable feedback, this work would not have been possible. Thank you for encouraging me to continue with this thesis.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Kurzfassung

Die modellgetriebene Softwareentwicklung bietet Vorteile im Entwicklungsprozess, indem sie die Abstraktionsebene erhöht und damit die Komplexität einer bestimmten Domäne reduziert. Domänenspezifische Sprachen (DSSs) erhöhen die Produktivität für Entwickler und verbessern die Kommunikation mit Domänenexperten. DSSs können in textuelle Sprachen (TS) und grafische Sprachen (GS) unterteilt werden. Die für die Erzeugung von DSSs erforderlichen Tools und Editoren sind in der Regel tief in eine spezifische Entwicklungsumgebung (IDE) integriert. Diese IDEs verwenden verschiedene Programmierschnittstellen, um die gleichen Funktionen für verschiedene Sprachen zu implementieren, was zu mehreren Implementierungen für eine IDE führt. Zu diesem Zweck wurde das Language Server Protocol (LSP) für TS und das Graphical Language Server Protocol (GLSP) für GS eingeführt. Diese Protokolle trennen den Editor von den Sprachkonzepten und ermöglichen so die Wiederverwendung eines Sprachservers über mehrere IDEs.

Die zunehmende Komplexität in der Softwareentwicklung führt zu einem erhöhten Auftreten von Fehlern in Programmen und Modellen und erfordert daher Debugging-Möglichkeiten sowohl für TS als auch für GS. Daher wurde das Debug Adapter Protocol (DAP) eingeführt, um die Kommunikation zwischen der IDE und einem konkreten Debugger zu standardisieren. Vergleichbar mit den Zielen des LSP und GLSP beabsichtigt das DAP eine generische grafische Benutzeroberfläche pro IDE zu verwenden und dieselben sprachspezifischen Debugger über mehrere IDEs hinweg wiederzuverwenden.

Diese Arbeit analysiert wie das DAP für TS und das GLSP für GS kombiniert werden können, um Modelldebugging in einer webbasierten Umgebung zu unterstützen. Des Weiteren wird evaluiert, ob die bekannten Debugging-Konzepte zum Debuggen von TS auf GS übertragen werden können. Diese Arbeit zielt vor allem darauf ab zu untersuchen, ob das DAP auf GS angewendet werden kann. Darüber hinaus beabsichtigt diese Arbeit, bestehende Debugging-Komponenten und -Frameworks, die in Bezug auf TS entwickelt wurden, wiederzuverwenden. Die Ergebnisse dieser Arbeit werden anhand von zwei Anwendungsfällen ausgewertet. Der erste Fall zielt darauf ab zu untersuchen, ob das DAP für TS auf GS wiederverwendet werden kann. Der zweite Anwendungsfall zielt darauf ab, die Wiederverwendbarkeit des entwickelten Frameworks im Hinblick auf weitere GS und Domänenprobleme zu bewerten. Die Ergebnisse der Fallstudie zeigen, dass das DAP eine effiziente Multi-Editor-Integration ermöglicht. Darüber hinaus zeigen die Ergebnisse, dass das entwickelte Debugging-Framework die Anforderungen eines modernen Debuggers erfüllt und die Integration zusätzlicher Debugger für weitere Sprachen erleichtert.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Abstract

Model-driven development (MDD) offers advantages in the development process by raising the level of abstraction and reducing the complexity of a specific domain. Domain-Specific Languages (DSLs) used in MDD raise the productivity for developers and improve communication with domain experts. DSLs can be divided into textual languages (TLs) and graphical languages (GLs). IDEs provide different application programming interfaces (APIs) for developing tool support for various languages. If tool support for a language is required in multiple IDEs, the implementation of the same language has to be repeated for each IDE based on their different APIs. For this purpose, the Language Server Protocol (LSP) for TLs separates the editor interface from the language logic and allows the reuse of one language server implementing the language logic across several LSP-based IDEs. Like the LSP, the Graphical Language Server Protocol (GLSP) was invented to transfer the advantages of extensibility and reusability to GLs.

Increased complexity in software development leads to an increased number of errors in programs and models and, therefore, it requires debugging facilities for both TLs and GLs. Therefore, the Debug Adapter Protocol (DAP) was invented to standardize the communication between the IDE and a concrete debugger. The DAP intends that an IDE offers one generic graphical user interface for debugging functionality and that there is one debugger per language implementing the language-specific debug logic that can then be reused among IDEs.

This thesis analyzes a way of combining the DAP for TLs and the GLSP for GLs to support model debugging in a web-based environment. Furthermore, it is evaluated whether the well-known debugging concepts for debugging source code can be transferred to GLs. The thesis intends to reuse existing debugging components and frameworks developed for TLs. The results of this work are evaluated in two use cases. The first use case is the running example and aims at investigating whether the DAP for TLs can be reused for GLs. The second use case intends to evaluate the reusability of the developed framework concerning further GLs and domain problems. The case study's results indicate that the DAP enables efficient multi-editor integration of debuggers, i.e., one language-specific debugger can efficiently be integrated with a DAP-based debugging interface and that a DAP-based debugging interface can efficiently integrate with multiple language-specific debuggers. Further, the results show that the developed debugging framework meets the requirements of a modern debugger and facilitates the integration of debugging support for further GLs.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Contents

Kurzfassung **ix**

Abstract **xi**

Contents **xiii**

1 Introduction **1**
1.1 Motivation 1
1.2 Problem Statement 2
1.3 Aim of the Work 3
1.4 Methodology 4
1.5 Structure of the Work 4

2 Background **7**
2.1 Model-Driven Development 7
2.2 Domain-Specific Languages 12
2.3 Debugging in Domain-Specific Modeling 19

3 Web-based Modeling Tools **25**
3.1 Language Server Protocol 25
3.2 Graphical Language Server Protocol 29
3.3 Debug Adapter Protocol 34
3.4 Summary 38

4 Applying the DAP to GLSP **41**
4.1 Running Example: Workflow Modeling Language + Diagram Editor . . 41
4.2 Requirements for Debugging the Workflow Modeling Language 47
4.3 Workflow Modeling Language Debugger 56

5 Architecture and Implementation **65**
5.1 Technological Background 65
5.2 Implementation Details 69
5.3 Challenges 76

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

6 Evaluation **81**
6.1 State Machine Modeling Language 81
6.2 Open Problems 94
6.3 Interpretation of the Results 95

7 Related Work **99**
7.1 Debugging Domain-Specific Models 99
7.2 Web-based DSML Environments 103
7.3 Other Related Work 103

8 Summary and Conclusion **105**
8.1 Summary 105
8.2 Comparison with Related Work 106
8.3 Limitations and Future Work 107

Bibliography **111**

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 1

Introduction

1.1 Motivation

The progress of technology leads to the increasing complexity of software components to ensure the functionality of modern systems. Model-driven software development (MDSD) offers advantages in the development process by raising the level of abstraction and reducing the complexity of a specific domain. Modeling languages are an essential part of MDSD and let users design the models for their systems. Domain-Specific (Modeling) Languages (DS(M)Ls) are languages that are explicitly defined for a specific domain and are represented through a concrete syntax. The concrete syntax can be further divided into textual concrete syntax and graphical concrete syntax. The advantages of the respective notation must be determined in relation to the concept used. If the focus lies on the relationships between objects, graphical languages should be preferred [1]. Although Integrated Development Environments (IDEs) of these DSMLs provide some sort of check and validation on modeled systems according to the related DSML's syntax and semantics descriptions, they do not have a built-in support for debugging these models [2]. That deficiency causes the model designers not to be sure on the correctness of their created models during the design phase. The model first has to be transformed into a general-purpose language using code generation. The generated source code can then be viewed with existing development environments and checked for errors utilizing the integrated debuggers. However, the generation process takes significant effort and time. Therefore, it should be possible to check and validate the created model already during the design phase. Thus, only correct models are used in the code generation process.

1.2 Problem Statement

Current development tools provide different application programming interfaces (APIs) for developing tool support for various languages. If tool support for a language is required in multiple IDEs, the implementation of the same language has to be repeated for each IDE based on their different APIs. However, this limits the extensibility and reusability of existing components for other languages. In this context, the Language Server Protocol[1] (LSP) for textual languages separates the editor interface from the language logic and allows the reuse of one language server implementing the language logic across several LSP-based IDEs.

These limitations regarding extensibility and reusability not only relate to textual languages, but also graphical languages. Since graphical languages were not considered in the LSP from the beginning, attempts by members within the Eclipse Community[2] to extend the LSP directly to include graphical modeling turned out to be impractical [3]. Instead, a dedicated protocol for graphical languages - the Graphical Language Server Protocol[3] (GLSP) - was invented to communicate model editing operations between the diagram editor and the graphical language server.
The use of a client-server architecture not only reduces the effort to integrate new languages with an editor but also supports the trend towards cloud-based development. Thus, the editor does not have to be installed locally and can be kept platform-independent. Through the shift to cloud-based solutions, modern web technologies such as HTML5[4] and CSS[5] are used, which offer new possibilities for graphical modeling and debugging approaches. Besides structuring the graphical user interface and integrating animations during the debugging procedure, it further enables cloud-based collaborative development.

Increasing the complexity in software programs leads to an increased occurrence of errors in programs, and therefore it requires debugging facilities for both textual and graphical languages. In current development tools, because of the different APIs used by each tool, implementing a new debugger involves significant effort. This issue led to the introduction of the Debug Adapter Protocol[6] (DAP) for textual languages to standardize the communication between a development tool and a debugger. The protocol enables to implement only one generic debugger UI per development tool and reuse the same language-specific debugger across several development tools.

It is obvious that there is a need to provide debugging facilities also for graphical modeling languages. The question arises whether the DAP defined for textual languages can also be used for graphical languages or a new protocol is required.

[1] https://microsoft.github.io/language-server-protocol
[2] https://eclipse.org/community
[3] https://github.com/eclipsesource/GraphicalServerProtocol
[4] https://w3schools.com/html
[5] https://projects.eclipse.org/projects/ecd.sprotty
[6] https://microsoft.github.io/debug-adapter-protocol

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

1.3 Aim of the Work

The work aims to provide a visual debugger for a web-based graphical modeling editor, which is based on the DAP and supports a user during the development of software components in a specific domain. As part of this it has to be analyzed, whether the DAP in its current version is suitable for graphical languages or an extension of the protocol is required. Therefore, the requirements for debugging graphical modeling languages are derived from an existing DSML, and potential inconsistencies in the DAP are dissolved. Furthermore, it must be analyzed whether the existing components and frameworks used by debuggers for textual languages, can be reused for graphical modeling languages. Subsequently, a generic debugger capable of core debugging functionalities will be designed. The debugger then is integrated into an existing GLSP client built on top of the Graphical Language Server (GLS) platform[7]. To test the generic debugger user interface, we require a language-specific debugger that resides on a remote server and a language-specific debug adapter that connects the development tool to the language-specific debugger. Figure 1.1 illustrates the interaction between the components, as mentioned above.

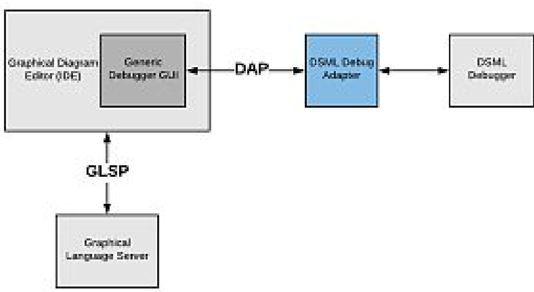

Figure 1.1: A representation of how the individual components interact with each other (adopted from [4]).

Finally, the following research questions will be answered:

RQ1: What should a debug adapter protocol look like to meet the requirements of graphical modeling languages?

RQ2: Can the DAP for textual languages be applied to graphical model debugging, and if not, what needs to be extended or changed?

RQ3: What generic client-frameworks and user interface/debug adapter client components may exist for such a debug protocol that can be reused across debugging use cases for multiple graphical domain-specific languages?

The main questions (RQ1 and RQ2) are concerned with answering how debugging concepts for textual languages can be applied to graphical modeling languages. RQ2

[7] https://github.com/eclipsesource/graphical-lsp

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

analyzes the existing DAP to show whether it can be used for graphical modeling languages. The question is answered by defining different DSMLs and deriving the requirements from those languages to implement a debugger. Based on these requirements, we can make a statement if the DAP fulfills the purposes or needs to be extended. For the implementation of the debugger, we use existing components and frameworks which we can extend for the requirements. The RQ3 is about which components that have been developed in the course of the thesis can be generalized so that they may act as a generic basis for developing visual debuggers for any graphical language.

1.4 Methodology

The objective of Design Science research is to build innovative IT artifacts and evaluate them. Since the central part of the thesis is to build an IT artifact that benefits its users, we use Design Science as a methodological approach. A. Hevner et al. present seven guidelines for Design Science in Information Systems Research [5]. Following these guidelines, the two major steps are *design* and *evaluation*.

Design-science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation. In this work, a generic debugger capable of core debugging functionalities will be designed. The debugger should support the DAP or an extension of it to allow easy integration of further graphical modeling languages. The debugger then is integrated into an existing GLSP client built on top of the GLS platform.

A designed artifact is complete and effective when it satisfies the requirements and constraints of the problem it was meant to solve. It can be evaluated [6, 7] in terms of functionality, completeness, consistency, accuracy, performance, reliability, usability, fit with the organization, and other relevant qualities [8]. In this work, the artifact to be created is implemented utilizing an already existing DSML. In the first step, we derive from the DSML the requirements for the debugger and implement them in a further step. For the evaluation, we develop another DSML from which we derive new requirements and evaluate the created artifact. We want to show that the debugger we developed can be reused to integrate further DSMLs easily.

1.5 Structure of the Work

This thesis consists of seven further chapters. Chapter 2 is concerned with model-driven development in general, the introduction into domain-specific languages, and applying debugging concepts to source code, models, and other artifacts. Chapter 3 presents the existing client-server architecture tools this work is based on such as the Language Server Protocol for editing textual languages, the Graphical Language Server Protocol to create models with graphical modeling languages, and the Debug Adapter Protocol for debugging textual languages. Chapter 4 introduces an existing graphical modeling language and subsequently derives the requirements for implementing a debugger for this language. Besides, a debugger implemented for this language and fulfilling the identified

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

requirements will be presented. Chapter 5 discusses the detailed experiences that were gained during the implementation of the debugger. The results of the evaluation based on a newly defined DSML are described in Chapter 6. Related work is presented in Chapter 7. Finally, Chapter 8 concludes by summarizing the presented work.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek, Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek, Your knowledge hub

CHAPTER 2

Background

This chapter presents essential knowledge regarding the development approaches and concepts used in this work. The first section presents the two mechanisms of code generation and model interpretation used in model-driven development. The second section gives an overview of the most important characteristics of domain-specific modeling languages. In the last section, we focus on debugging concepts used in code debugging and how these concepts can be translated into debugging modeling languages.

2.1 Model-Driven Development

In this section, will we introduce the Model-Driven Development (MDD) approach. Furthermore, we discuss the two mechanisms code generation and model interpretation used in MDD.

2.1.1 Introduction to MDD

MDD aims to find domain-specific abstractions and make them accessible through formal modeling. The development of software through MDD consists of creating a high-level representation of the required software facilities and deriving an running application [9–13]. Intermediate steps can be enclosed to enable some kind of user interaction with the generation process [1].

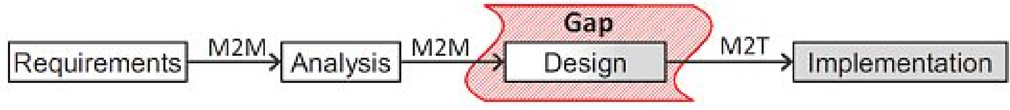

Figure 2.1: A typical MDD-based software development process [1].

Figure 2.1 shows a typical development process, as described in MDD. In each phase of the model development process, models are (semi)automatically generated using a model-to-model transformation (M2M). Finally, the designed models are translated into the final source code utilizing a model-to-text transformation. The example from Figure 2.1 shows the development process using a code generator, while model interpretation skips the implementation phase and executes the model directly [1]. In the further course of this section, we will discuss these two approaches in detail and present the advantages of using the respective tools.

2.1.2 Benefits and Challenges

One of the main advantages of model-driven approaches is bridging the communication gap between requirements/analysis and the final implementation. Looking at the organization level in enterprises, this is a major problem, as there is also a gap between business needs and IT realizations or support. Models are a valid solution as a *lingua franca* among stakeholders from business and IT divisions. Furthermore, models capture and organize the understanding of the application in a way that simplifies the discussion among the actors and eases the integration of new team members into the development process. The benefits of MDD can be summarized as [1].

- Increasing the effectiveness of communication between the involved parties.
- Raising the productivity of the development team due to the automation of the development process.
- Reducing the number of errors in the final code through the automation process.

One disadvantage of MDD is that modeling and programming do not mix very well. Existing General-purpose language (GPL) tools and IDEs cannot be reused for modeling languages without integration issues. Programming languages commonly have a textual concrete syntax, where source code is transformed through scanners and parser into an abstract syntax tree. Modeling languages have either a textual or graphical concrete syntax in which the latter is represented in the form of a diagram. Diagram editors manipulate directly the abstract syntax [14]. Table 2.1 shows the main differences between modeling and programming.

Table 2.1 shows that debuggers rarely exist for DSLs, while a debugger is almost always available for GPLs. As discussed in Section 1.2, it takes significant effort and time to develop a new debugger for a language and to integrate the debugger into an IDE. Therefore, many DSLs are first transformed into GPL code to be able to use an existing GPL debugger. This confirms our work to simplify the integration of new debuggers for graphical modeling languages into web-based IDEs.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

	Modeling	Programming
Define your own notation/language	Easy Sometimes	possible to some extent
Syntactically integrate several languages	Possible, depends on tool	Hard
Graphical notations	Possible, depends on tool	Usually only visualizations
Customize generator/compiler	Easy	Sometimes possible based on open compilers
Navigate/query	Easy	Sometimes possible, depends on IDE and APIs
View Support	Typical	Almost never
Constraints	Easy	Sometimes possible with Findbugs etc.
Sophisticated mature IDE	Sometimes, effort-dependent	Standard
Debugger	Rarely	Almost always
Versioning, diff/merge	Depends on syntax and tools	Standard

Table 2.1: Main differences between Modeling and Programming [14].

2.1.3 Code Generation and Model Interpretation

A model is an executable model [15] when its operational semantics are fully specified. In practice, not all models have to be executable since the executability depends more on the adopted execution engine than on the designed model itself. On the one hand, some models are not entirely specified, but there exist execution tools that can fill the gap to execute them. On the other hand, there might be very complex and fully specified models, that cannot be executed because an execution engine [16–19] is missing [1]. However, there exists also modeling languages (e.g. class diagram) that are not executable. These modeling languages are converted to GPL code by code generation and executed with the GPL execution engine.

In order to execute models, there exist two different alternative approaches: code generation and model interpretation.

Code Generation

Code generators transform the models into executable code for interpretation or compilation in an automatic way [20,21]. This process is similar to what is known from compilers, and in this sense, code generators are referred to as *model compilers*. Generators can be classified into declarative and operational or a mixture of both. In the declarative approach, semantic mappings between the model's elements and the target language are defined, whereas the operational approach uses rule-based transformations to produce target code step by step out of the source model. Typically the target code is some kind of GPL. After generation, the code is already complete from the perspective of a modeler, and no further manipulation is necessary. Nevertheless, it is possible to view and edit the generated code with an IDE before executing the generated artifact. Figure 2.2 shows the process of generating source code from a set of models [1].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

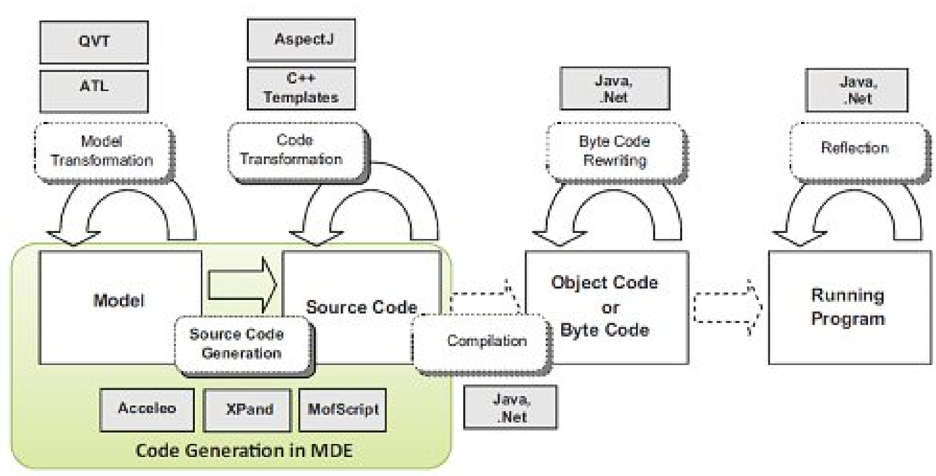

Figure 2.2: Code generation process [1].

In the following, we summarize several benefits of the code generation approach [1].

- Code generation protects the intellectual property of the modeler. By generating the current application, the conceptualization and design of the model are not shared with the customer. The model can be reused and evolved for future projects. The customers only receive the source code of the application.
- The generated code is provided in a standard programming language known to any programmer.
- Customers can choose their runtime environment. As a result, customers are not dependent on technologies used by the vendor of the MDD approach. Since code generation usually produces code in a standard language, the customer is not dependent on the MDD approach in the future, since the source code can be refined manually with available IDEs.
- Existing programming artifacts can be reused with code generation. The existing code can be generalized and used as templates for code generation of new parts of the software.
- In contrast to model interpreters, code generators are easier to maintain, debug, and track. An interpreter has a generic and complex behavior to cover all possible execution cases, whereas code generators typically consist of rule-based transformations.
- Comparing the code generator and model interpretation in terms of execution speed, one can say that a generated application performs better than interpreting the model.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

In addition to the advantages of code generation, there are also disadvantages. One issue within code generation is that the generated source code might not look familiar to the developers. Although the execution behaves as expected, and the code is written in the desired programming language, the code's structure can be completely different from the code the programmer would write. The developers may not feel very confident to edit or evolve the source code in the future. For this reason, Marco Brambilla et al. [1] proposed a Turing test for code generation, similar to the classic Turing test for AI[1]. They defined the Turing test for code generation tools as follows.

"A human judge examines the code generated by one programmer and one code generation tool for the same formal specification. If the judge cannot reliably tell the tool from the human, the tool is said to have passed the test." ([1], p. 31)

Code generation tools that pass the Turing test have proved that they can produce code in a way that humans would do and should, therefore, be acceptable to them.

Model Interpretation

Model interpretation uses a generic engine that parses the elements of the model and executes them on the fly with an interpretation approach. Model interpretation can be characterized by the following properties and benefits [1].

- In the case of model interpretation, there is no need to delve into the application's source code. The model replaces the code, and therefore no code exists which could be inspected.
- Modifications on the model can be done during the runtime, if the interpreter supports continuing the execution by parsing the updated version of the model.
- There is no code generation step required after changing the model. This significantly shortens the turnaround time in incremental development approaches.
- The running application's behavior can be changed by simply extending the interpreter while keeping the same models.
- There is no deployment phase required because the designed model is already the executable artifact.
- Compared to code generation, model interpretation [22,23] allows easy debugging [24–26] of models during runtime. Interpreters can process models step by step. With code generation, it requires sophisticated tools because the executable application must be hooked to the model concepts, and the modeling tool must catch events from the running application.

[1] https://en.wikipedia.org/wiki/Turing_test

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Although many users know about the advantages of model interpretation, some of them are not ready to make themselves dependent on the MDD tool vendor [1]. Since the source code is not available, one cannot simply continue executing and evolving the application. Furthermore, performance concerns are often one of the reasons why this approach is discarded [1]. Besides, the users are instructed to install a new platform (model interpreter) in their IT Infrastructure that may be critical in large enterprises due to strict IT architecture policies and security issues [1].

2.2 Domain-Specific Languages

This section introduces the characteristics of Domain-Specific Languages (DSLs). The main ingredients of DSLs, abstract syntax, concrete syntax, and semantics are essential to design a valid DSL and will be introduced in this section. Furthermore, we will present the benefits and challenges and give some examples of DSLs. Please note that in the modeling realm, DSLs are called Domain-Specific Modeling Languages (DSMLs) to emphasize that they are modeling DSLs. However, DSL is by far the most commonly used term, and therefore we will stick to it in the upcoming discussions.

2.2.1 Introduction to DSLs

DSLs raise the level of abstraction beyond programming by defining the solution in a language that uses concepts and rules from a specific problem domain [27]. DSLs are programming languages used by humans to instruct computers to do something. They should be designed in a way that they are easier to read and understand. However, DSLs should also be executable by a computer. The expressiveness comes from expressions and the way they are composed together. DSLs have limited expressiveness and should, therefore, only provide the features needed for the specific domain [27]. Therefore, DSLs are only used to implement a particular aspect of a system instead of building an entire system. Finally, DSLs have a clear focus on a small application domain, which is why they cannot be reused for other domains [27].

General-purpose languages (GPLs) are in contrast to DSLs Turing-complete. This means that anything computable with a Turing machine can be implemented with a GPL. Furthermore, all GPLs are interchangeable. Everything that can be expressed in one language can also be expressed in any other language. Nevertheless, there are also problems in the area of GPLs, where special features are required [28]. Thus, many different GPLs have been developed over the years, e.g. Java, C#, Python. Table 2.2 shows the main differences between GPLs and DSLs. While DSLs and GPLs may have characteristics from both the second and the third column, DSLs are more likely to have characteristics from the third column [14].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

	GPLs	DSLs
Domain	large and complex	smaller and well-defined
Language size	large	small
Turing completeness	always	often not
User-defined abstractions	sophisticated	limited
Execution	via intermediate GPL	native
Lifespan	years to decades	months to years (driven by context)
Designed by	guru or committee	a few engineers and domain experts
User community	large, anonymous and widespread	small, accessible and local
Evolution	slow, often standardized	fast-paced
Deprecation/incompatible changes	almost impossible	feasible

Table 2.2: Domain-specific languages versus general-purpose languages [14].

Fowler identified two main categories of DSLs, namely, *internal* and *external* DSLs [27].

- **Internal DSLs** are usually built on top of a general-purpose language or extending the language. The DSL is written using a particular grammar and only uses a subset of the host language's features to handle one small aspect of an overall system. The grammar is restricted to the legal syntax of its general-purpose language but should give the feeling of a custom language. Due to the same syntax rules, the host language's existing tools can be reused for the internal DSL. Besides, error handling and reporting in the main language can be partially reused for the internal DSL.

- **External DSLs** have their custom syntax and are separated from the main language of the application with which they work. A model of an external DSL, whether textual or graphical, is parsed and then executed by an interpreter or translated into another language, typically a GPL, using a code generator. In contrast to internal DSLs, there exist no utilities that will support the use of the newly created language. One will have to figure out a practical way for common features like parsing, syntax highlighting, exception handling, and reporting.

2.2.2 Benefits and Challenges

In the following, the benefits of DSLs identified by Fowler [27], Voelter et al. [14], and Kelly et al. [29] are listed.

- **Productivity.** A higher level of abstraction makes it easier to read and understand a model and leads to better productivity. Few lines of DSL code replaces a lot of GPL code, thus reducing the overall complexity of a program. This makes it easier to find errors and modify the system. Finally, users of DSLs are not dependent on learning a huge and complex GPL like Java. Instead, they only need to understand a fraction useful in their problem domain.

- **Quality.** DSLs increase not only productivity but also quality in terms of fewer bugs, better architectural conformance, and increased maintainability. This approach is also known as correct-by-construction. If the DSL is designed in the right way, only the construction of correct programs is allowed.

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

- **Validation and Verification.** Since DSLs are semantically richer than GPLs, they are not cluttered with implementation details. DSLs are aware of the domain concepts. Therefore, error messages use more meaningful wording, and analyses are easier to implement. Due to the involvement of domain experts, manual review and validation become more efficient.

- **Involvement of Domain Experts.** On the one hand, developers are not aware of the domain, they communicate with the domain experts to construct a well-designed DSL. On the other hand, domain experts do not have to deal with complex implementation details but can easily create programs. Visualizations or simulations can further facilitate understanding.

- **Productive Tooling.** Many external DSLs come with their custom IDEs that are aware of the language and improve the user experience. Such features include static analyses, code completion, visualizations, debuggers, simulators, and more. This not only makes it easier for new users to learn the language but also increases user productivity.

In addition to benefits, there are also challenges to overcome [14, 27, 29].

- **Language Cacophony.** Using multiple languages in one project is much more complicated than using a single one. For new project members, it is hard to understand what is going on in the system. Many forget that learning a new DSL is much easier through the level of abstraction and takes less effort than learning a GPL. However, it must be considered whether it would be more sensible to learn the underlying model instead of the DSL used.

- **Effort of building the DSLs.** Before using a DSL in a project, one has to build it first. It is important to consider whether the effort and costs involved in generating the DSL are worth it. This must be taken into account in the overall cost-benefit analysis right from the start of the project. For technical DSLs that address aspects such as components, state machines, or persistence mapping of software engineering, this question does not usually arise because they have great potential for reuse. In the meantime, there are modern tools available that significantly reduce the effort required to develop new DSLs. Generally speaking, three factors reduce the cost of DSLs: in-depth knowledge about a domain, experience of the DSL developer, and productivity of the tools.

- **Language Engineering Skills.** As mentioned in the previous point, it takes a certain amount of experience and skill to build a new DSL. Although modern language workbench facilitates the introduction of new languages, there is still a certain learning curve. Building relevant languages and defining the right abstraction level requires experience and practice, which can be gained over time while working with different languages and problem domains.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- **DSL Hell.** The constant simplification to create new DSLs poses another issue. Many developers tend to introduce a new DSL instead of searching for and learning an existing DSL for that specific domain. This ends up in a bunch of incomplete DSLs, each covering common domains but not compatible. One way to avoid this is to create DSLs is such a way so they can be easily extended or used as a starting point for other languages. Features can be easily added to the existing language.

2.2.3 Examples

In this section, we discuss examples of existing DSLs. In particular, the DSLs HQL, BPMN, and WebML are introduced.

Hibernate Query Language (HQL)

HQL[2] is a DSL with a textual syntax that combines object-oriented coding with relational databases. HQL is similar to the Structured Query Language[3] (SQL) syntax, but instead of working with database tables and columns, HQL queries use persistent objects and their properties. The Hibernate O-R Mapper translates the HQL query into a SQL statement that matches the corresponding database type. Listing 2.1 shows an example query in HQL [27].

```
select person from Person person, Calendar calendar
where calendar.holidays['national day'] = person.birthDay
      and person.nationality.calendar = calendar
```

Listing 2.1: HQL example [27].

Business Process Model and Notation (BPMN)

BPMN[4] is a well-known standard for business process modeling that consists of notational structures for designing business processes based on flowcharting techniques. BPMN aims to provide business process modeling for business users and technical users at the same time by displaying both simple and complex systems. The BPMN 2.0 specification[5] enables the execution of business process models as well as the transformation of models into other execution languages, especially into the Business Process Execution Languages[6] (BPEL) [30].

Furthermore, BPMN was designed to provide a simple, readable, and understandable language for all business stakeholders. This might be the business analyst who creates and completes the processes or the technical developer who implements the processes or the business manager who controls and manages the processes. Consequently, BPMN can

[2] https://docs.jboss.org/hibernate/core/3.3/reference/en/html/queryhql.html
[3] https://iso.org/standard/63555.html
[4] http://bpmn.org/
[5] https://omg.org/spec/BPMN/2.0
[6] https://oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

be used to overcome the communication gap, which might occur between business process design and implementation [30]. Figure 2.3 shows a BPMN model, which describes the process of hiring people in a company.

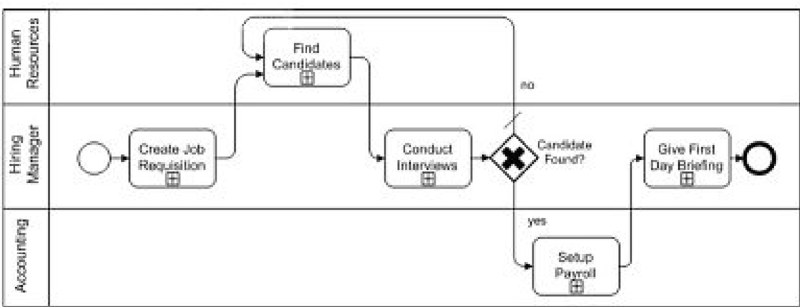

Figure 2.3: BPMN example [1].

Web Modeling Language (WebML)

WebML[7] is a high-level modeling language for hypertext specifications. WebML consists of simple notational constructs that defines a hypertext as a set of pages. The notation consists of control units, operations, and edges, which connect the control units. WebML is strongly oriented towards the notation of well-known conceptual modeling languages like Entity-Relationship[8] (ER) and Unified Modeling Language[9] (UML): every concept has a notational construct, and specifications are diagrams. Therefore, users familiar with UML or ER do not have to worry about learning a new language [31]. Figure 2.4 shows a WebML model that describes the user interface of a website for navigating a product catalog by category.

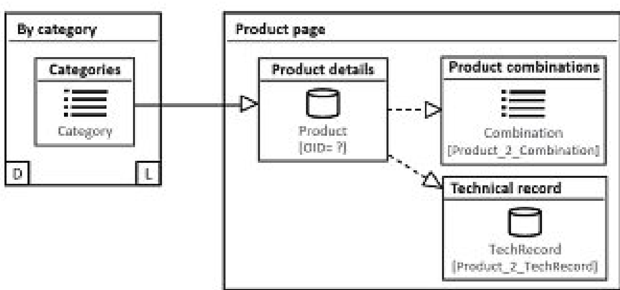

Figure 2.4: WebML example [1].

[7] http://webml.deib.polimi.it/page1.do.html
[8] http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.123.1085
[9] https://uml.org/

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

2.2.4 DSL Language Constructs

In this part of the section, we would like to discuss the components of a DSL, as it is important to understand them when creating a new language. DSLs, like languages in general, consist of syntax and semantics. The syntax is further divided into abstract syntax and concrete syntax, where the abstract syntax addresses the structure and grammatical rules of a language, and the concrete syntax defines the representation of the language to the user. The definition of the language's meaning is referred to as semantics [1, 14, 29, 32, 33]. Figure 2.5 shows the three ingredients of a DSL and their relationship to each other.

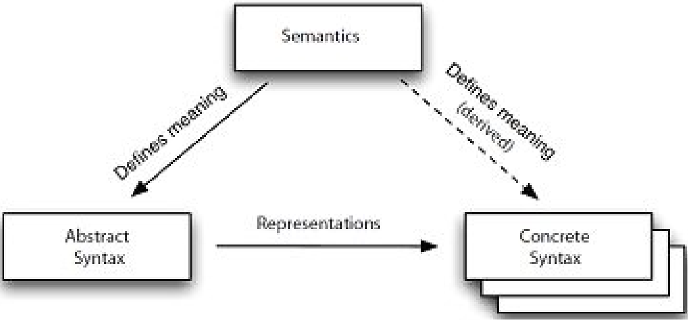

Figure 2.5: The three main ingredients of a modeling language and their relationships [1].

Abstract Syntax

Describes the conceptional structure of a language, its properties, model hierarchy structures, model correctness rules, and how the different elements can be connected. The abstract syntax is specified in the metamodel [34, 35]. The term "meta" is used because the language specification is one level of abstraction higher than the models. Figure 2.6 shows the metamodel of a finite-state automaton. The metamodel consists of the *FSAModel* containing *States* and *Transitions*. A state has the properties *isInitial*, *isFinal*, and *name*. Furthermore, a state has optional outgoing and incoming transitions. Transitions have a property *event* to trigger the transition. A transition has exactly one source state and exactly one target state. These can be two different states or one state that represents both source and target. The syntax of a language also includes grammatical rules (e.g. a model contains zero or up to an undefined amount of states) which are taken from the problem domain. The advantage of defining the rules in the language is that the effort required to check the model for errors is reduced. Considering several code generators exist for one language, the checking would have to be done for each code generator. The grammatical rules determine how models can be constructed: they define the legal values, relationships between concepts, and how certain concepts should be used [1, 29].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

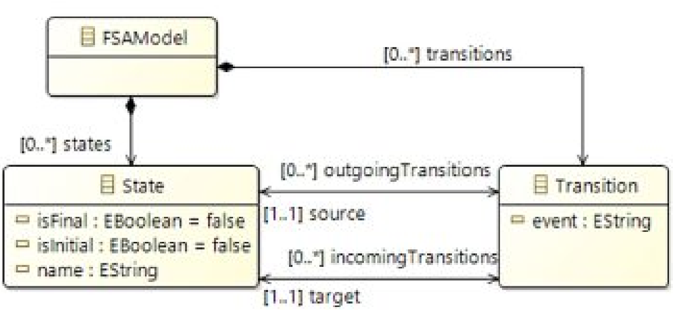

Figure 2.6: The abstract syntax of a finite-state automaton [36].

Semantics

Every language has a meaning, also called semantics. When a new element is added to a model or two elements are connected, meaning is created. Usually, the semantics are derived directly from the problem domain for which the DSL is designed. For instance, the implementation of an infotainment system for a car already has a well-defined meaning within the application domain. In this problem domain, the modeling concepts, like a knob, a menu, or an event, can be found. In general-purpose languages, it is up to the developers to adapt the semantics and language concepts to a specific problem domain. Every individual developer implements the mapping between concepts and problem domains differently. For this reason, different types of models are also created for the same problem domain. This is the advantage of a DSL that insists on using the concepts and semantics of the application domain [29].

In the following, we introduce different kinds of methods to formally define the semantics.

The operational semantics method uses an interpreter to define a language. The interpreter produces an evaluation history when it interprets the program. The evaluation history is a sequence of internal interpreter configurations. The meaning of a program in the language is the produced evaluation history [37].

The *denotational semantics* method uses a valuation function to bind a program directly to its meaning, called its denotation. In contrast to operational definitions, the denotational definition does not require an interpreter or computation steps [37].

The *axiomatic semantics* method does not explicitly define the meaning of a program. Properties that are expressed with axioms and inference rules from symbolic logic are defined for the language constructs. To construct a formal proof, the property of a program is deduced from the axioms and rules. The kind of properties that can be proved are used to determine the character of an axiomatic definition. Axiomatic definitions are more abstract than denotational and operational definitions. The properties proved about a program may not be enough to completely determine the program's meaning [37].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Concrete Syntax

The concrete syntax is the most visible part of the language, as it describes the notation of the language and is strongly oriented towards the domain concepts. The representation is either graphical or textual but can also be visualized in the form of a matrix or table. Modeling languages are usually represented graphically combined with text. Figure 2.7 shows the domain concept and the associated notational structures of a finite-state automaton. Which form of notation is used for a DSL depends strongly on the real representation of the domain concept. To stay with the car example: A knob for a car infotainment system should have a similar visualization in the modeling language. Regarding the completeness of representation and representation fidelity, it can be said that the presence of exactly one notational construct for each concept is an essential criterion for interpreting different modeling concepts and notations. This approach avoids overloading the notational constructs and ensures that all concepts can be represented with the language [29].

Concept	Notation
State	<<name>>
Initial State	<<name>> (initial)
Final State	<<name>> (final)
Transition	<<event>> source: <<name>> target: <<name>>

Figure 2.7: The concrete syntax of a finite-state automaton [36].

2.3 Debugging in Domain-Specific Modeling

There are two important facets of the development process of domain-specific modeling. Developing transformations and developing models. Model transformation [38–41] in the context of Model-Driven Engineering (MDE) is used to convert a source model into a target model. If both the source model and the target model match the same metamodel, this is called an endogenous transformation and is used to perform model refactoring and optimization. However, if both have a different metamodel, it is an exogenous transformation and used for tasks like code generation and reverse engineering. There exist also studies that deal with the debugging of model transformations [42, 43]. Since this work focuses on the generation of models and their interpretation, we will deal in the course of this section only with debugging models and artifacts itself, but not with

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

debugging transformations. First, debugging of source code, including the procedure and the advantages of debugging, are discussed. Afterward, the individual debugging concepts and how they are used for debugging code are represented. Finally, it will be analyzed how these debugging concepts can be transferred to modeling languages.

2.3.1 Debugging Code

Debugging is an essential part of every developing process. Any software that is developed should be bug-free before delivery. Debugging is a process used by developers and software testers to find errors and fix them [44]. Several studies deal with the most common causes of errors and why some errors cause more damage than others [45]. To track down and resolve bugs, concepts like stepping through code, interrupting the execution, or investigating information about the current program state can be used. The debugging process can be divided into six main phases [46]:

1. **Identify Error:** The earlier the error is found, the more time and costs can be saved. Errors that occur at the customer site increase the costs and damage the company's reputation.

2. **Identify the Error Location:** After the error has been detected, it is necessary to identify the exact location within the source code that causes it. With the knowledge of the exact position of the faulty code, the error can be fixed faster.

3. **Analyze Error:** In this phase, a suitable procedure must be found to analyze the error. The debugger provides information regarding the program's state and the data structure to analyze the error. This is a critical phase because by fixing the bug that has occurred, new errors can creep into the code.

4. **Prove the Analysis Result:** Once the error has been analyzed, the focus must be on the software's further errors. This includes writing test cases to ensure the correct functioning of the software.

5. **Cover Lateral Damage:** In this phase, tests must be performed to check the changed code. Those test cases which do not pass the test must be investigated. As soon as all test cases pass the test, the debugging process's last phase can be initiated.

6. **Fix and Validate:** This is the last phase, all bugs are fixed, and all test scripts are executed.

The advantages of software debugging can be summarized as follows [46]:

- **Saves time.** Performing debugging during the development process saves time because software developers do not need to write complex code to find the error.

- **Reports Errors.** As soon as errors occur, debuggers provide an error report. This enables early detection of errors and thus simplifies the software development process.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- **Easy Interpretation.** The information provided through the debugger about the execution state and the data structures makes it easier to interpret errors. The software developer can fix errors before the product is delivered to the customer.

The following debugging facilities are commonly used in modern IDEs and their debuggers. However, it is only a selected list of all available debugging facilities, but that is sufficient as an effective basis for debugging source code [42].

Execution Modes

All available IDE debuggers provide two basic functionalities: First, to execute a program continuously until the program is terminated or interrupted by the developer and second step-by-step execution of one statement at a time. Further execution features are the stop and pause commands for the terminating or non-terminating halt of the execution. Using the step-by-step mode or the start/continue command, the programmers re-enters continuous mode [42].

Steps

There are usually three kinds of step commands in every debugger: step-over, step-in, and step-out. The step-over command executes the current statement as one block. On the step-in command, the debugger goes through all the sub-statements contained by the current statement step-by-step. This further requires a change of scope. The step-out command executes the application until the end of the current scope is reached, and the debugger is one level higher in the call stack [42].

Runtime Variable I/O

When the application's execution is paused, debuggers provide the functionality to read and change the program's global and local variables. While the execution is paused, developers can assign any values to variables, apart from basic types. The significant advantage of this feature is that no print statements are necessary for the user to investigate the program's current state [42].

Breakpoints

The developer can specify breakpoints to interrupt the program's execution at a specific point in the source code. Breakpoints are set on statements to pause the execution before the debugger executes the given statement. Most debuggers also support more advanced breakpoints: Breakpoints with boolean conditions, breakpoints with hit counts, exception breakpoints, data breakpoints, or function breakpoints. Condition breakpoints halt the execution if any from the developer specified condition over the state is fulfilled. Hit counts for breakpoints pause the execution when the statement is executed an arbitrary number of times. Exception breakpoints pause the execution as soon as a certain exception is thrown. Data Breakpoints interrupt the execution when the value of a

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

certain variable changes. Function breakpoints lead to a halt in the program when calling a certain function. Exception breakpoints, data breakpoints and function breakpoints are not associated with a specific source location [42].

Stack Traces

Stack traces provide the developer with a view of the function calls that led the program into its current state. While the execution is paused, stack traces become visible. The programmer can dive into any other level in the stack trace for variable viewing and source code editing [42].

Exceptions

Exceptions are thrown during runtime and inform the user that the system is in an error state and may halt the program's execution. The exception contains information on the error, the type of exception, and the state (exception context) of the program when the error occurred. There are many exceptions predefined in languages to report events like null-pointer dereferencing and I/O problems. When an error occurs, the runtime system climbs up the call stack and searches for a method that contains a block of code that can handle the exception. This code is also known as an exception handler. Appropriate exception handlers can be defined by developers to *catch* an exception and handle the exception (take appropriate action) to recover from the identified issue. If no exception handler can handle the thrown exception, the runtime system terminates. Finally, it should be mentioned that programmers can also implement new types of exceptions to enable support for system-specific exceptional situations [42].

2.3.2 Debugging Models

Hans Vangheluwe and Raphael Mannadiar [42] identified two different debugging scenarios in domain-specific modeling. These two scenarios have roughly the same basic workflow but differ in the different expertise of the users. On the one hand, designers are debugging model transformations and artifacts generated by them. They are fully aware of the model transformation that describes their models' semantics and generates artifacts. Their goal is to ensure the correctness of the generated artifact. On the other hand, the modelers are interested in the correctness of their domain-specific model and are only familiar with the semantics of their model without being aware of artifacts generated from them. Below we re-visit the debugging concepts described in the previous section and translate them to debugging in domain-specific modeling [42].

Execution Modes

Playing and stopping the execution require functionality to run or kill the generated model or program remotely. Considering stepping, one must first identify how a step can look in a particular model. This can be challenging, depending on the language. Vangheluwe and Mannadiar [42] propose in their paper a very general definition.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

"Any modification of the state of a domain-specific model constitutes a step, where the notion of the state encompasses the values of the parameters of every construct in the model." ([42], p. 15)

Like stepping, pausing the execution also presents a certain challenge. The pause mode should depend on the structure and semantics of the domain-specific model. The most common and widely used approach is to pause the execution right before the execution of the next step at the model level. While this approach is sufficient for modelers, it should be noted that it could be further refined to apply to designers. It is a great advantage and sometimes even necessary for designers to pause or step on the level of generated code statements [42].

Steps

The meaning of stepping over, into and out, may be considered from two orthogonal perspectives. On the one hand, the notation can be translated one-to-one for DSLs that support hierarchies and composition. Taking a statechart-based language when the current state is within a composite state, stepping out would continue the execution of the model until it leaves the parent state. On the other hand, considering that an implicit hierarchy is created between the generated artifact and the source model, it can be useful for the designer, in case of the step-in command, to switch the execution pointer of the debugger to the corresponding lower-level entities. Keeping up with the statechart example, the use of the first step-in command would allow the designer to step through the underlying statechart, while another step-in command would allow the designer to step through the statements of the code generated from the statechart. Stepping out would bring the designer back to the higher-level representations. These different stepping modes confirm the requirement for two different debug modes for designers and modelers [42].

Runtime Variable I/O

While the execution is paused, variable values of domain-specific models and lower-level generated artifacts can be viewed and modified using the model editing tool. The real challenge is that a change of a variable at one level of abstraction must also be visible on every other level to ensure overall consistency [42].

Breakpoints

Breakpoints, as known from typical IDE debuggers, can be translated one-to-one to the DSM world. Nevertheless, there are differences between the requirements of a modeler and those of a designer. For the modeler, it is sufficient to specify breakpoints only on the model level, whereas for the designer, it is relevant to set breakpoints on each level of abstraction between model and end artifact. Depending on the modeling language, it must be defined which elements of a model can be marked as breakpoints. Looking

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

again at the statechart-based language, it probably makes most sense that individual states support the setting of breakpoints. In the course of transitions that use actions, it should also be considered whether one can set a breakpoint before executing the action and thus pausing the application. In the case of Petri-Nets [47], where the state is implicitly represented, this approach is not practical. Hans Vangheluwe and Raphael Mannadiar [42] instead porpose for this case that breakpoints are specified as patterns. When the patterns are matched in the model, the execution gets paused. As in code IDEs, hit counts and boolean conditions could be specified in the same manner. While function breakpoints are not important for the modeler, they can be useful for the designer concerning generated artifacts and the functions called at this level. Exception breakpoints are strongly oriented on how exceptions are handled in the domain-specific models, and the lower level generated artifacts [42].

Stack Traces

As known from code debugging, stack traces in the domain-specific model and artifact debugging are bound to the terms stepping in and stepping out. From the perspective of the modeler, the structures of the hierarchy and composition of DSMLs are reflected. On the other hand, the designer debug mode might display sequences of related actions at different levels between model and generated artifact [42].

Exceptions

As we mentioned in the previous section about DSMLs, only correct models can be created due to the abstract syntax's definitions. Therefore, when debugging models, exceptions do not occur at the level of the model but during the execution of the synthesized artifact. Although the models are animated and updated in real, only the artifacts are executed. Exceptions can be caused by I/O errors or bugs in the execution engine or third party libraries but also through operational semantics transformation of lower-level models. Some of these exceptions can be translated one-to-one into domain-specific terms. In contrast, other exceptions require propagation and translation facilities for the presentation to the designer or modeler. Furthermore, it should be noted that some exceptions are only of minor importance for the modeler, and others may provide information that is useful only for the designer. The DSL architect's task is to decide between exceptions that are passed on to the highest level of abstraction and those that are silently handled [42].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 3

Web-based Modeling Tools

This chapter discusses existing technologies for implementing web-based modeling tools. In particular, the protocols LSP, GLSP, and DAP are introduced.

3.1 Language Server Protocol

The number of programming languages has increased rapidly over the last few years. Besides, more and more IDEs were created [48]. Nevertheless, it is difficult for the IDE developer to keep up and constantly offer new language support. Language providers are very interested in offering as many IDE integrations as possible for their language to gain a broader user community. The result is an m-times-n complexity to integrate all existing languages into every available IDE. In 2016, Microsoft, RedHat[1], and Codenvy[2] introduced the Language Server Protocol[3] (LSP) to separate the language-specific logic from the integration into an IDE. The language server, which is offering the language logic, can thus be implemented in any programming language. Code-completion, goto definition, and refactoring are only a few functionalities computed in a language-specific server process [49].

LSP's goal is to communicate messages over a standardized protocol between the language server process and the IDE client process, which integrates the language into an IDE. This process reduces the complexity of integrating all languages into every existing IDE to m+n. Since the introduction of the LSP, more than 100 language servers and more than 30 tools have been implemented that support LSP. These include popular languages like Java, TypeScript, C, and Python. On the editor side, the LSP is supported

[1] https://redhat.com/en
[2] https://codenvy.com
[3] https://microsoft.github.io/language-server-protocol

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

by IDEs like Eclipse[4] or VSCode[5]. In addition to general-purpose languages, there already exist various language servers for domain-specific languages (e.g. HTML, LateX[6], CSS). However, Xtext[7] [50] is the only language workbench that currently supports the language server protocol. This allows languages that are implemented with Xtext to take advantage of the LSP's features and benefits [49].

3.1.1 Architecture of the LSP

The protocol uses the stateless and lightweight remote procedure call protocol JSON-RPC[8] in its version 2.0, which uses JSON as a data format. JSON-RPC was designed to transmit remote procedure calls over a network. However, it can also be used to transfer data from a client to a server running on the same physical machine. Most language server implementations run on the same physical machine as the operated IDE. The LSP distinguishes between three different types of messages: *requests*, *responses*, and *notifications*. A *request* message describes a request between the client and the server. Every processed request must send a *response* message as a result of a request back to the sender of the request. A *notification* message works like an event and notifies the receiver of an occurred activity. All messages are divided into a header and a content part. The header specifies the content length and content type. The content part contains the actual content of the message [51].

Figure 3.1 shows an editing process and the resulting communication between the three actors, the user, the the client (IDE / development tool), and the language server. The user, shown on the left side of the illustration, starts the communication between the client and the server through its actions. The arrows between the client and the server mark the messages that have been standardized in the language server protocol. The session starts when the user opens a document. The development tool then notifies the language server that a document has been opened. As a next step, the user edits the document. Again, a notification is sent to the server from the development tool. After the server has received the notification, it analyzes the changes done by the user. If errors or warnings are detected by the server, they are returned to the client. In a further step, the user executes a *goto definition* action. The client sends a request to the server with the current cursor position within the document. In Listing 3.1, the request *textDocument/definition* of the *goto definition* action is shown in detail as a JSON-RPC call. Besides, the URI to the text document, the line, and character location are transferred to the language server. The server then determines the exact position of the variable declaration and sends the resulting location back to the development tool as a response. The corresponding JSON-RPC call is shown in Listing 3.2. The response includes the new URI and the location range of the target position. The development

[4] https://eclipse.org/ide
[5] https://code.visualstudio.com
[6] https://latex-project.org
[7] https://eclipse.org/Xtext
[8] https://jsonrpc.org/specification

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

tool is responsible for opening the correct document based on the location and setting the cursor to the specified position. The last message notifies the server that the user closed the document. Based on the example shown, it can be seen that the protocol is based on documents and the positions within them. As a result, the protocol remains independent of the specific language for which the LSP is used.

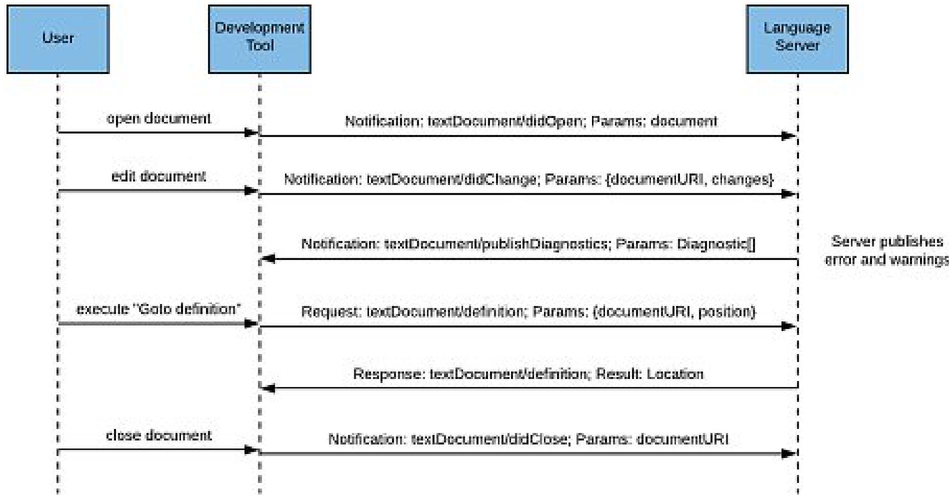

Figure 3.1: Communication between the development tool and the language server during the editing session [51].

```
1  {
2    "jsonrpc": "2.0",
3    "id" : 1,
4    "method": "textDocument/definition",
5    "params": {
6      "textDocument": {
7        "uri": "file:///p%3A/mseng/VSCode/Playgrounds/cpp/use.cpp"
8      },
9      "position": {
10       "line": 3,
11       "character": 12
12     }
13   }
14 }
```

Listing 3.1: *Goto definition* action in a C++ file [51].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "uri": "file:///p%3A/mseng/VSCode/Playgrounds/cpp/provide.cpp",
    "range": {
      "start": {
        "line": 0,
        "character": 4
      },
      "end": {
        "line": 0,
        "character": 11
      }
    }
  }
}
```

Listing 3.2: Response to the *goto definition* request in a C++ file [51].

3.1.2 Features of the LSP

Since not all languages and IDEs support all the protocol features, the supported features are sent as so-called "capabilities" from the IDE to the language server during the initial requests. The language server interprets missing *Capabilities* as not being supported by the client. Unknown *Capabilities* are ignored. The language server responds to the initial request with the *Capabilities* it supports. In the following, we briefly describe the LSP key features, which are supported by most language server implementations [51].

Code completion. The client sends the request *code completion*, including the current document and the cursor position within the document to the language server. The server responds to the request by sending the computed completion elements for the given position back to the development tool. The client process then forwards the completion proposals to the context menu from which the user selects the element. In the case of an expensive completion, the completion can be represented by a *handler*. As soon as the user selects an element from the context menu, the *handler* sends another request to the language server to request the entire completion text.

Hover. The development tool sends the request *hover* to the language server to compute additional information shown in a tool tip. The hover menu feature further supports text-formatting, including line breaks, lists, and indentations.

Goto Definition. Through the request *goto definition*, the language server responses the position of a given symbol within a document. The request *goto type* determines the position of a given type. For the implementation location of a given symbol, the request *goto implementation* is available in the LSP.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Workspace symbols. The development tool sends the request *workspace symbol* taking a query string to the language server to reveal all matching workspace symbols within the workspace. Workspace symbols are an index (or tag) file of names found in source and header files of various programming languages.

Find references. The request *find references* returns a project-wide list of locations based on the current cursor position. Each of the locations references the symbols at the current cursor position.

Diagnostics. Diagnostics is executed by the language server and might be handled per file or for a complete project. Through diagnostics, errors and warnings are detected and sent via notification to the development tool. If no errors or warnings occurred, the server generates a notification with an empty list of diagnostics. Therefore the client always replaces the current diagnostics, and no merge is required.

In addition to the capabilities already mentioned, the LSP also contains other interesting features such as code lens or signature help. New features will continue to be integrated into the LSP, and so the LSP in version 3.16.0 will support features like call hierarchy or semantic token.

3.2 Graphical Language Server Protocol

The Graphical Language Server Protocol (GLSP) was introduced by EclipseSource[9] in 2018 and follows the same architectural pattern as the LSP for textual languages, but uses it for graphical modeling languages. Initial attempts to reuse the LSP for graphical modeling had to be rejected due to the different requirements of textual and graphical languages. Thus, a new protocol has been defined. The GLSP is primarily based on the client-server communication protocol of the diagram framework Eclipse Sprotty[10]. Additionally, GLSP comes with edit functionalities and server-specific communication. Probably the most important use case for the protocol is the development of web-based diagram editors, which connect to a language server and transmit messages via the protocol. Besides the defined protocol, the Graphical Language Server (GLS) platform[11] provides a client and a server framework that can be integrated into existing IDEs. The client is responsible for rendering and time-critical operations, such as fade out effect, whereas the server is in charge of the language-specific semantics [52]. In Chapter 4, we will use the platform in the running example to integrate a diagram editor into the web-based IDE framework Eclipse Theia[12].

[9] https://eclipsesource.com
[10] https://projects.eclipse.org/projects/ecd.sprotty
[11] https://eclipse.org/glsp
[12] https://theia-ide.org

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

3.2.1 Architecture of the GLSP

As already mentioned, the GLSP builds strongly on the client-server protocol of Eclipse Sprotty and its architecture. Sprotty is a web-based diagram framework using modern web-technologies like SVG for rendering and CSS for styling. A unidirectional event-cycle is an essential part of the architecture with a virtual DOM as opposed to the model-view-controller pattern. Therefore, the event flow is always clear, does not form feedback loops, and takes less effort to test [53]. Figure 3.2 shows an overview of the Sprotty architecture, which is described in the following.

All relevant information regarding the diagram and the operations that can be performed on it is stored on a graphical language server. The client is only equipped with the most necessary details, which it needs to render the diagram. Sprotty was developed to take over the visualization part of language servers. The Sprotty server extends language servers created with the language workbench Xtext. This facilitates the creation of graphical representations for DSLs. It is precisely this approach on which the GLSP is based: Separation of language logic from IDE integration. Instead of using the LSP for textual languages, the GLSP for graphical modeling languages can be used. Besides, the GLSP extends the static visualization of Sprotty with edit functionalities [52].

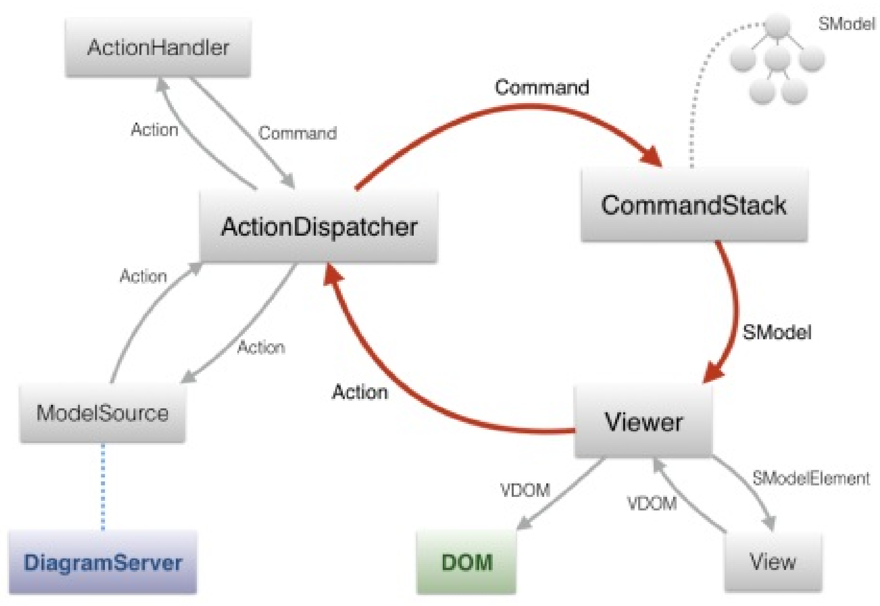

Figure 3.2: The architecture of the diagram framework Eclipse Sprotty [53].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

In the following, we describe the main concepts of the Eclipse Sprotty architecture, which is essential for the GLSP [53].

- **SModel.** The diagram is stored in a graph model called *SModel*. The diagram's elements are organized in a tree hierarchy and have the properties *parent* and *children*. Based on the model root (i.e. *SModelRoot*), elements can be searched within the tree through their ID. All elements of the model inherit from *SModelElement*. An *SModelElement* consists of a unique ID and a type to reference its *View*. A *View* knows how to turn a graph model element and its children into a virtual DOM node. To use the client/server architecture of Sprotty, the graph model needs to be serializable. The schema of an *SModelElement* is a serialized JSON type. Within the schema, cross-references are represented by the ID of the element to be referenced. When looking at an edge, an *SEdgeSchema* refers to its source using the ID of its source *SNode*. Finally, to create the model out of the corresponding schema, deserialization is taken over by an *SModelFactory*.

- **Actions.** Actions are executed on the graph model and its elements. They are defined as JSON objects and are transmitted between client and server. From the model source or the *Viewer*, actions are sent via the *action dispatcher*. The *action dispatcher* takes the received action and converts it into a command using the appropriate action handler. There are two different ways on how to control the model. First is through a local model source, which directly changes the model on the client, and secondly through a remote source controlled by a diagram server.

- **Commands.** Commands take the current model, execute the operation on it, and return a new model taken up by the *Viewer*, which initiates a new rendering process. Commands are passed to the command stack, which has a redo and undo stack in addition to the stack to be executed. Thus, individual commands can be repeated or undone. The execution stack can be used to merge individual commands. This makes especially sense in the context of a move command, where only the starting point and endpoint are of interest.

- **Viewer.** The *Viewer* is in charge of creating a new virtual DOM from the model. The information necessary to create the virtual DOM is stored in the *View* of a model element. A *View* must be registered in the *ViewRegistry* so that the *Viewer* can look up the *View*, based on the ID of the model element. Furthermore, it is possible to add event listeners or animation to the *Viewer* using its *Decorators*.

- **Features.** Features are supported by model elements and describe the interaction between the user and the diagram element. A feature contains the action that is triggered and the corresponding command that has to be executed. Furthermore, a feature requires a singleton *Symbol* as an identifier to check if a model element supports this feature. An *SModelExtension* can be added to store further information in the model to support the feature. Additionally, listeners can be registered

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

to listen to events that execute the interaction on the DOM elements. Finally, a *Module* is required to register the above information to the client infrastructure.

Figure 3.3 shows the communication between the user, client, and graphical language server while adding new elements to the diagram. The client's first message to the server represents various requests from the Eclipse Sprotty framework base protocol, which, for example, request the model and the bounds from the server. The bounds are the position (x, y) and dimension (width, height) of an model element. Then the development tool asks the language server to transmit all available operations. The operations received from the server are added to the palette view of the development tools' diagram editor. They can then be used to add new model elements to the diagram. In a further step, the user creates a new node within the diagram. The client sends a *create node* operation with the element type and the location within the diagram to the server. This modifies the model and the language server sends the new model back to the client using an *update model* action, which triggers the rendering of the diagram with the new model. In another use case, the user creates a new edge between two existing elements. A *create edge* operation is executed and sent to the language server. In turn, the server modifies the model and adds a new edge between the two, based on the source and target ID of the respective elements. After the model has been changed, the new model is sent back to the client, taking over the rendering.

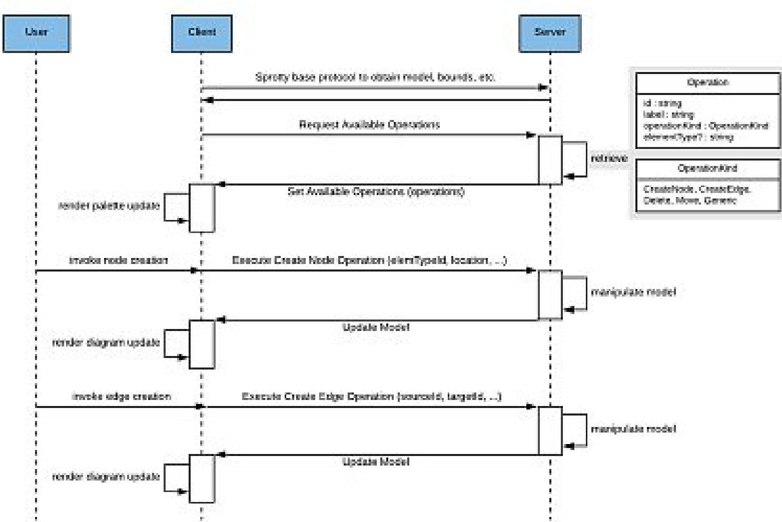

Figure 3.3: A user invokes a GLSP-based communication between client and server through adding new elements to a diagram [52].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

3.2.2 Features of the GLSP

In the following, we briefly describe the key features of the GLSP [52].

Request and Update Model. The *request model* action is sent as the first message from the client to the graphical language server to initiate the communication between the two and, at the same time, request a graphical model. The server responds to this client request, either with a *set model* action or an *update model* action. The *update model* action is sent to the client after each modification of the model on the server. If the client does not have a model, the *update model* action behaves like a *set model* action. If a model already exists, the old model root schema is replaced by a new schema.

Operations Request. The request *operation action* is sent from the client to the server to request the list of available operations. Operations are requests that change the model. The server, in turn, responds with a *set operation* action. GLSP currently supports six different types of operations:

- **Create node:** This action creates a new node within the diagram. Specifying the element type, location, and container within which the operation is to be executed.
- **Create connection:** This action creates a new connection between two elements. The action transfers the ID of the source element as well as the ID of the target element. This operation can be extended to create edges.
- **Delete:** This action performs the deletion of a specific item based on the ID.
- **Change bounds operation:** This action is used to change the position or dimension of an element.

Element Pop-up. As soon as the user moves the mouse over an element, the client sends the request *popup model action* to the graphical language server to get detailed information about the element. The server responds with a *set popup model* action to display the pop-up menu on the one hand and to remove it in the diagram editor on the other hand.

Select. A *select* action is triggered as soon as the user clicks on a selectable element. The action can be handled locally by the development tool or sent to the graphical language server to respond to a selection change. Furthermore, the server can change the selection remotely and inform the client about this via the *select* action. Finally, the GLSP has a *select all* action to select or deselect all the model elements at once.

Command Palette. The *command palette action* sent from the client to the server requests the available operations for one or more selected elements. The server uses the received IDs to determine the available operations and sends them back to the client with a *set command palette* action.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Reconnect and Reroute Connection. The *reconnect connection* action is sent from the client to the server to change either the source element or the target element of an existing connection. The transmitted message contains the ID of the connection, as well as the ID of the new source and target elements. Compared to the deletion of an existing connection and the resulting re-creation of a connection, reconnecting has the advantage that the object identity and the properties of the connection are already on the server. The *reroute connection* action changes the existing route of a connection, where the new route is transmitted as an array of routing points.

3.3 Debug Adapter Protocol

The Debug Adapter Protocol (DAP) was introduced before the Language Server Protocol but follows a similar approach. Due to the different APIs of development tools, it takes significant effort to implement a debugger for programming languages. The DAP aims to standardize the communication between the development tool and debugger and thus to make it possible to implement a single generic user interface that supports the DAP on the client-side and a well-specified protocol on the server-side. A debugger then implements the debugger logic of a particular language also follows the DAP protocol. This makes it easy to integrate debuggers for different languages into an IDE and re-use debuggers across IDEs. Since existing debuggers may not adopt the DAP, the debug adapter should adapt existing debuggers or runtime APIs to the DAP. Figure 3.4 shows on the left side different development tools, where each tool integrates the implementation of a debugger user interface to support the language-specific debugger. On the right side, the development tools are using generic debugger user interfaces. The DAP-based tools are connected through a debug adapter, supporting the DAP to the language-specific debuggers. This reduces the effort to support a new debugger considerably [4].

3.3.1 Architecture of the DAP

The DAP was designed before JSON-RPC was released. Therefore, it uses its own protocol, which is very similar to JSON-RPC. The messages consist of a header and a content part separated by a carriage return and line feed. The most important data types which are used by the DAP are Strings because the user sees exactly this in the graphical user interface. This allows an easy implementation of the debug adapter. The debug adapter can either function as a debugger itself or be connected to an external debugger. The DAP has many features, but not all debug adapters support them as well. Each feature has a capability flag, which lets the development tool know if a debug adapter supports this feature or not. Therefore, these feature flags are summarized in the DAP under the interface *Capabilities*. If a flag does not exist, the development tool assumes that the adapter does not support this feature [4]. Since the DAP is not stateless, the server needs to know all kinds of states and sequences of what has happened. If the server terminates, the debug progress on the client-side is lost [4].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

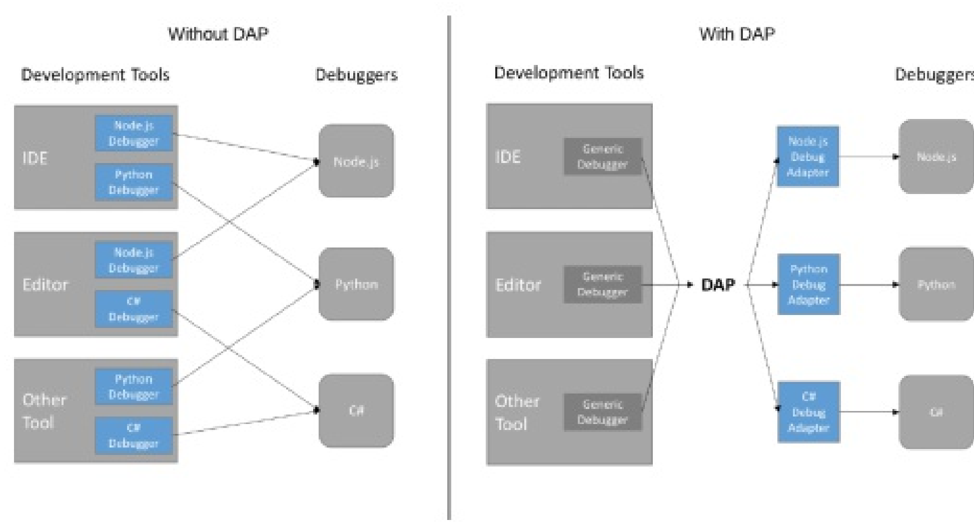

Figure 3.4: Communication between development tools and debuggers with and without DAP [4].

The DAP consists of three types of protocol messages. First, *requests* containing the command that should be executed by the debug adapter and arguments that carry the necessary information for the execution process. Secondly, *responses* that act as acknowledgment and return the generated information from the execution process back to the development tool. Finally, *events* initiated by the debug adapter inform the tool about the current state of the execution. An example of such an event is the stop event, which informs the development tool that the execution has been stopped for some reason. This may be due to a breakpoint or an exception.

Figure 3.5 shows the communication between the development tool, debug adapter, and a GNU debugger[13]. The communication is started by the user, creating a new debug session. Subsequently, the development tool sends the *initialize* request to the debug adapter. In the initialization phase, the *Capabilities* are exchanged between the IDE and the debug adapter. The debug adapter then connects to the GNU debugger and starts the debugger. In addition to the *Capabilities*, the initial request also contains other useful information about the development tool, such as its name or the format of the file paths (native or URI) used.

As soon as the debug adapter is ready to receive configuration requests, it generates a new *initialized event* to inform the development tool about its status. Due to this process, the adapter does not require a buffering implementation for configuration information. After receiving the *initialized event*, the development tool starts sending

[13]https://gnu.org/software/gdb/

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

the configuration information to the debug adapter. These are a sequence of requests that set the different types of breakpoints, followed by the *configuration done* request that signals the completion of the configuration process. The breakpoints are forwarded to the GNU debugger for validation, and the result is sent back to the development tool via the debug adapter. After the successful completion of the configuration process, the *launch* request follows, starting the debugger's execution. The debug adapter forwards the name of the file to be debugged to the GNU debugger and runs the execution.

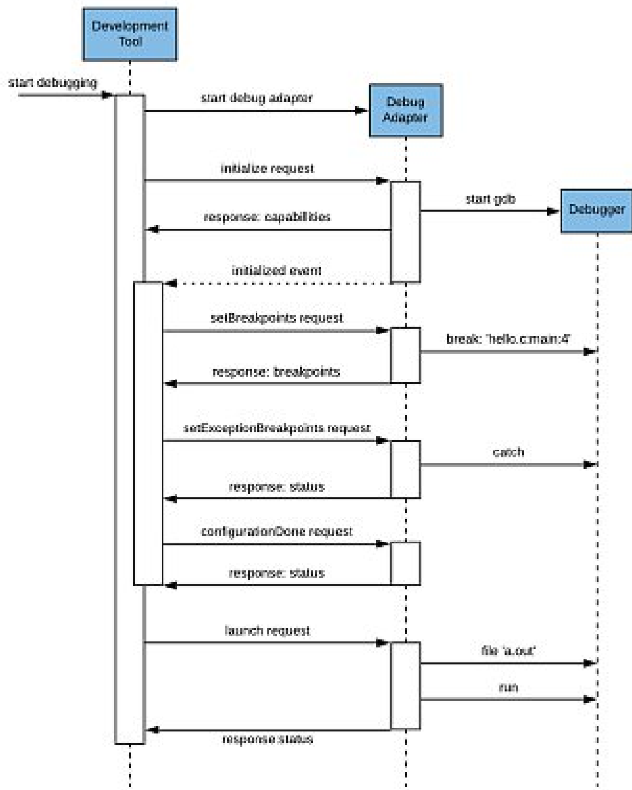

Figure 3.5: Communication between the development tool, debug adapter, and debugger during the start of a new debug session following the DAP [4].

3.3.2 Features of the DAP

In the following, we represent the features of the DAP regarding the debugging concepts introduced in Section 2.3 [4]. All requests, responses, events, and data types are clearly presented in the Debug Adapter Protocol specification[14].

[14] https://microsoft.github.io/debug-adapter-protocol/specification

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Execution Modes

After a debug session has been started and the configuration process has been executed, the client sends the *launch* request to the debug adapter to start the debugger. This request contains debugger-specific information such as the debugger server address or a stop on entry flag. The stop on entry flag informs the GUI that the debugger will be stopped when it enters the file and before the first statement is executed. The execution mode can be started with or without debugging. Once the debugger is started and is in the stop state, the debugger can be continued by using the *continue* request. Through a *pause* request sent by the client, the execution can be stopped again. For this purpose, the debug adapter generates a stop event to inform the development tool about the halt of the execution. After the debug adapter was successfully launched, the development tool sends a *threads* request to the adapter and starts listening for new thread events. All requests exchanged between the development tool and the debug adapter during the debug session must specify the corresponding thread ID. Depending on whether the debug adapter supports multiple threads or not, the debugger responds either with all currently existing threads or at least one (dummy) thread. To terminate the debug session, the development tool sends a *terminate* request to the debug adapter to terminate the session gracefully. This gives the debugger the ability to complete all remaining tasks and finally ends the debug session. If the debugger continues to run, the development tool tries to terminate the debugger again, but this time it uses the *disconnect* request, which terminates the session forcefully.

Steps

In order to step through the execution, the debug adapter protocol provides the *next* request to run again for one step. The debug adapter responds with a new stop event, including the appropriate reason and thread ID, after the step is complete. Using the *step-in* request, the debug adapter steps into a function or method, if possible. Otherwise, the *step-in* request behaves like a *next* request. To reach a level higher after a step-in command, the client sends a *step-out* request to the adapter.

Runtime Variable I/O

The exchange of the variables takes place using the *variable* request, which returns all or only a part of the variables. The child variables of a value can we retrieved using the variable reference. The variables reference relates to a specific variable. In addition to displaying variables, they can also be modified. For this purpose, the DAP provides a *set variable* request in which the name of the variable and the new value are sent to the debug adapter. In response, the adapter sends the new value and its data type to the client.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Breakpoints

The DAP supports most types of breakpoints. For each type of breakpoint, requests are sent to the debug adapter. Using the *set breakpoints* request, the development tool transmits all breakpoints that exist for a single source. The adapter deletes all previous breakpoints for this one source and then sets the breakpoints contained in the request. The DAP also supports function breakpoints. The transferred breakpoints are validated through the debugger and sent back to the development tool as the actual breakpoints via a response message. This is to communicate to the client if a breakpoint could not be set at the defined position or if the breakpoint was set to a different position by the debugger. In addition to specifying hit counts and conditions, exception breakpoints and data breakpoints can also be transferred via the protocol. As soon as a breakpoint is hit during execution, the debug adapter creates a new stop event and adds the respective type of breakpoint (e.g. function breakpoint) to signal the cause of the execution's halt.

Stack Traces

In order to receive the stack trace of the current execution state, the development tool can send a *stack trace* request to the debugger. The request contains arguments to filter the returned stack frames. This means that only those stack frames that belong to the thread specified in the request are returned. A unique ID identifies a stack frame across all threads. This ID can be used to obtain the scopes of a frame using the *scopes* request or to restart the execution of a specific frame. The stack frame further stores the name of the frame, typically a method name, as well as the line and column of the frame within the file. Additional optional properties, such as the source of the frame, can also be transferred from the debugger to the development tool.

Exceptions

During the execution of a program, there may occur exceptions. In this case, the debug adapter generates a new stop event with the corresponding reason (e.g. an exception). There are two ways an exception can be processed and displayed by a development tool. In the first, an output event is generated after the stop event. The output event can be used to output a sequence of characters to the debug console. In the second variant, the development tool sends an *exception info* request to the debug adapter after receiving the stop event. The response that follows is filled with detailed information about the exception that occurred. After the development tool has received the *exception info* response, the information can be further processed and displayed on the debug console.

3.4 Summary

To sum up the previous sections, LSP, GLSP, and DAP are protocols that standardize the communication between an IDE and a language-specific server that provides editing or debugging support. The LSP separates the language-specific logic from the actual

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

editor logic. Since the LSP is only suitable for textual languages, the GLSP was defined to build client-independent language servers for graphical modeling languages. The DAP allows the implementation of one generic debugger interface for a development tool that can communicate with different debuggers via debug adapters. The debug adapters can be reused across multiple development tools, which significantly reduces the effort to support a new debugger in different tools.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 4

Applying the DAP to GLSP

In Chapter 2, we discussed which features a modern debugger should provide and how these debugging concepts can be applied to graphical modeling in general. Afterward, we introduced both the GLSP and the DAP in Chapter 3 and showed which features and advantages these web-based tools offer. In this chapter, we will discuss how to apply the DAP to the GLSP and integrate a new debugger for a graphical modeling language. First, we will introduce the Workflow Modeling Language (WFML) and its custom diagram editor. The requirements for the debugger are then derived from the WFML. Finally, we introduce the implemented debugger and discuss the features using visualizations.

4.1 Running Example: Workflow Modeling Language + Diagram Editor

The WFML[1] was designed by EclipseSource in the course of the development of the Graphical Language Server (GLS) platform. The WFML is an executable/interpretable language and is therefore very well suited for the purposes to develop a debugger for graphical modeling languages. In Section 2.2, we discussed the components of which a domain-specific language consists and that were used to define the WFML. The abstract syntax of the WFML was defined using the Ecore Modeling Framework (EMF) [54]. The EMF is a modeling framework and code generation facility for generating tools and other applications based on a structured data model. The Java code generated from the EMF Ecore model was integrated into the server framework of the GLS platform. The concrete syntax was designed using the diagram framework Eclipse Sprotty, which we already presented in the course of the introduction of GLSP. The default shapes of the Sprotty framework can be adapted/extended to fit the requirements of the WFML.

The WFML describes a workflow within a system or organization. This workflow can be a process consisting of a sequence of tasks and decisions. The language consists of

[1] https://github.com/eclipse-glsp/glsp-examples

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Bibliothek, Your knowledge hub

task nodes, activity nodes, and edges that connect the nodes. Furthermore, it is possible to introduce hierarchical structures within the WFML. To do so, one can create new diagrams, which are named according to the respective tasks.

The metamodel of the WFML shown in Figure 4.1 introduces the relevant language constructs. The WFML metamodel extends the GLSP metamodel[2] which serves as the basic structure for the GLS platform. The language constructs of the GLSP metamodel can be extended by the specific language constructs. Every element of the model is a *GModelElement*. A *GModelElement* can have zero or an undefined amount of *GModelElement* children. The GLSP metamodel shows a *GGraph* class which extends the *GModelRoot* class. The *GGraph* class stores the model, which consists of *Edges*, *TaskNodes*, and *ActivityNodes*. A *TaskNode* can be further divided into a manual task or an automated task using the property *taskType*. The type of an *ActivityNode* can be specified via the property *nodeType*. The WFML distinguishes decision nodes, merge nodes, fork nodes, and join nodes. The *WeightedEdge* class has a property *probability* and extends the *GEdge* class. The *GEdge* class has the properties *sourceId* and *targetId*. Finally, a *GEdge* has optional source and target model elements.

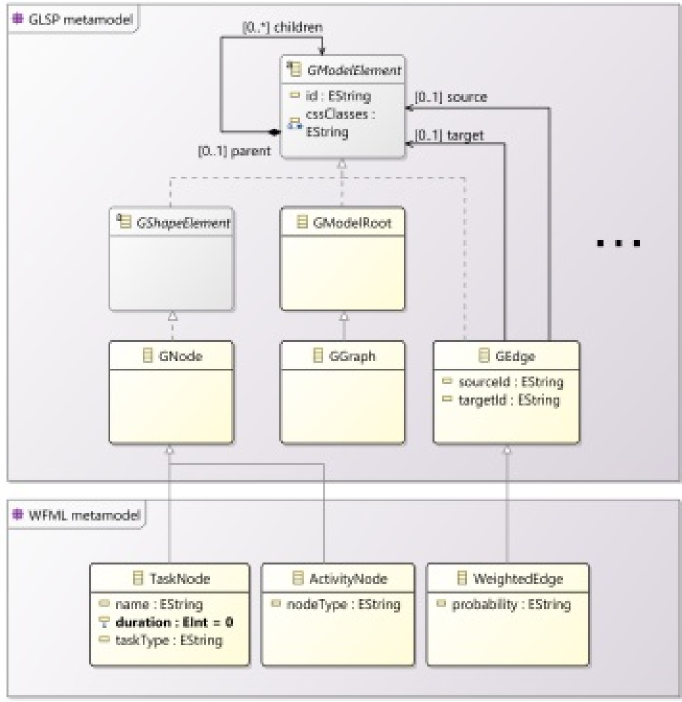

Figure 4.1: The WFML metamodel extending the GLSP metamodel.

[2] https://github.com/eclipse-glsp/glsp-server/blob/master/plugins/org.eclipse.glsp.graph/model/glsp-graph.ecore

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

In the following, we present the modeling concepts of the workflow language in detail:

- **Task.** A task describes a specific task, which is to be executed during the process run. The notation of a task is a rounded rectangle with the task name inside the border. A task has a property *duration* to specify the time needed to execute the task. The WFML distinguishes between the following two tasks.
  - **Manual Task.** A manual task is expected to be performed without any process execution engine or application. The duration for manual tasks is set to zero.
  - **Automated Task.** A process engine executes an automated task. The task is completed when the interpreter has executed the script defined for the task. An automated task requires a certain amount of time to complete the task. The time needed is specified in the duration property.

  The concrete syntax of automated tasks differs from manual tasks in that both have a different background color. Manual tasks are colored blue and automated tasks are colored gray. Besides, there is an icon next to the name of the task, which either depicts an "A" for automated or by an "M" for manual.

- **Activity Node.** Activity nodes are control nodes used to coordinate the flows between other nodes. The workflow language supports the following four types of activity nodes:
  - **Decision node.** This node accepts exactly one incoming flow and selects exactly one outgoing flow from one or more outgoing flows. Which of the flows is taken depends on the evaluation of the guards at the outgoing edges. The notation of a decision node is a diamond-shaped symbol with one ingoing edge and one or more outgoing edges.
  - **Merge Node.** This node merges multiple incoming alternate flows to accept a single outgoing flow. Merge nodes are used together with decision nodes. The notation of a merge node is a diamond-shaped symbol with one or more incoming edges and one outgoing edge.
  - **Fork node.** This node accepts an incoming flow and has several outgoing flows on the other side. The incoming flow is split into multiple concurrent flows to provide parallelism in workflow diagrams. The notation of a fork node is a line segment with one incoming edge and two or more outgoing edges.
  - **Join node.** This node has multiple incoming flows and only one outgoing flow and is used to synchronize incoming concurrent flows. Join nodes are used together with fork nodes to support parallelism in workflow diagrams. The notation of a join node is a line segment with several incoming edges and one outgoing edge.

- **Edge.** Edges are direct connections between nodes and describe the further flow of the process. The source node and target node of an edge must be in the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

same diagram as the edge. **Weighted Edges** have an additional probability and are evaluated during runtime to determine which of the edges will be traversed. Weighted edges are used in the context of decision nodes. In the current version of the WFML, weighted edges are the only way to define guards for decision nodes. Additional guards can be integrated by extending the relevant components of the client/server framework of the GLS platform. The notation of an edge is an open arrowhead line connecting two nodes. The graphical representation of weighted edges and standard edges is different in color. The standard edges are colored in black, and the weighted edges are shown in blue color.

Figure 4.2 illustrates an example defined with the WFML. The example shows how the individual language concepts interact with each other during the brewing process of a coffee machine. The task without any incoming edges is considered the starting point. The process starts when the machine's brew button represented through the manual task "Push" is pushed. Compared to the manual task "Push", the automated task "ChkWt" has an additional property *duration*, which specifies the time needed to execute the task. The automated task "ChkWt" is responsible for checking the water tank level of the coffee machine and is connected to a decision node via a standard edge. The decision node is recognizable by the number of incoming and outgoing edges. The decision node shown has two outgoing weighted edges, which are colored in blue and are selected depending on the respective probability. Depending on which weighted edge is selected, either the manual task "RflWt" or the automated task "WtOK" is executed. It should be noted that in a real environment, the next step is determined by the result of the previously executed manual task "ChkWt". However, in the current version of the WFML, there is only the possibility of making decisions based on the weighted edges' probabilities.

The manual task "RflWt" represents the process of refilling the water tank, while the automated task "WtOK" represents the output on the information display of the coffee machine. A merge node is used to merge alternative flows. The merge node is represented by an activity node with two incoming standard edges and exactly one outgoing standard edge. From the merge node, the outgoing standard edge leads to the automated task "ChkTp". The automated task "ChkTp" measures the brewing unit's current temperature and includes the child diagram shown in Figure 4.3. The diagram to check the temperature of the brewing unit consists of the automated tasks "ReadSensors", "ProcessData", and "DetermineTp". We want to point out that the diagram is only an example that allows us to represent the hierarchical structure of the WFML. The tasks presented here for measuring the temperature may differ from those of a real system.

The "ChkTp" task is followed by a further decision-making process. In this process, either the manual task "PreHeat" preheats the brewing unit to the required temperature or the manual task "KeepTp" keeps the current temperature. After the decision-making process has been completed, the alternative flows are merged by another merge node. The transition to the last task "Brew" is executed through a standard edge. This task is used to finally brew the coffee. Since the manual task "Brew" has no outgoing edge, the brewing process ends.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

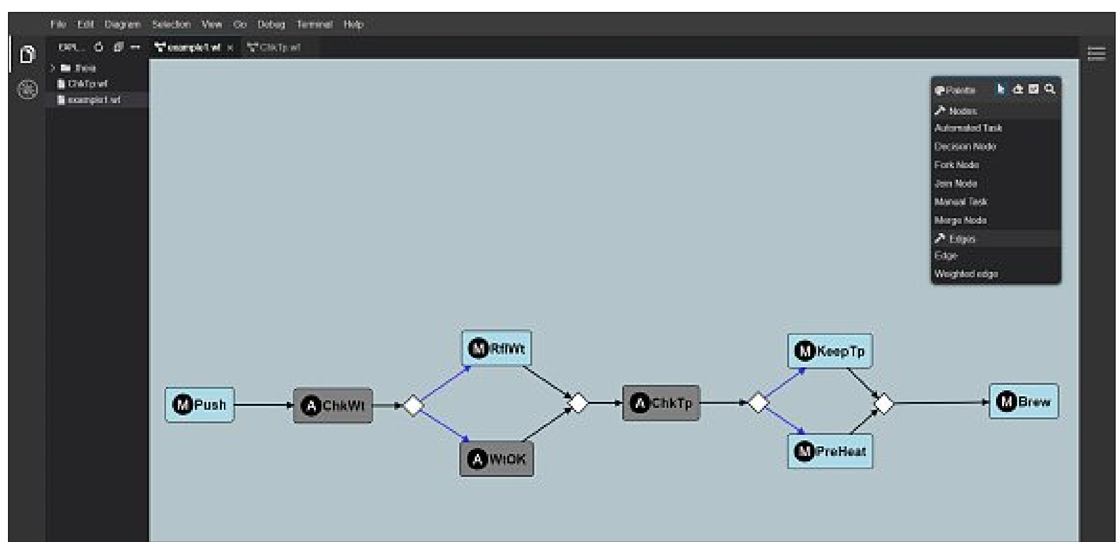

Figure 4.2: The WFML diagram editor with the brewing workflow example.

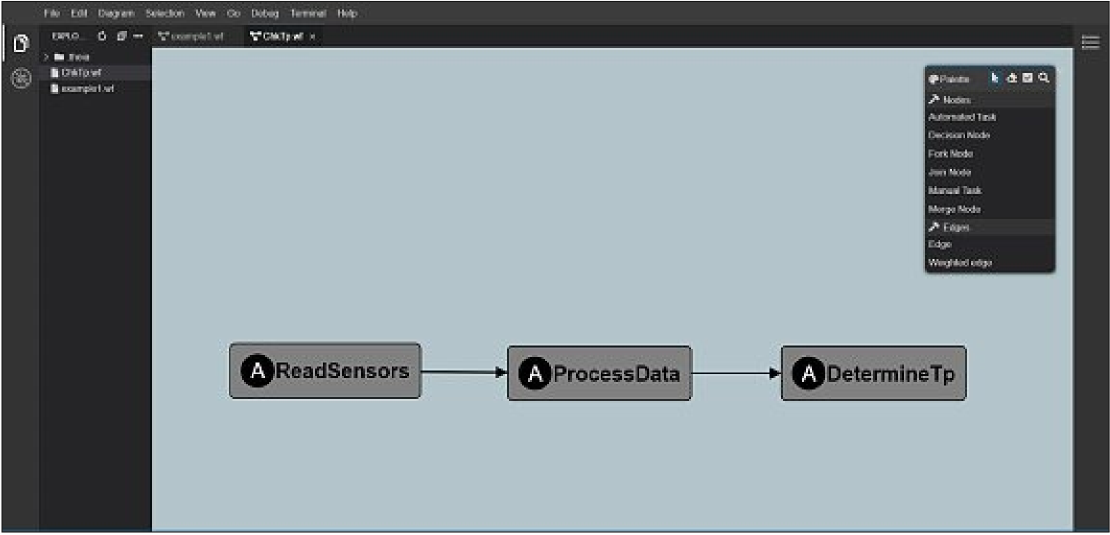

Figure 4.3: The WFML diagram to check the temperature of the coffee machine.

Diagram Editor

The GLS platform comes with a basic implementation for diagram editors. The diagram editor is part of the client framework provided by the GLS platform and can be deployed stand-alone or integrated into custom web-based IDEs such as Eclipse Theia or VSCode. These custom IDEs can be adapted using extensions developed by the communities of the IDEs. The basic implementation of a web-based IDE such as Eclipse Theia provides various workspace facilities (e.g. create, open, save file, repeat/undo command, paste,

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

copy, and cut). Additionally, one can also reveal and hide various window views, such as the debug view or the output console. Furthermore, through the integration of the diagram editor into the web-based IDE, GLSP specific functions are integrated into the IDE's graphical user interface. In the following, we will introduce some of the available features of the diagram editor:

- **Modification of the diagram view.** The GLSP diagram editor provides several features to modify the diagram.
  - **Center.** Aligns the desktop screen to the center of the diagram.
  - **Export.** Allows to create an SVG representation of the diagram and save it in the same workspace as the model source.
  - **Fit to screen.** Rearranges the view of the diagram so that all elements are visible.
  - **Layout.** The layout command aligns all elements of the diagram in an orderly manner.
  - **Align.** The align command realigns the individual elements within the diagram.
  - **Resize.** The resize command allows changing the size of the elements. The height or the width of the individual elements can be adjusted to the size of a selected element.
- **Tool Palette.** As already mentioned in Section 3.2 about the features of GLSP, available operations can be requested from the server utilizing the *request operation* action. The obtained operations can then be displayed via the Sprotty framework in a tool palette, as shown in Figure 4.2 on the right. The WFML diagram editor provides operations to create the previously presented language concepts of the WFML. The design of the tool palette can be individually adapted to the requirements of the language. The WFML tool palette has a *select on click* function represented by the mouse pointer icon and *delete on click* function represented by the eraser icon. To insert an element into the diagram, the element is picked from the tool palette (e.g. manual task), and added to the diagram by clicking on the desired position.
- **Hover Elements.** If the user moves the mouse pointer over the element in the diagram, a pop-up appears, displaying detailed information about the element (e.g. the duration of an automated task).
- **Command Palette.** By right-clicking within the diagram, another pop-up appears, which offers the user additional features. Depending on whether the user clicks on an existing element or a free position within the diagram, the user can either delete the existing element or create a new one. Further features can be added by extensions of the client or server framework and adapted to the languages used.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

- **Use of the Keyboard.** Besides using the mouse, individual features can also be controlled via the computer keyboard. Deleting elements with the delete key is an example.

4.2 Requirements for Debugging the Workflow Modeling Language

One of the aims of this work is to reuse existing frameworks and parts of the graphical user interface of the Theia-debug extension for textual languages, thus minimizing the effort required to introduce new graphical modeling language debuggers. In Section 2.3, we discussed the debugging concepts for textual languages, and afterward, we discussed how to translate these concepts for debugging graphical modeling languages. We realized that the basic functionalities needed for graphical modeling languages are very similar to those of the textual languages and, therefore, the existing components can be reused. For this reason, we use the existing open-source Theia-debug extension[3] for textual languages, which can be integrated into the Eclipse Theia framework. The Theia-debug extension comes with a generic graphical user interface for debugging features. The debugger user interface shown in Figure 4.4 offers visual representations on digital control panels to interact with the user. The debug view on the left side of Figure 4.4 includes various control commands and information views. Besides, there exists an additional debug menu tab, which lists all available control commands. In Section 4.3 we will discuss the individual control panels and information views. Furthermore, the Theia-debug extension supports the DAP, and offers various contribution points to adapt/extend the debugger facilities to the requirements for graphical modeling languages. This way, the specific debug adapter developed for the WFML will be integrated into Eclipse Theia. In the following, we will address the requirements for debugging the WFML and how the existing frameworks can be used to fulfill them.

Execution Modes

The user needs functionalities to start, pause, and stop the execution of the model. Furthermore, the user requires a graphical user interface that can be used to operate these commands and display relevant information to the user.

For this purpose, we first need to look at the DAP in terms of whether language-specific information is relevant for the requests to start, pause, and stop the execution. Secondly, we have to analyze if the debug window provided by the Theia-debug extension can be reused. Finally, we need an external debugger that is available on a remote server. This debugger should know the WFML language constructs and offer functions to debug and interpret a WFML model.

[3]https://npmjs.com/package/@theia/debug

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

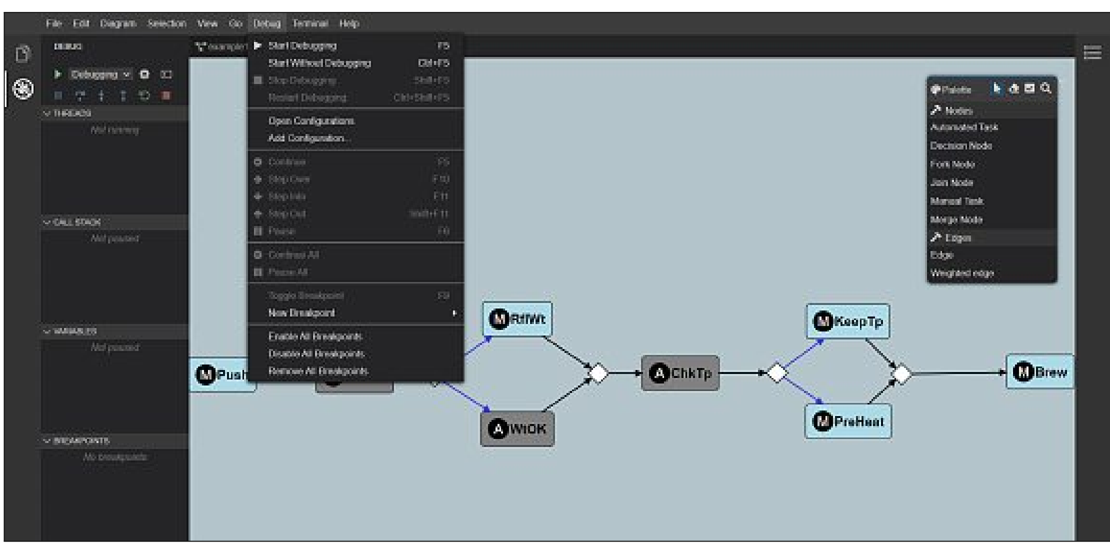

Figure 4.4: The debugger GUI of the Theia-debug extension.

1. **Language-specific debug request data:** Looking at the DAP and the corresponding messages to start the debug session, one can see that they are independent of the kind of language used. The initially exchanged messages only contain information regarding the development tool and the debug adapter. This allows reusing the components for starting the debug session without investing much time in adapting the components. Furthermore, pausing the debug session and the *pause* request for textual languages can be transferred to graphical modeling languages without any further adaptations. For this, the running thread's ID is passed to the adapter, which pauses the specific thread. After the thread has been paused successfully, the debug adapter first sends the response and then a stop event (with reason 'pause'). Upon-receipt, the development tool first requests all threads that exist at that point of time and then the stack trace for the thread mentioned in the stop event. Then the variables for the current stack frame are requested. The received information is displayed via the debug view. The development tool sends the DAP's *disconnect* request to terminate the debugger. The debug adapter sends the response, followed by a terminate event that indicates that debugging of the debugger has terminated.

2. **Debug window of the Theia-debug extension:** The debug window and the containing items such as icons, menus, and further windows of the Theia-debug extension allow the user to interact with the debugger tools. These components use a generic approach that supports various textual languages. Since the components are not dependent of the syntax or semantics of the language, these components can be reused to support graphical languages.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

3. **Language-specific debugger:** For the WFML debugger, a client-server architecture is required that allows the debug adapter to connect to the specific debugger and exchange messages. We assume that both the IDE and the debugger are on the same physical machine. Thus, we do not have to include the diagram file in the start request. We only send the path of the file to the debugger. In turn, the debugger must be able to parse the model, store it accordingly, and execute the model.

Steps

The user should be able to step through the model and execute one model element at a time. Before looking at which components can be reused or must be extended to implement the defined requirement for the WFML debugger, the question arises: "What does it mean to take a step within the WFML?". As already discussed in Section 2.3, it varies from language to language, what it means executing a step within the model. In the WFML, we use a step to execute a task or to control the workflow. Furthermore, the step-in and step-out commands are required for hierarchy structures. If the step-in command is used on a composite task, the child diagram should open, and the execution should stop at the first task in the child diagram. Using the step-out command within the child diagram, should open the parent diagram and stop the execution before the next step.

In the following, we analyze the DAP in terms of whether language-specific information is relevant for the requests to step, step in, and step out. Besides, the language-specific debugger requires methods to step through a model, step into a child model, and step out back to the parent model.

1. **Language-specific debug request data:** The components of the Theia-debug extension suitable for this purpose are based on the thread ID of the executed debug session. The *next* request of the DAP transmits the thread ID to the debug adapter, which in turn forwards the command to the debugger. The debugger performs a step and sends the information of the new stack frame back to the debug adapter, after the step is completed. Subsequently, the debug adapter sends a stop event, including the received information to the development tool.

   The DAP offers requests to transfer step-in and step-out information between the development tool and the debug adapter. These protocol messages depend on the ID of the current running thread. To step into the child model, the development tool sends the *stepIn* request to the debug adapter. The *stepOut* request forces the debugger to step back into the parent model. The debug adapter then forwards the information to the debugger.

2. **Language-specific debugger:** For the stepwise execution of the model, a concept must be developed in the course of the implementation of the WFML debugger, which executes the individual model elements one after the other. The model elements should be executed in the order in which they are connected in the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

diagram by the edges. The selection of an edge is performed randomly based on the specified weights. The debugger requires functionality to step into the child model and start a new debugging process for the child model. When the debugger is forced to step out of the child model, the debugging process of the child model should be finished and the execution should halt at the next model element of the parent model. Besides, a solution must be found on how the individual model elements can be interpreted to finally execute the task. In Chapter 6, we will deal with this issue in detail and present one solution.

Breakpoints

Like textual languages, where breakpoints can be set in the source code, the user should be able to set a breakpoint on a specific model element in the WFML diagram editor. To support hierarchical structures, each workflow is defined in a separate diagram that is stored in a GLSP model schema. These model schemas are treated independently in the graphical language server of the WFML. It should be noted that this allows model elements in different diagrams to have the same ID. Thus, breakpoints in a WFML model require a reference to the respective diagram in which they were set. A breakpoint should be visible after it has been set by marking the corresponding element. In addition to the graphical rendering of the breakpoint within the diagram, it is also necessary that a breakpoint window is available within the debug window, as known from conventional textual language debuggers. This additional display of the breakpoints makes sense because the user not only wants to see the breakpoints within one diagram but a list of all breakpoints within the whole project. Finally, the breakpoint window should provide functions to enable and disable breakpoints.

To implement the requirements mentioned above, we first need to analyze which existing components of the tools available can be reused and which ones have to be modified. On the one hand, the client framework of the GLS platform is required to implement the graphical modifications within the diagram. On the other hand, the Theia-debug extension is needed, to take over the management of the breakpoints and further to list all breakpoints in the breakpoint window. At the same time, it is important to analyze if one of the existing DAP requests for setting breakpoints in the WFML can be reused.

1. **Language-specific debug request data:** We first analyzed if the existing protocol messages for setting breakpoints in textual languages also meet the requirements for graphical modeling languages. In Section 3.3, we discussed source breakpoints, function breakpoints, data breakpoints, and exception breakpoints among the types of breakpoints supported by the DAP. Source breakpoints are set within the source code for textual languages and store the path of the source file as well as the line on which the breakpoint is located. On the other hand, graphical modeling languages have no lines, but consist of connected model elements. The elements of the WFML are based on the diagram framework Sprotty, which uniquely identifies

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

model elements via their ID. The GLSP uses the data type String for the ID of model elements, while the line in which a source breakpoint is located is transferred over the DAP as a number data type. This discrepancy between the different data types and the different representation of elements prevented us from using the *set breakpoints* request of the DAP for source breakpoints.

Subsequently, we looked at the *set function breakpoints* request that exists for function breakpoints, which are also provided by the DAP. Function breakpoints compared to source breakpoints are identified by the name of a method, whereas the property *name* is stored as a String data type. Model elements within the GLSP also have a property *name*, which is stored as String data type. This consistency in the data types would suggest reusing the *set function breakpoints* requests. However, function breakpoints are not defined for a single source file but are valid throughout the entire project. In the WFML, certain tasks with the same name may be defined within one diagram or the entire project. Therefore, the execution might encounter breakpoints that were not marked by the user. This would contradict the requirements we identified for the debugger to set breakpoints on individual model elements. Furthermore, the definition of a function corresponds to an entire workflow rather than a single task. The same applies to other DAP requests, such as exception breakpoints and data breakpoints, to send breakpoints to the debug adapter. Exception breakpoints pause the execution as soon as a certain exception is thrown. Data breakpoints interrupt the execution when the value of a certain variable changes. Furthermore, the data breakpoints and the exception breakpoints are transmitted to the debug adapter without the unique affiliation to a particular diagram.

Since none of the existing requests can be used for the requirements, a concept must be designed that allows the setting of *GLSPBreakpoints* and their transmission to the debug adapter. For this purpose, the Theia-debug extension offers the possibility to extend the protocol with custom requests. Therefore, a new request can be defined, which transfers the *GLSPBreakpoints* to the debug adapter. A *GLSPBreakpoint* requires, on the one hand, a property to store the ID of the model element for that a breakpoint should be set and, on the other hand, a property that stores the source, i.e., the path of the diagram in which the breakpoint is located. As mentioned initially, model elements can have the same ID in different workflows because each workflow is stored in a separate GLSP model schema. Therefore, we need the model element's ID and the path to the diagram in which the model element was defined. This allows the breakpoint to be unambiguously assigned to one element.

2. **Breakpoint window of the Theia-debug extension:** The components of the Theia-debug extension responsible for the management of breakpoints must be extended to support the newly defined *GLSPBreakpoints*. This also includes the display of breakpoints within the breakpoint window.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

3. **GLSP extension for breakpoints:** To implement the graphical requirements for the WFML debugger, the existing client framework of the GLS platform must be extended. In Section 3.2, we have already discussed the architecture of the GLSP in detail and mentioned, among other things, features that are supported by model elements. For the model elements of the WFML, to support breakpoints, a new breakpoint feature must be defined. The breakpoint feature is described in detail in Section 5.2. Furthermore, additional actions and the corresponding commands within the client framework of the GLS platform must be created to support the process of setting breakpoints. On the one hand, an action is needed that allows setting breakpoints and, on the other hand, an action that removes those breakpoints from the model elements. Besides, actions and their associated commands have to be defined to enable and disable breakpoints.

   Another point is the extension of the context menu, which appears when the user right-clicks on a model element. The commands for adding and removing breakpoints must be added to the menu and connected to the previously mentioned actions for setting and removing breakpoints.

   Finally, the breakpoint feature requires a decorator responsible for highlighting the model element's frame (e.g. displaying the frame in a reddish color). All these GLSP extensions have to be combined in a separate *GLSPDebugModule* to ensure easy integration into the client infrastructure. In Section 5.2, the breakpoint decorator and the *GLSPDebugModule* are presented in detail.

4. **Language-specific debugger:** The WFML debugger should be implemented so that the *GLSPBreakpoints* transmitted by the debug adapter can be stored based on the diagram source. If further breakpoints are added during execution, the current breakpoints should be overwritten by the new ones. As a result of running the model, each model element must be checked to see if there is a breakpoint at that position. If this is the case, further execution should be stopped, and the user should be informed. As soon as the user restarts the execution via the development tool, the WFML debugger should continue at the interrupted state.

Stack Trace

The user wants to view the procedure calls that are currently on the stack. Therefore, a call stack window is required that shows the order in which the nodes of the WFML are getting called. The stack frame which is currently being called is at the top of the stack. Below is the stack frame, which called the current stack frame. At the bottom of the stack lies the top call workflow from which the underlying workflow's call was started. Besides, the user wants to see the current stack frame within the diagram. The model element, which is currently executed and lies at the top of the stack (i.e. the current stack frame), should be highlighted (e.g. a green frame). As discussed in Section 2.3, the stack trace in model debugging is strongly bound to the terms stepping in and stepping out. Since the WFML supports hierarchy and composition, the call stack can be used to reflect these structures.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

For these requirements, we first have to analyze if the existing request for stack traces of the DAP can be reused. Second, we have to look at the components of the Theia-debug extension, whether they can be reused to control and display the stack trace. Third, we have to integrate new actions in the GLSP to highlight the current stack frame. Finally, the language-specific debugger requires functionalities to deal with the hierarchical structures and to determine the call stack.

1. **Language-specific debug request data:** For the exchange between the debugger UI of the IDE and the debug adapter, the DAP offers the *stack trace* request, which we already introduced in Section 3.3. The interface for stack frames stores the name of the frame as a String data type and contains an optional attribute *source*, which stores the frame's location.

2. **Call stack window of the Theia-debug extension:** The existing call stack window of the Theia-debug extension and the corresponding components necessary to determine the stack trace will be used. This is possible since the representation in the call stack window works with String data types. Thus, we need to send the IDs of the model elements as a String data type to the development tool.

3. **GLSP extension for current stack frame:** For the implementation of the graphical representation of the current stack frame, the client framework of the GLS platform must be extended with further actions. On the one hand, an action and the associated command to set the current stack frame is needed and, on the other hand, an action to remove the stack frame. In Section 5.2, we explain what actions and commands look like based on the action and the command that are necessary to add and remove breakpoints. The command belonging to the action checks whether the model element is valid as a stack frame or not. A model element is valid as a stack frame if it supports the stack frame feature. The feature, as well as the actions and commands, can be added to the *GLSPDebugModule*.

4. **Language-specific debugger:** The WFML debugger requires a separate process for each hierarchy level of the model, determining the current stack frame at each step. Thus, the call stack can be filled with the stack frames of the respective model levels. When the execution of child models has finished, the WFML debugger must jump back into the process execution of the parent model and continue the execution on the current stack frame of the model now present.

Runtime Variable I/O

When the model's execution is paused, the user requires a view where the variable values of the current stack frame are shown.

To implement these requirements, we first have to analyze the requests for variables of the DAP. Second, we will look at, how we can reuse the variable view of the Theia-debug extension. Finally, the implementations for the language-specific debugger will be derived.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

1. **Language-specific debug request data:** In the course of representing the key features of the DAP, we have already presented the *variable* request, which is sent to the debug adapter from the Theia-debug extension. Furthermore, variables in the DAP are defined by the name of the variable, the value of the variable, and the type of the value. The variables are sent from the WFML debugger, where the variables are stored in terms of the textual languages (e.g. Java) in which the debugger is implemented. Therefore, a specific mapping to reuse the *variable* request for graphical languages is not required. This aspect further allows reusing not only existing components for managing and displaying the variables in the Theia-debug extension but also the corresponding protocol messages of the DAP. The debug adapter should forward the *variable* request to the WFML debugger, which determines the variables of the current element (e.g. duration of an automated task). The WFML debugger should then send back the results via the debug adapter, which in turn uses the DAP's *set variable* response to forward the variables to the development tool or, more precisely, to the Theia-debug extension. The extension should be responsible for displaying the received variables in the graphical user interface.

2. **Variables window of the Theia-debug extension:** The Theia-debug extension has a variable view, which is part of the debug view. As with the call stack view, the variables are also presented as String data types. This allows us to reuse the existing components for our requirements.

3. **Language-specific debugger:** The WFML debugger requires functionalities to parse and store variables of a model element. After each step or the occurrence of an exception, the current variable, as well as the corresponding values, should be sent to the development tool via the debug adapter.

Exceptions

The user wants to be informed in case an exception occurs during the execution of the model and what caused the exception. This must be presented to the user, who usually only has knowledge of the modeling language, in an understandable form.

First, we have to analyze, how an exception can occur in the model interpretation approach. Second, we will look at the debug console of the Theia-debug extension to display exception information. Finally, we discuss the language-specific debugger implementation.

1. **Definition of exceptions in a specific language:** In Section 2.3, we talked about the levels at which exceptions occur and how they can be treated. Looking at the WFML, the grammatical rules defined in the abstract syntax can be used to ensure that errors are avoided when creating models, and thus only correct models are constructed. Nevertheless, errors may occur during runtime, or more precisely, in the case of the execution of the synthesized artifacts (e.g. the model schema). We mentioned in the introduction of the GLSP that the graphical model is stored

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

as a JSON object. Using this object, the model can be parsed through the WFML debugger, and finally, the elements executed step by step. In this parsing process, errors may already occur, which were caused by an incorrect translation of the graphical model into the JSON format. During the execution, further exceptions can be triggered due to programming errors in the real debugger or the frameworks used.

2. **Debug console of the Theia-debug extension:** For the implementation of exceptions, the components of the Theia-debug extension can be used again, since displaying of exceptions is usually a single output on the debug console. In Section 3.3, we discussed the possible variants for the transfer of exceptions via the DAP. In the following, we go into more detail about the approach using the debug console for showing the exceptions to the user. For this purpose, we would like to take as an example a situation that leads the execution of the model into an error state. The WFML allows decision nodes to be used with both standard edges and weighted edges, of which the latter contain a certain probability of being selected during execution. When running the model with standard edges, however, there is an error state of the execution engine, which can be attributed to the fact that the execution engine did not find any probabilities in this situation and could not continue the execution. The resulting *NullPointerException* is described in terms of the language in which the debugger was written (e.g. Java for the WFML debugger). This exception has to be *catched* and converted into domain-specific terms. In the case of the example, the output should contain the location (e.g. the model element) where the error occurred, the cause, and, if possible, suggested solutions. The WFML debugger should stop the execution and transmit the exception information to the debug adapter, which, in turn, should generate a stop event informing the development tool about the stop of the execution due to an exception. Through an output event, the debug adapter should send all information provided to it by the WFML debugger to output the exception in the development tool's debug console.

3. **Language-specific debugger:** The WFML debugger must detect the error and provide the appropriate exception handlers that can handle the occurring exceptions. Different exceptions may occur depending on the language, which is why exceptions handlers must be adapted to the languages used. Subsequently, the debugger must be able to stop execution after an exception occurs and send the respective information such as the call stack, variable values, and the exception information to the debug adapter. The debug adapter should only be responsible for forwarding the information it receives from the WFML debugger to the development tool.

Requirements Overview

Table 4.1 gives an overview of the requirements for debugging the WFML. The cells colored in green represent the existing components we can reuse, whereas the cells colored in red represent the components we have to implement/adapt. The shortcut

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

"NA" stands for "No Action", i.e., no implementation or adaption is required. In the first column, the requirements mentioned above are listed. The second column shows which components we can reuse from the Theia-debug extension. The third column shows the requests and events that can be used from the DAP. The fourth column represents the modification using the GLSP to integrate graphical animations. The fifth column shows the implementations necessary for the WFML debug adapter. Therefore, we will extend the vscode-debugadapter module[4] to implement the requirements. The last column lists all requirements for the debugger implementation. This part is the most affordable since we have to implement the debugger from scratch.

Requirement	Theia-debug extension (GUI)	DAP	GLSP	Debug Adapter (vscode-debugadapter)	Debugger
Start	command view	launch request	open diagram	connect to debugger, send model	parse, execute, and interpret model
Pause	command view	pause request	NA	forward pause request	pause execution
Stop	command view	disconnect request	NA	forward stop request	stop execution
Step	command view	next request	NA	forward step request	get next element, send element
Step In	command view	stepIn request	open child diagram	forward stepIn request	step into child model
Step Out	command view	stepOut request	open parent diagram	forward stepOut request	step out of child model
Breakpoints	Extending the breakpoint view for GLSPBreakpoints	setGLSPBreakpoints request	add/remove & enable/disable breakpoints	send GLSP-Breakpoints	stop execution on breakpoint
Stack Trace	call stack view	stackTrace request	highlight current stack frame	response stack trace	send stack trace
Variable	variable view	variables request	NA	response variables	send variables
Exceptions	debug console	output event	NA	response exception	handle exception

Table 4.1: Requirements for Debugging the WFML.

4.3 Workflow Modeling Language Debugger

This section introduces the developed debugger for the WFML. We implemented the previously defined requirements and concepts to provide the user with a debugger that supports the debugging concepts we discussed in Chapter 2.

Using the example of a model in the WFML introduced in Section 4.1, we want to present the features of the WFML debugger. In Figure 4.5, the web-based IDE containing the debug view window, which we were able to reuse from the Theia-debug extension, is shown. The debug view is further subdivided into a command view, threads view,

[4] https://npmjs.com/package/vscode-debugadapter

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

call stack view, variables view, and breakpoints view. Within the command view, the following buttons are provided to control the execution:

- **Play.** Starts a new debug session and opens the diagram to be debugged.
- **Continue.** Starts the debugger to run again after the execution was paused.
- **Step Over.** Starts the debugger to run again for one step.
- **Step In.** Forces the debugger to step into a child diagram. If there is no target to step into, the step-in command behaves like step-over.
- **Step Out.** Forces the debugger to step out from a child diagram.
- **Restart.** Restarts the execution of the debug session.
- **Stop.** Stops the execution of the debug session.
- **Pause.** Pauses the execution of the debug session. Only visible while the execution is continued.

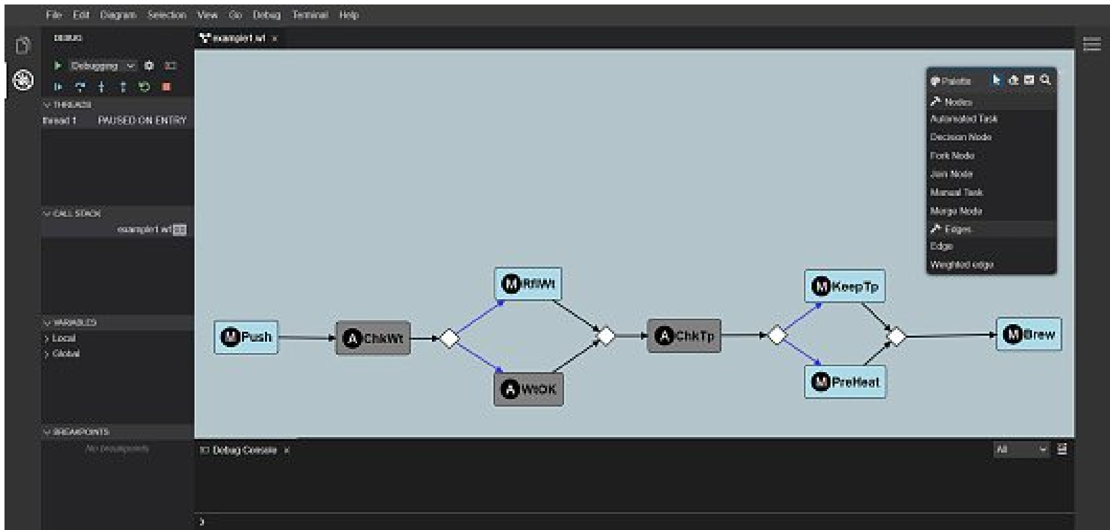

Figure 4.5: The WFML debugger is paused directly after starting the debug session.

In the threads view, the current thread is displayed, and the status in which the thread is located. In Figure 4.5, the thread with the ID *1* has the status "Paused on Entry". This means that the debug session was started, and the execution stopped immediately before the first element. Stopping before the first element is a debug adapter specific setting, which is specified using a boolean flag during the initialization phase between the development tool and debug adapter. A missing flag would mean an immediate execution of the model after the start of the debug session. However, if a breakpoint was set, the execution halts at the breakpoints location (e.g. the model element). As described in

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Section 3.3, the DAP also supports multiple threads, which, however, for the sake of simplicity, were left out of the WFML debugger, and thus only one thread is treated.

After a debug session has been started, the user can start to step through the model using the step commands. Figure 4.6 shows the debugger after taking the first step, where *thread 1* changed to the status "Paused on Step". The information about the current stack frame transmitted by the WFML debugger is now clearly displayed in the debug view. Underneath the threads view, the call stack view is located, which represents the ID of the first model element within the diagram. The ID of the model element is displayed because, on the one hand, the WFML is a language that supports the reuse of certain tasks in the model and, on the other hand, because edges and activity nodes both have no labels. Next to the ID of the model element, the source of the diagram in which the element is stored is shown in the call stack view. Furthermore, the client framework of the GLS platform was extended to add functionality to change the graphical representation of the current stack frame within the diagram. Figure 4.6 shows that the model element that represents the current stack frame is highlighted through its green frame.

Another requirement for the WFML debugger was to display the variables of the model elements in a readable form. For this, the variable view, which is shown in Figure 4.6, was used. The variables for the current element are listed based on the variable name and the associated value. Besides, if the mouse hovers over the name of the variable, the corresponding data type is displayed.

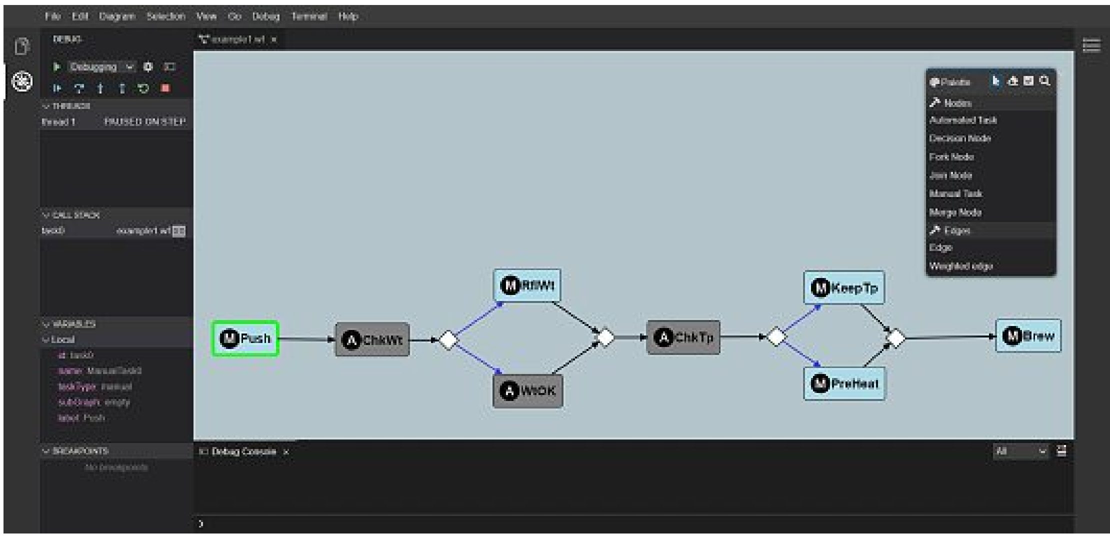

Figure 4.6: The WFML debugger is paused at the first model element and shows the information corresponding to the current stack frame.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

TU Bibliothek Your knowledge hub WIEN

In addition to stepping through the model, the WFML debugger also offers the feature to jump into a child diagram. This changes the current stack frame and the call stack. Figure 4.7 shows the new call stack. In this case, the first task within the newly opened diagram is on top of the stack frame of the parent model, as shown in the call stack view. In addition to the IDs of the two tasks, the sources of the respective diagrams are shown. Another feature of the debug view allows opening the corresponding diagram in the editor by double-clicking on the respective source. Thus, the user can easily switch between the diagrams. The new stack frame, recognizable by the green frame, has been moved to the child diagram's first element. In the further course, the user can use the step-out command to jump back into the higher-level diagram. The remaining tasks of the child diagram are completed, and the execution is paused again before the execution of the next element of the parent diagram. Instead of using the step-out command, the user can also step to the child diagram's last task and return to the parent diagram via a further step command.

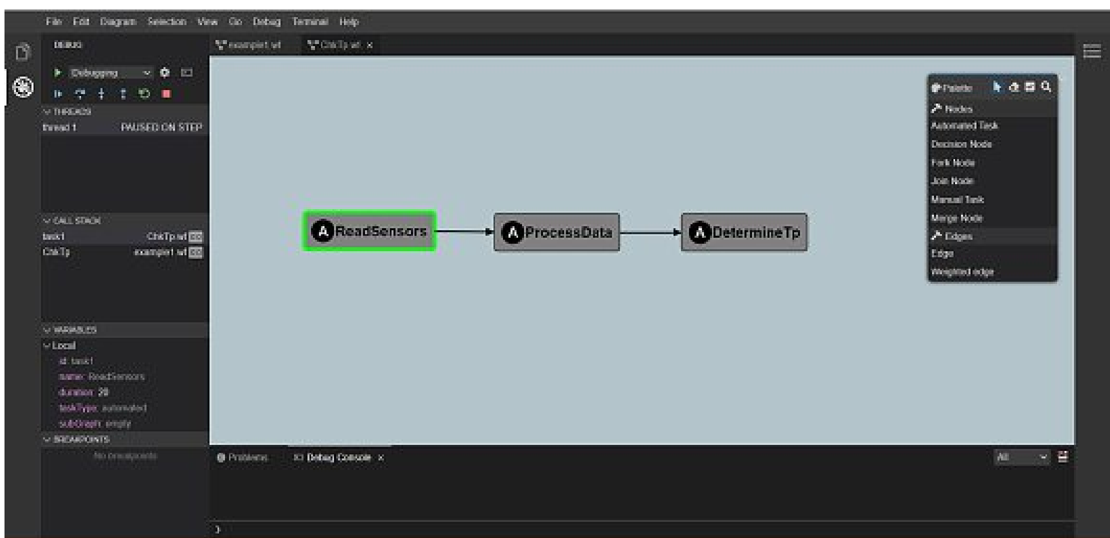

Figure 4.7: After the step-in command was executed, the debugger opens the child model.

The user can right-click on a model element to add or remove a breakpoint. After that, the context menu pop-up appears, which is also used to add and remove elements. Figure 4.8 shows the pop-up with the add breakpoint and remove breakpoint commands. After the breakpoint has been marked, it appears in the breakpoints view, which is part of the debug view. As in the call stack view, the element's ID is used in the breakpoints view to identify the breakpoint uniquely. Based on the source, the user knows in which diagram the breakpoint is located and can immediately open the corresponding diagram by double-clicking on it. Besides, the client framework of the GLS platform was extended as described in Section 4.1 with further actions to highlight an model element with a red frame, if it was marked as a breakpoint. In Figure 4.9, two elements from different diagrams were marked with a breakpoint, and therefore the color of the frames changed to red.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

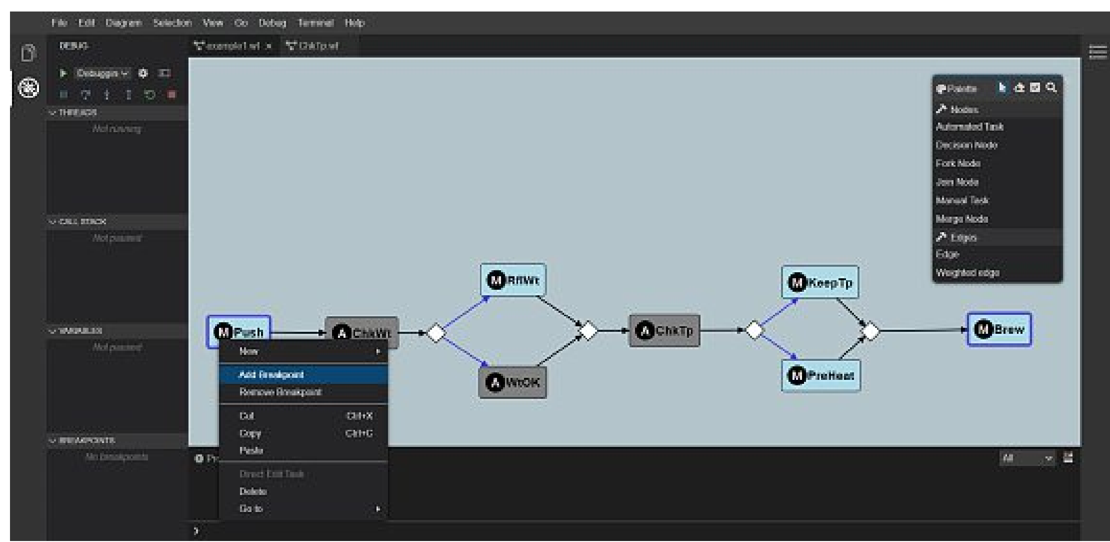

Figure 4.8: The context menu appears after a right-click on one or more model elements and provides the option to add and remove breakpoints.

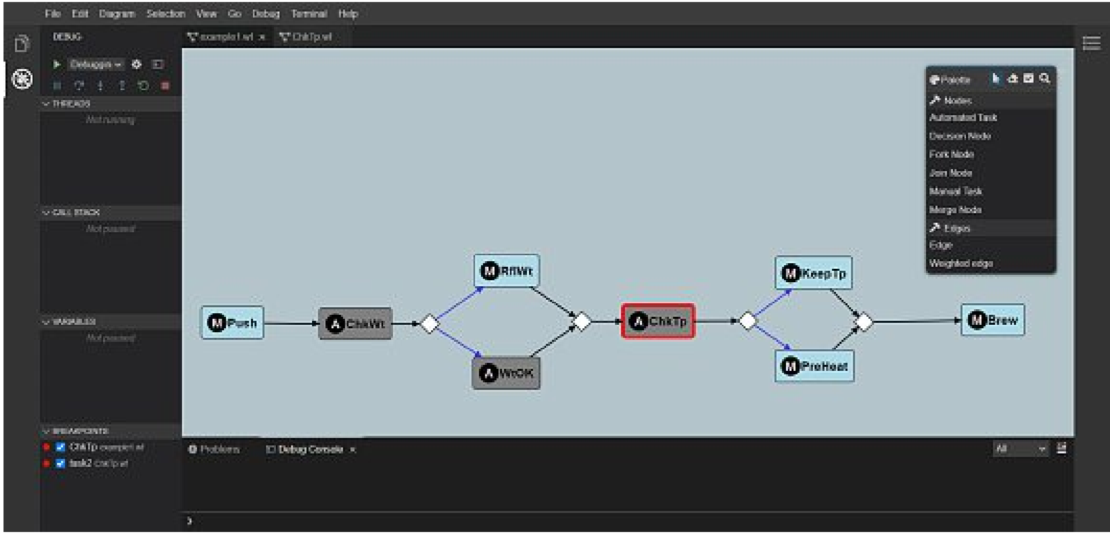

Figure 4.9: After breakpoints were set in the individual models, they appear in the breakpoint window and the affected elements are highlighted with a red frame in the WFML editor.

If a breakpoint is set, it is enabled by default. However, the user may disable breakpoints, so they are not included in determining when the debugging process should stop next. A breakpoint that is no longer required during debugging can be deleted entirely. In Figure 4.10, the possible options to enable, disable, and remove breakpoints through

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

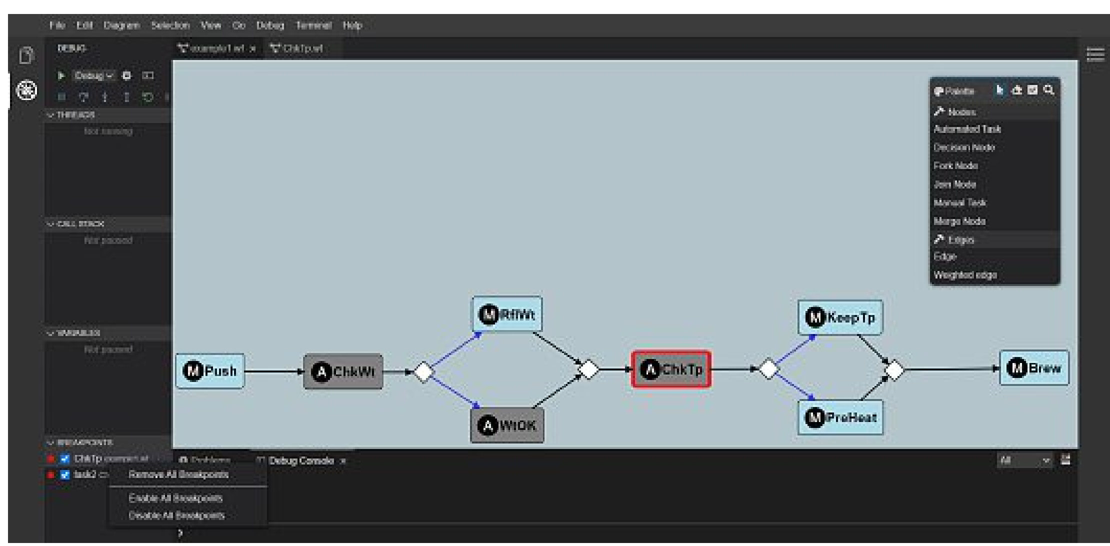

Figure 4.10: Further commands to enable or disable breakpoints appear after right-clicking into the breakpoints window.

the breakpoint view are shown. By right-clicking into the breakpoint view, a new pop-up appears that includes the commands to remove all breakpoints, enable all breakpoints, and disable all breakpoints. These commands apply for the whole project and also remove the red frame marking a breakpoint on a model element. Besides enabling and disabling all breakpoints, it is also possible to only enable and disable a single breakpoint through the checkbox shown next to the ID of a breakpoint. Figure 4.11 shows two breakpoints in their respective diagrams, whereas the first breakpoint is unchecked and thus disabled. The coloring of the element border in the diagram disappeared as well.

After the user has set the breakpoints and started the execution of the model, the breakpoints are transmitted to the WFML debugger using the *set GLSPBreakpoints* request. If the debugger encounters a breakpoint during execution, the further execution is stopped, and the halt of the debugger is communicated to the user. Figure 4.12 shows a situation where exactly this case occurred. The executed thread changed to the status "Paused on Breakpoint". The call stack view shows the current stack frame, and the variable view presents the corresponding variables. Within the diagram, the red-colored frame, which marks a breakpoint, has changed to a green frame to display the current stack frame. Using the step commands, the user can step through the remaining elements of the model or continue the execution with the continue command.

There may also be errors in the execution of the model, so one of the requirements for the WFML debugger was that exceptions are *catched* and presented to the user in an understandable form. For this use case, the example model was changed such that another outgoing edge was connected to a task. The modified model example can be seen

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

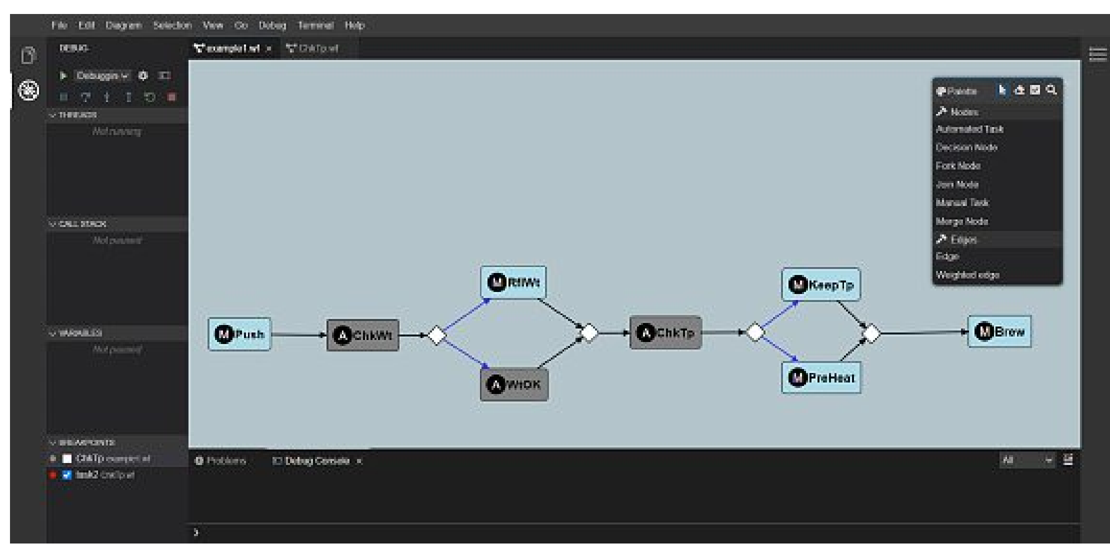

Figure 4.11: Disabling the breakpoint removes the breakpoint from the execution process.

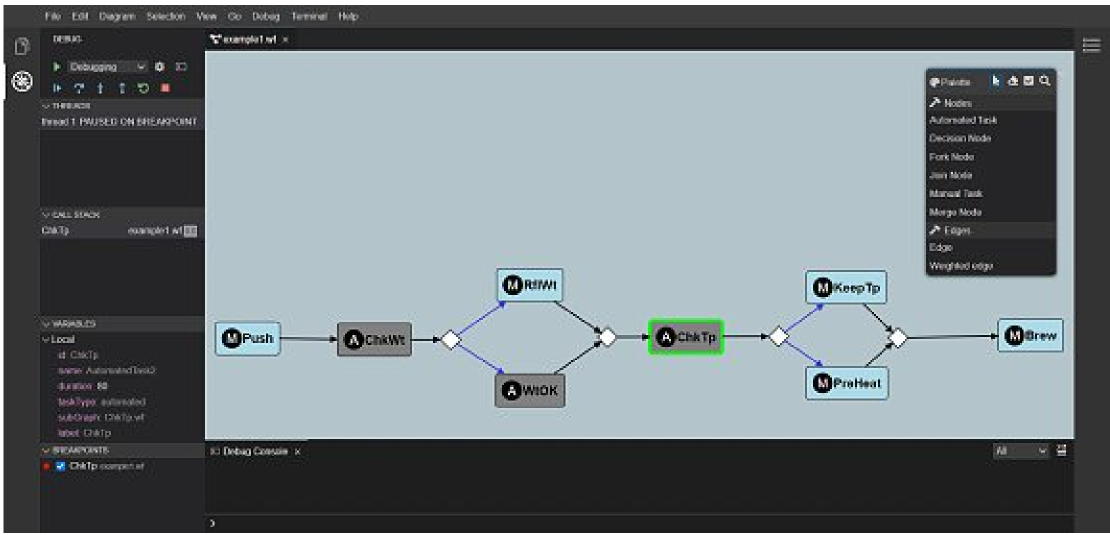

Figure 4.12: The WFML debugger paused on a breakpoint and presents information about the current stack frame.

in Figure 4.13. In addition to the previously existing edge between the automated task *ChkWt* and a decision node, a further edge between the task *ChkWt* and the automated task *WtOK* was added. The WFML debugger parses all model elements at the start of the debug session. The elements will then be executed step by step in the sequence they are connected. However, when the execution hits the *ChkWt* task, the debugger only expects one outgoing edge from the task. Several outgoing edges are known to the WFML debugger only in the context of activity nodes. In the case of a task having

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

several outgoing edges, the debugger is in an error state and can no longer continue the further execution of the model. The debugger changes to the status "Paused on Exception". An exception message is shown, which provides the location of the error, the cause, and a suggested solution for the occurred error. As shown in Figure 4.13, the exception is output via the debug console. The user is informed that alternative flows are not allowed on the element and that a decision node should be used instead. In addition to the output of the exception in the debug console, the current stack frame is shown graphically in the diagram as well as in the debug view.

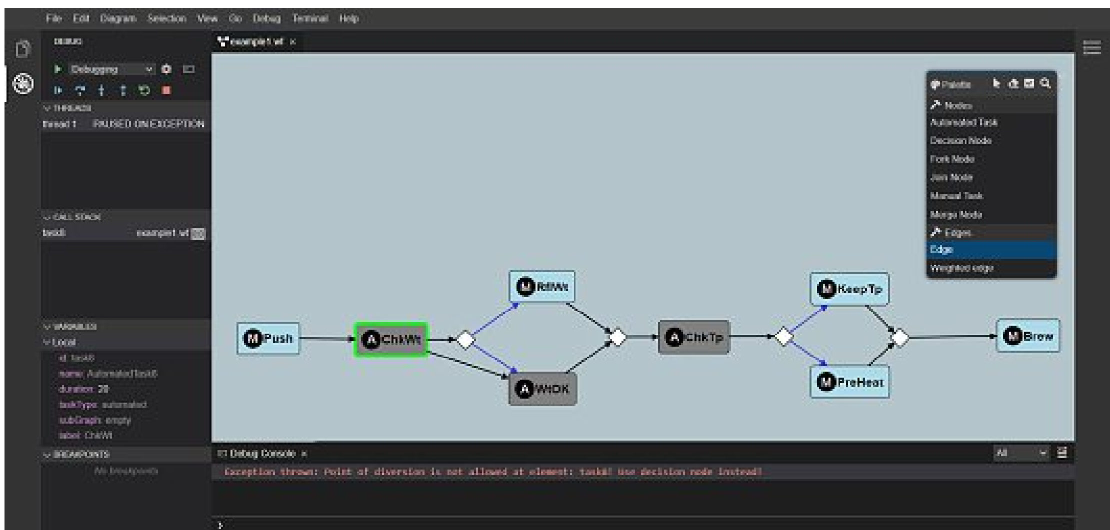

Figure 4.13: The WFML debugger paused on an exception and presents the information about the current stack frame and the thrown exception.

In summary, the WFML debugger provides all the necessary features of a modern debugger, and graphical animations have supplemented these. In the course of the development, existing frameworks available for textual languages could be reused or adapted for the requirements of a debugger for graphical languages. Finally, the DAP was successfully used in connection with the GLSP. For the interested reader, we refer to Chapter 5 for the detailed architecture and implementation of the WFML debugger. In Chapter 6, we will define another DSML and implement a debugger for this new language based on the developed framework.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 5

Architecture and Implementation

This chapter presents details on the implementation of the WFML debugger. We first define the technologies used besides the web-based modeling tools, namely Eclipse Theia and InversifyJS. After that, we show how the main debugging concepts described in the previous chapter can be implemented. Finally, we present the challenges we discovered during the implementation of the WFML debugger. Most of the challenges not only affect the WFML in particular but are also relevant for other DSMLs. The implementation of the WFML debugger GUI adaption[1] including the WFML debug-adapter, and the WFML debugger[2] is available on GitHub.

5.1 Technological Background

This section introduces technical background information on the two frameworks Eclipse Theia and InversifyJS used in the practical part of this thesis.

5.1.1 Eclipse Theia

Eclipse Theia[3] is an open-source framework for creating desktop or web-based IDEs. To support both variants with a single source, Theia executes two separate processes. One of these two processes is the front-end, representing the client and responsible for rendering the user interface. The other process is the back-end running on Node.js[4]. The communication between the two processes works through JSON-RPC messages that are exchanged via WebSockets[5] or a REST API[6] via HTTP[7]. In the desktop-based version

[1] https://github.com/EderH/graphicalLSP
[2] https://github.com/EderH/gml-interpreter
[3] https://theia-ide.org
[4] https://nodejs.org/en
[5] https://tools.ietf.org/html/rfc6455
[6] https://restfulapi.net
[7] https://w3.org/Protocols

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Electron, however, both front-end and back-end run locally. On the other hand, in a remote context, the back-end would run on a remote host. Theia has been designed so that a simple extension of the functionalities is possible at any time. For this purpose, Theia supports two different expansion mechanisms, namely extensions and plug-ins. They may make use of services and contribution points.

Extensions. Extensions allow easy access to existing extensions like the Monaco Editor[8] and adapt/extend it to one's custom requirements. For the communication between the individual extensions, the Theia framework uses dependency injection, which adds the extensions to the remaining components during the build time. We will present Dependency Injection in Section 5.1.2.

Plug-ins. Compared to extensions, plug-ins do not block Theia core processes since the code is running in a separate process. Plug-ins can be installed/uninstalled during runtime without recompiling the whole IDE. This makes sense in the case of a tool provider who delivers a full custom solution, where customers can integrate their own plug-ins without affecting the stability of the base tool. In order to contribute a new plug-in, Theia provides several APIs in the Theia API extension *Theia-plugin*.

Services. Services can be used by components of other extensions. Whereas plug-ins stick to the APIs offered by the Theia extension for Theia-plugins. For example, one instance can provide the *SelectionService*, so that other extensions can get an instance injected and use it. The *SelectionService* can be reused and extended to select a model element of the WFML.

Contribution Points. In order to provide a hook within extensions for others to contribute, one should define a contribution point. The contribution point is defined as an interface and can then be implemented by other extensions. An example of a contribution point is the *OpenerService* within the "Editor Extension", which allows registering *OpenHandlers*. *OpenHandlers* are necessary to open a new widget within the editor. The *OpenHandler* gets bound to the component through a contribution provider. A contribution provider is basically a container for contributions where contributions are instances of a bound type. In this work, we bound the *OpenHandler* to the diagram editor to open the diagram widgets.

In order to create an IDE with Theia, a package.json file is required that contains the package metadata like name and version as well as the runtime and build-time dependencies. There are already various Node.js packages available, which can be integrated into custom IDEs and subsequently adapted, depending on which individual requirements are placed on the IDE.

[8] https://microsoft.github.io/monaco-editor

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Debug Extension

Figure 5.1 gives an overview of the Theia-debug extension architecture, focusing on the interaction between the debug session and the debug adapter. When a user decides to debug an application they request a specific debug configuration, which will be resolved by the debug adapter. In the running example of the thesis, the launch configuration requests the WFML debug adapter. The launch configuration for the WFML is described in Section 5.2.4. The debug adapter can modify a passed debug configuration by adding, changing, or removing attributes. This already happens before the debug adapter is started. The debug adapter is integrated into the Theia back-end process (server). After the debug adapter is started a new debug session is initialized on the front-end process (client). The debug session handles the communication between the client and the server. The debug session manager controls the debug sessions on the client-side. This includes starting a new debug session for the received debug configuration, managing breakpoints across debug sessions, and terminating a running debug session. The client session factory is used to integrate a custom implementation of the debug session. If a new debug adapter needs to be added, the debug contribution is used as a contribution point. The contribution provides all available debug configuration types and tells the server-side architecture how to start the debug adapter. The adapter factory is used to start the debug adapter inside a separate container. After the debug adapter is started, the debug adapter session gets started. The debug adapter session works as a proxy between the client and the debug adapter itself. To customize the debug adapter session the adapter session factory can be used. The debug service provides functionality to configure and start a new debug adapter session based on the debug configuration. The debug adapter session manager is responsible to manage all available debug adapter sessions and, therefore, to create or remove a debug adapter session from the running application [55].

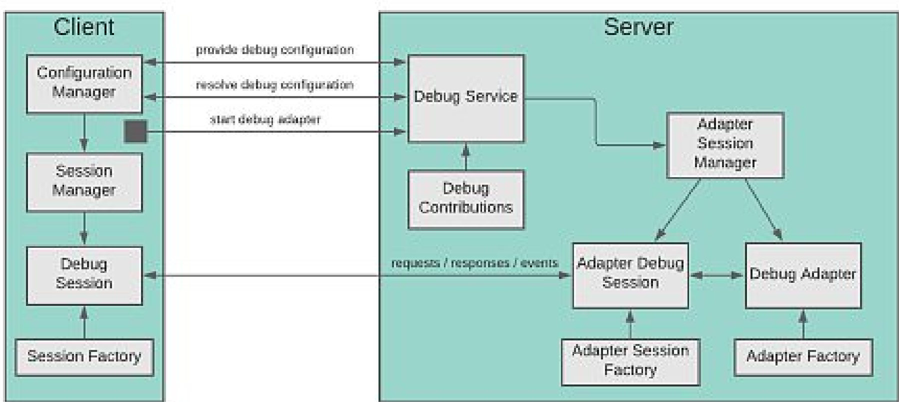

Figure 5.1: Communication between the Theia front-end process (client) and the Theia back-end process (server) in the Theia-debug extension (adopted from [55]).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Furthermore, the Theia-debug extension has a generic debugger graphical user interface which we reused. The debug windows are implemented as widgets, which can be integrated into the existing Theia graphical user interface via dependency injection. These widgets get the information from the debug session, which in turn gets all the information from the debug adapter. This allowed us to manipulate the flow of information so that information about graphical modeling languages can also be displayed in the debug windows of the Theia-debug extension graphical user interface.

5.1.2 InversifyJS

The open-source framework InversifyJS[9] is a lightweight inversion of control (IoC) container for TypeScript and JavaScript applications. Inversify injects the different dependencies into the component at creation time. Listing 5.1 shows how Inversify allows a class to have a direct dependency on other classes. In the example, this can be seen in the classes *Katana*, *Shurikan*, and *Ninja*. Each class that needs to be injected must be annotated with the *@injectable* decorator. Through the constructor, the two classes *Katana* and *Shuriken* are injected into the *Ninja* class, and subsequently, the methods of the injected classes are accessed from the *Ninja* class. In Line 25 of the example, a container is created. The container API provides some helpers to resolve multi-injections and ambiguous bindings.

Furthermore, there are additional decorators like *@inject* and *@postConstruct*. The *@inject* decorator marks the classes to be injected within the constructor or variable declaration. However, this is only necessary for interfaces, since an identifier must be defined for them. On the other hand, classes work without this identifier because the TypeScript compiler generates the metadata for it. The *@postConstruct* decorator can be used to annotate class methods. These methods are then executed after an object has been instantiated, but before any activation handler has been executed. This is useful if initialization logic should be performed after the constructor was called, but the component has not yet been initialized. Besides, other useful features provided by Inversify are used in the course of Eclipse Theia. It is advantageous to familiarize yourself with the basic features of InversifyJS before using Eclipse Theia.

[9]https://github.com/inversify/InversifyJS

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
1  import { Container, injectable, inject } from "inversify";
2  @injectable()
3  class Katana {
4    public hit() {
5      return "cut!";
6    }
7  }
8  @injectable()
9  class Shuriken {
10   public throw() {
11     return "hit!";
12   }
13 }
14 @injectable()
15 class Ninja implements Ninja {
16   private _katana: Katana;
17   private _shuriken: Shuriken;
18   public constructor(katana: Katana, shuriken: Shuriken) {
19     this._katana = katana;
20     this._shuriken = shuriken;
21   }
22   public fight() { return this._katana.hit(); };
23   public sneak() { return this._shuriken.throw(); };
24 }
25 var container = new Container();
26 container.bind<Ninja>(Ninja).to(Ninja);
27 container.bind<Katana>(Katana).to(Katana);
28 container.bind<Shuriken>(Shuriken).to(Shuriken);
```

Listing 5.1: InversifyJS example using dependency injection for classes [56].

5.2 Implementation Details

This section provides implementation details about the practical part of this thesis. It demonstrates how the client framework of the GLS platform can be extended to add the graphical requirements derived in the previous chapter. Furthermore, we present the contribution points to implement new debuggers in Theia for further graphical modeling languages.

5.2.1 GLSP Extensions for Breakpoints

The client framework of the GLS platform offers contribution points to extend the existing functionalities with further actions. For the implementation of the graphical representation of the debug features, it was necessary to introduce new actions. Lines 1-5 in Listing 5.2 shows the structure of the *AddBreakpoint* action, which is triggered as soon as the user sets a new breakpoint on an element. The elements selected by the user are passed to the constructor of the action. In Lines 6-27, the corresponding *AddBreakpointCommand* is listed. Command injections are created via dependency injection and should take the respective action as an injected constructor parameter. Commands must define a static constant KIND, which is used to map an action kind. Commands further consist of the methods *execute*, *undo*, and *redo*. The *AddBreakpointCommand* extends the abstract class *SystemCommand*, which is a special subtype of the *Command* class. There are other subtypes like the *ResetCommand*, *PopupCommand*, *HiddenCommand*, and *MergableCommand*. Compared to the other commands, the *SystemCommand* is not

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

placed on the command stack. This means that *SystemCommands* cannot be influenced by redo and undo operations.

```
1  export class AddBreakpointAction implements Action {
2    static readonly KIND = 'addBreakpoint';
3    kind = AddBreakpointAction.KIND;
4    constructor(readonly selectedElements: SModelElement[]) { }
5  }
6  @injectable()
7  export class AddBreakpointCommand extends SystemCommand {
8    static readonly KIND = AddBreakpointAction.KIND;
9    constructor(@inject(TYPES.Action) public action: AddBreakpointAction) {super();}
10
11   execute(context: CommandExecutionContext): CommandReturn {
12     const index = context.root.index;
13     for (const selectedElement of this.action.selectedElements) {
14       const element = index.getById(selectedElement.id);
15       if (element && hasBreakpointFeature(element)) {
16         element.breakpoint = true;
17       }
18     }
19     return context.root;
20   }
21   undo(context: CommandExecutionContext): CommandReturn {
22     ...
23   }
24   redo(context: CommandExecutionContext): CommandReturn {
25     ...
26   }
27 }
```

Listing 5.2: The action and the corresponding command for adding breakpoints to model elements.

The execute method in Line 11 receives a parameter of the type *CommandExecutionContext*. The *CommandExecutionContext* stores the model root through which the model elements can be accessed using the ID. Furthermore, the context contains the *ModelFactory*, which converts the serializable model schema into the model representation. Within the execute method, one can iterate over the selected elements and subsequently check whether the model element is an element that supports the breakpoint feature or not. Listing 5.3 shows the *BreakpointElement* that extends the *SModelExtension* to add additional properties to the model element. Regarding this work, we added a *breakpoint* flag to identify a model element as a breakpoint. Using the *hasBreakpointFeature* function presented in Listing 5.3, the model element is verified for whether it supports the breakpoint feature. If this is the case, the function returns a *BreakpointElement* where the breakpoint flag can be set or removed in the *execute* method. After the action is finished, the model root is returned.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
1  export const breakpointFeature = Symbol('breakpointFeature');
2
3  export interface BreakpointElement extends SModelExtension {
4    breakpoint: boolean;
5  }
6
7  export function hasBreakpointFeature(element: SModelElement): element
8    is SParentElement & BreakpointElement {
9    return element instanceof SParentElement && element.hasFeature(breakpointFeature);
10 }
```

Listing 5.3: Model element extension for the breakpoint feature.

Furthermore, we implemented the following actions and their corresponding commands:

- *RemoveBreakpointAction* and *RemoveBreakpointCommand*: Responsible for removing an active breakpoint.
- *AnnotateStackAction* and *AnnotateStackCommand*: Responsible for annotating a model element as current stack frame.
- *ClearStackAnnotationAction* and *ClearStackAnnotationCommand*: Responsible for removing the annotation of the current stack frame from a model element.
- *DisableBreakpointAction* and *DisableBreakpointCommand*: Responsible for removing the graphical representation of an existing breakpoint. The breakpoint is not deleted from the breakpoint view.
- *EnableBreakpointAction* and *EnableBreakpointCommand*: Responsible for restoring the graphical representation of an existing breakpoint after the breakpoint was disabled.

Finally, we needed a *stackFrameFeature*, which can be supported by model elements. For this, we expanded the model element by a *current* flag. The *AnnotateStackCommand*, *ClearStackAnnotationCommand*, and the *StackFrameDecorater* check whether the model element supports the *stackFrameFeature* and can be represented as a current stack frame.

5.2.2 Graphical Representations

After a breakpoint has been set, it should be recognizable by a change in the graphical representation. The responsible class *BreakpointDecorator* is shown in Listing 5.4. This class implements the *IVNodePostprocessor* interface to manipulate an existing node. The decorate method checks whether the element is a breakpoint and sets the model element's CSS class *breakpoint*. The modified *VNode* will be returned, the model rendered, and the graphical change is made visible to the user.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
1   @injectable()
2   export class BreakpointDecorator implements IVNodePostprocessor {
3
4     decorate(vnode: VNode, element: SModelElement): VNode {
5       if (hasBreakpointFeature(element) && element.breakpoint) {
6         setClass(vnode, 'breakpoint', true);
7       }
8       return vnode;
9     }
10
11    postUpdate(): void {}
12  }
```

Listing 5.4: The *BreakpointDecorator* class to modify the CSS style of a model element annotated with a breakpoint.

Furthermore, we implemented the *StackFrameDecorator* to highlight a model element as the current stack frame in the graphical representation. With the *StackFrameDecorater*, the class *current* is set to true instead of the class *breakpoint*, as in the *BreakpointDecorator*.

5.2.3 GLSP Debug Module

Listing 5.5 shows the *glspDebugModule* to which the commands and the decorators for the graphical modification of the model are added. For this purpose, a new dependency injection container is created which can be injected into the language-specific diagram editor.

```
1   const glspDebugModule = new ContainerModule((bind, _unbind, isBound) => {
2     configureCommand({ bind, isBound }, AnnotateStackCommand);
3     configureCommand({ bind, isBound }, ClearStackAnnotationCommand);
4     configureCommand({ bind, isBound }, AddBreakpointCommand);
5     configureCommand({ bind, isBound }, RemoveBreakpointCommand);
6     configureCommand({ bind, isBound }, EnableBreakpointCommand);
7     configureCommand({ bind, isBound }, DisableBreakpointCommand);
8     bind(TYPES.IVNodePostprocessor).to(StackFrameDecorator).inSingletonScope();
9     bind(TYPES.IVNodePostprocessor).to(BreakpointDecorator).inSingletonScope();
10  });
11
12  export default glspDebugModule;
```

Listing 5.5: The *glspDebugModule*, to integrate the GLSP actions into the client infrastructure.

5.2.4 Debug Adapter Contribution

The Theia-debug extension provides contribution points to add a new debug adapter to the web-based IDE easily. To connect the WFML debug adapter to the Theia-debug extension the class *AbstractVSCodeDebugAdapterContribution* is used that inherits from *DebugAdapterContribution*. In Section 5.1, we have already discussed contribution points within the Eclipse Theia framework. The *DebugAdapterContribution* serves as a contribution point to connect the WFML debug adapter extension with the Theia-debug extension. The debug-adapter is integrated into the back-end process of the Theia-debug extension and communicates through WebSockets with the front-end.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

For the implementation of the debug adapter, we used the existing npm-modules vscode-debugadapter[10] and vscode-debugprotocol[11], which simplifies the development of new debug adapters. In order to contribute to the debug-related contribution points, a debug adapter extension requires a package.json, which contains the contributions specific to debug extensions. Like any other extension, the package.json declares the fundamental properties like name and version of the extension, as can be seen in Listing 5.6. In the debuggers section of the contributes section the debugger for the WFML is defined with the attribute *type* as "workflow-debug". The user can take the debug type as a reference in launch configurations. The optional attribute *label* can be used to give the debug type a proper name when showing it in the user interface. The *program* attribute defines the relative path to the debug adapter that implements the debug protocol. If the path to the debug adapter is not executable, the optional attribute *runtime* provides an appropriate runtime [57].

The *configurationAttributes* attribute section contains the schema for launch configuration arguments specific to the debugger. This schema is used to validate the launch.json and support IntelliSense and tool tip when editing the launch configuration. The user is instructed to specify the absolute path to the diagram to be debugged. One can specify whether the debugger should stop the session before executing the first model element [57].

```
1  {
2    "name": "theia-workflow-debug",
3    "version": "0.1.0",
4    ...
5    "contributes": {
6      "debuggers": [{
7        "type": "workflow-debug",
8        "label": "Theia Worflow Debug",
9        "program": "./out/workflow-debug.js",
10       "runtime": "node"
11       "configurationAttributes": {
12         "launch": {
13           "required": [
14             "program"
15           ],
16           "properties": {
17             "program": {
18               "type": "string",
19               "description": "Absolute path to a diagram.",
20               "default": "${workspaceFolder}/${file}"
21             },
22             "stopOnEntry": {
23               "type": "boolean",
24               "description": "Automatically stop after launch.",
25               "default": true
26             },
27           },
28         },
29       },
30       ...
31     }]
32   }
33 }
```

Listing 5.6: package.json file of the WFML debug adapter

[10] https://npmjs.com/package/vscode-debugadapter
[11] https://npmjs.com/package/vscode-debugprotocol

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Please note that the detailed specification of the package.json is partly removed due to space limitations. It can be found in the code repository[12].

Figure 5.2 shows the launch configuration for the WFML example introduced in Section 4.1. The launch configuration is stored in the settings.json file. The configurations section shows parts of the attributes mentioned earlier. The requested debug adapter has the type "workflow-debug" and is used to launch the debugging process. The "stopOnEntry" flag is set to true to halt the execution before the first element of the model. The program attribute shows the path to the model to debug, namely "example1.wf".

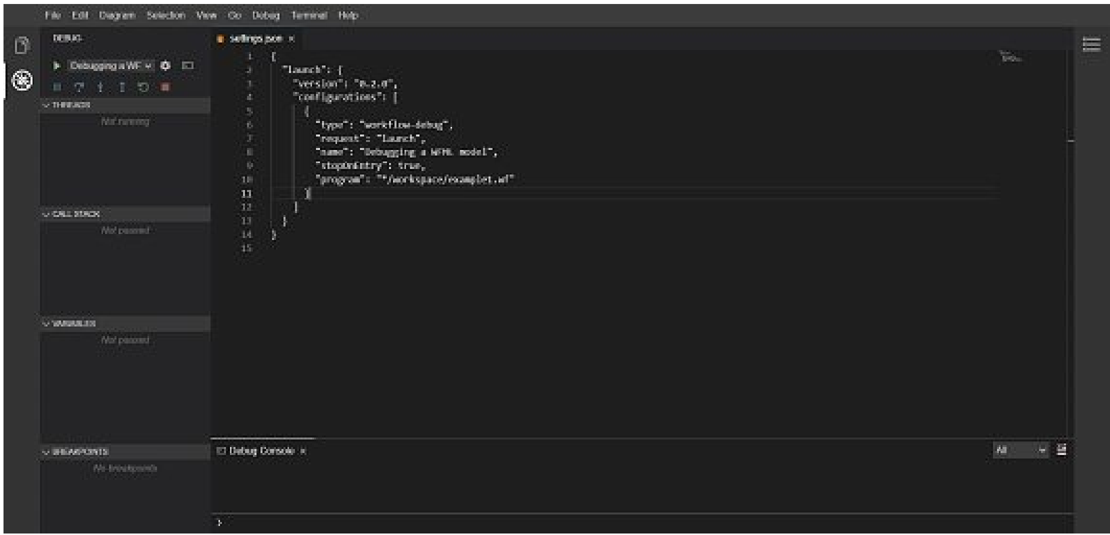

Figure 5.2: The launch configuration of the WFML example.

5.2.5 Communication between the Actors

In Figure 5.3 the communication between the user, development tool, WFML debug adapter, and WFML debugger is shown. Before starting the debugging process the user sets a breakpoint on the model element with the ID "task1". After the user invokes the debugging process the debug adapter gets started and the initialize phase between the development tool and the WFML debug adapter begins. During the initialize phase the capabilities are exchanged. Afterward, the configuration phase starts, in which the development tool sends the *GLSPBreakpoints* to the WFML debug adapter. When the configuration phase has finished the *launch* request starts the debugging of the model. The WFML debug adapter connects to the external WFML debugger and sends the path of the diagram to debug. Furthermore, the WFML debug adapter sends the breakpoint with the ID "task1" and the associated path of the diagram to the WFML debugger.

[12] https://github.com/EderH/workflow-example-debug-adapter

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

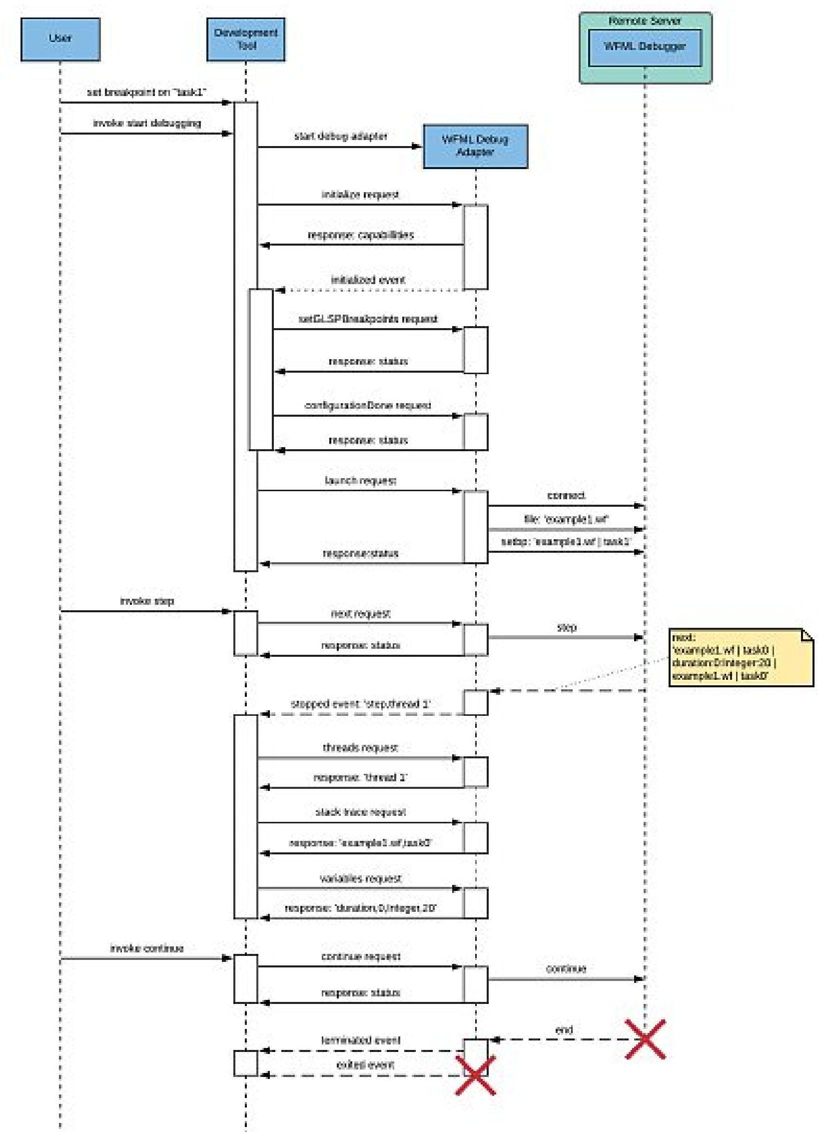

Figure 5.3: A user invokes the DAP-based communication between the development tool, WFML debug adapter, and WFML debugger during the debugging of a WFML diagram.

In a further use case, the user invokes a step on the GUI of the development tool, which in turn sends the *next* request to the WFML debug adapter. The WFML debug adapter informs the WFML debugger about the request by sending a *step* command. The WFML debugger performs a step and responds with the next model element in the diagram. The response includes the diagram path and the ID of the next model element. Furthermore, the variables of the model element are included in the response.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

In this use case, the model element has a local (i.e. "0" for local and "1" for global) variable "duration" that is an Integer data type with the value "20". The last row of the response shows the stack trace. The stack trace in the example consists of one stack frame, namely "task0". When the WFML debug adapter receives the response a *stopped* event gets emitted for "thread 1" and the reason "step". As a result, the development tool requests all available information for the mentioned thread. This includes information about the stack trace and the variables. The development tool then displays the received information in the appropriate windows of the GUI.

In the last use case, the user invokes a *continue* request sent from the development tool to the WFML debug adapter. The WFML debug adapter forwards the request to the WFML debugger by sending the *continue* command. The WFML debugger continues the execution till the end of the model and sends the *end* command back to the WFML debug adapter. The WFML debug adapter emits a *terminated* event to inform the development tool that the debugging process has finished. Finally, the WFML debug adapter emits an *exited* event and exits from the running development tool.

5.3 Challenges

In this section, we discuss some of the challenges of implementing the WFML debugger. The challenges were identified while working with the different frameworks for textual and graphical languages used in the practical part. The concepts we have developed to solve these issues apply not only to the WFML but also to the debugging framework for graphical modeling languages in general.

Text Editor vs. Diagram Editor

The combination of the DAP developed for textual languages with the GLSP developed for graphical modeling languages presented us with various challenges during development. To be able to write or view source code at all, each IDE requires a so-called text editor. The editor provides one with various actions to edit the source code. The Theia-debug-extension, used for the implementation of the framework was developed for use in connection with textual languages. As a result, a text editor is used to open and edit source code files within this component. The text editor used is the so-called Monaco Editor[13], which was created by Microsoft in the course of the development of VSCode and which is not suitable for editing graphical modeling languages.

In contrast, the GLS platform provides an extendable diagram editor for creating, displaying, and editing models, which must be extended for a specific modeling language. In the course of the development of the WFML debugger, we had to think of a concept that would allow us to integrate the Diagram Editor of the GLS platform into the debug session component of the Theia-debug extension and thus replace the Monaco Editor.

[13] https://microsoft.github.io/monaco-editor

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

In the following, we present one solution: In the first step, we found out that the Monaco Editor is firmly anchored in the default implementation of the debug session component, which is why this component became unusable for us. If the debug session's default behavior is not sufficient, the Theia-debug extension provides a contribution point to integrate a custom implementation of the debug session. This contribution point is called *DebugSessionContribution* and consists of the debug type for which the debug session is created and a *DebugSessionFactory*, which creates the specific debug session. Using this contribution point, it was possible in a further step to create a custom *GLSPDebugSession*. Instead of the Monaco Editor we integrated the *GLSPDiagramEditor* via dependency injection into the custom *GLSPDebugSession*. Through this solution, the model to debug will be opened at the start of the debug session via the newly integrated diagram editor and can thus be animated during the debug session.

Consistency between Diagram Editor and Breakpoint View Window

Synchronizing the diagram editor and the breakpoints view was another challenge we faced when managing *GLSPBreakpoints*. On the one hand, it should be possible to add and remove breakpoints via the diagram editor while ensuring the correct listing of breakpoints within the breakpoints view. On the other hand, the commands to enable, disable, and remove breakpoints are available to the user via the breakpoints view. Due to the graphical representation of the breakpoint in the form of the red-colored frame, it was necessary to inform the diagram editor about the commands executed via the breakpoint view. This required a two-way communication between the components of the Theia-debug extension and the client framework of the GLS platform.

In the Monaco editor for textual languages, the editor responds to an editor mouse event and then transmits the position of the breakpoint to the BreakpointManager of the Theia-debug extension. The interface IEditorMouseEvent[14] is used to listen to the mouse event. Whenever a breakpoint is removed from the breakpoint view, the Theia-debug extension calls the editor to remove the breakpoint decoration. To change the decoration the interface IModelDeltaDecoration[15] is used. We were not able to reuse any of these components required for the communication between the Monaco Editor and the Theia-debug extension, since we use the diagram editor provided by the GLS platform. In the following, we present the solution for adding interaction behavior between the diagram editor and the breakpoints view.

The first direction - from the diagram editor to the breakpoint view component - can be managed using the *GLSPTheiaSprottyConnector*. The *GLSPTheiaSprottyConnector* bridges the gap between GLSP dependency injection containers and a specific connection client from the Theia dependency injection container. Thus, the *GLSPBreakpointManager* can be injected into the *GLSPDiagramManager* to extend the default *BreakpointManager*

[14] https://microsoft.github.io/monaco-editor/api/interfaces/monaco.editor.ieditormouseevent.html

[15] https://microsoft.github.io/monaco-editor/api/interfaces/monaco.editor.imodeldeltadecoration.html

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

implementation of the Theia-debug extension. The *GLSPDiagramManager* is responsible for controlling and managing the diagram editor. Via the connector, the methods of the *GLSPBreakpointManager* can be accessed within the client framework container of the GLS platform, and thus, breakpoints can be added or removed in the diagram. The *GLSPBreakpointManager* is then responsible for listing the available breakpoints in the breakpoints view.

The second direction - from the breakpoints view component to the diagram editor - can be simply accomplished using the new breakpoint actions described in Section 4.2. In order to use these actions within the breakpoint components of the developed framework, access to the *action dispatcher* of the GLS platform is required. We were able to solve this issue by filtering all existing diagram widgets from the application shell containing an *action dispatcher*. Finally, the breakpoint actions can be triggered via the *action dispatcher* of the individual diagram widgets. The actions, in turn, modify the model and thus change the graphical representation of the breakpoint.

Model Interpretation

In Section 2.1, we presented the model interpretation approach used in the course of model-driven development. This approach aims to interpret the individual model elements and to execute the task or process that the model element represents. A DSML consists of different abstract syntax elements that have different execution behavior. Therefore, we needed to develop a concept that would allow the debugger to execute these different types of model elements. Besides, we had to make sure that the behavior of the model elements could change in the future, or additional elements are added to the DSML. In the following, we would like to present the solution to the issue described above.

In the course of the literature research concerning interpreters, we came across the visitor pattern [58], which belongs to the behavioral design patterns [59, 60]. The visitor design pattern shows how to separate the structure of an object hierarchy from the behavior of traversals over that hierarchy. Figure 5.4 shows the conceptional architecture of the visitor pattern. The visitor pattern consists of the following concepts.

- **Visitor.** The *visitor* interface or abstract class defines a set of visiting methods in which concrete elements of an object structure are passed as arguments. If the language in which the program was written supports method overloading, these visiting methods have the same name, but the type of parameter that is passed differs.
- **Concrete Visitor.** Every concrete visitor must implement all visit methods declared in the abstract visitor. Each visitor is responsible for different operations.
- **Element.** The interface *element* declares a method to accept a visitor. This method should have only one parameter, which has the *visitor* interface as the type.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

- **Concrete Element.** The concrete element must implement the *accept* method. The task of this method is to redirect the call to the visitor's method corresponding to the current element class.

- **Client.** The client is the consumer of the classes of the visitor pattern. It usually represents a collection or other complex objects. The client can access elements and instruct them to accept a visitor to execute the associated operation.

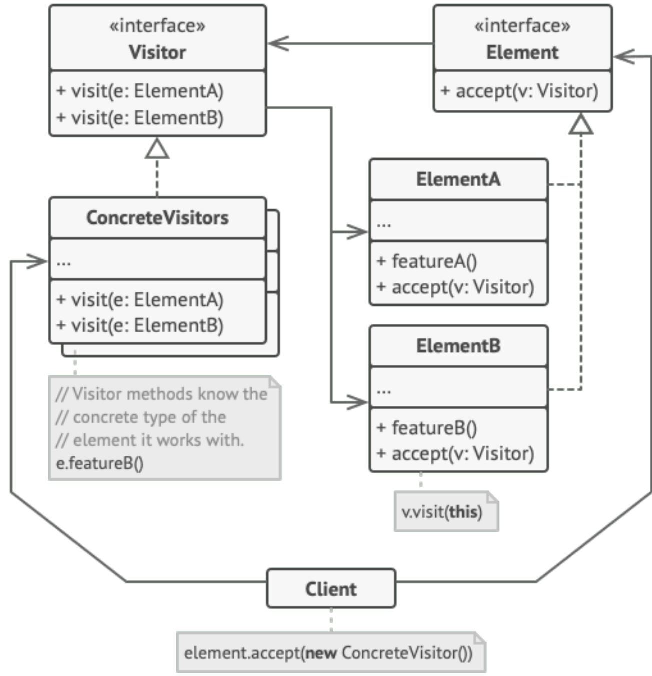

Figure 5.4: Architecture of the visitor pattern [61].

The visitor pattern applied to the running example of this thesis is shown in Figure 5.5. The client is the specific WFML debugger that we implemented. The WFML debugger has a parser that reads the model from a JSON object and stores it in corresponding data structures (e.g. *TaskNode*, *ActivityNode*, *Edge*). Subsequently, the model elements are executed based on their connections to each other. The current model element to be executed by the debugger accepts the *WorkflowInterpreter*. The *WorklfowInterpreter* represents a concrete visitor and implements the visitor interface. The visitor interface declares its own visit methods for task nodes, activity nodes, and edges. In the *WorkflowInterpreter* class, these visit methods are overwritten, and the concrete operations are implemented. The classes used to parse and store the model are extended by the accept method.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

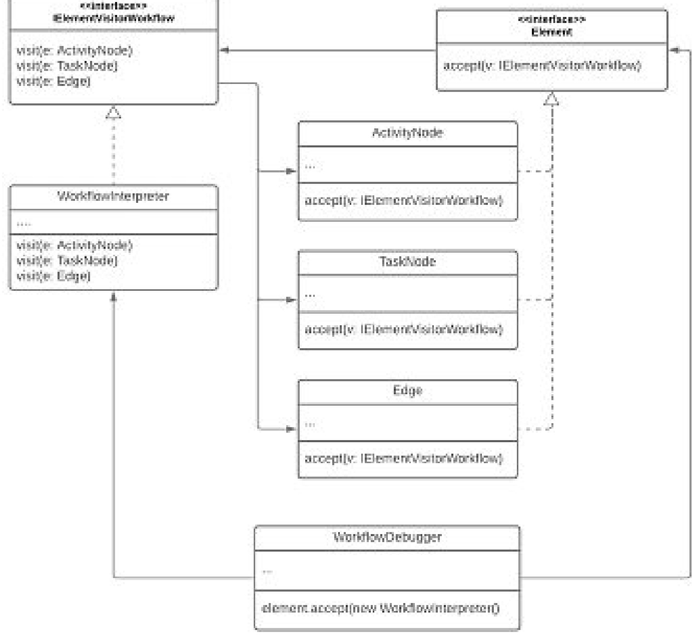

Figure 5.5: The visitor pattern applied to the running WFML example.

This solution for the model interpretation approach allows easy adaption to the behavior of individual model elements in the future by rewriting the *visit* method of the element. Besides, adding new types of model elements to the DSML and its debugger can be done without much effort. The only extensions necessary are creating new concrete elements and adding the corresponding *visit* methods to the *WorkflowInterpreter*.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 6

Evaluation

The debugger for graphical modeling languages described in chapters 4 and 5 was developed based on the existing WFML. This chapter introduces the second DSML used, from which, in addition to the existing requirements, we derive new ones and evaluate the implemented framework to see whether it can be applied to this newly defined language. With this, we want to ensure the reusability of the framework for other languages and domains.

6.1 State Machine Modeling Language

The State Machine Modeling Language (SMML) is an executable/interpretable language based on the concepts of finite-state machines. A state machine is a finite machine when it can be assumed that the number of states is finite. The SMML consists of the following language constructs: states, state transitions, and actions.

A state is a description of the system's status that is waiting to execute a transition. In the context of the SMML, we defined three different kinds of states. *Simple State* models a situation during which some invariant condition holds. The notation of a simple state is represented as a rectangle with rounded corners and the state's name inside the rectangle. The other two kinds of states are so-called pseudostates. The *Initial Pseudostate* is the starting point of a state machine and the source of a single transition to a simple state. There can be at most one initial state within a diagram. An initial state is represented as a small solid filled circle. The *Final Pseudostate* is a state in a sub-state machine that signifies that the execution of the sub-state machine has been completed. A sub-state machine is a decomposition mechanism that allows factoring of common behaviors and their reuse. A state of a sub-state machine inherits all the behavior actions and transitions of the composite state. A final state is represented as a circle surrounding a small solid filled circle.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Bibliothek, Your knowledge hub

A state transition is a transition between two states. This transition is executed when the defined logical conditions, events, or inputs are fulfilled.

Actions determine the output of a finite-state machine, which takes place in a certain situation. The SMML only supports actions that are triggered through an transition event and are performed during a state transition.

Similar to the WFML, we use the EMF to define the abstract syntax of the SMML. The generated Java classes are integrated into the server framework of the GLS platform. Figure 6.1 shows the metamodel of the SMML. As with the WFML, the SMML metamodel extends the GLSP metamodel[1] integrated into the GLS platform. In the SMML, the *GEdge* class is extended by the *Transition* class having the properties *event* and *action*. The *State* class extends the *GNode* class and has the properties *name* and *kind*. The *kind* property specifies the type of state. The SMML distinguishes between *initialState*, *simpleState*, and *finalState*.

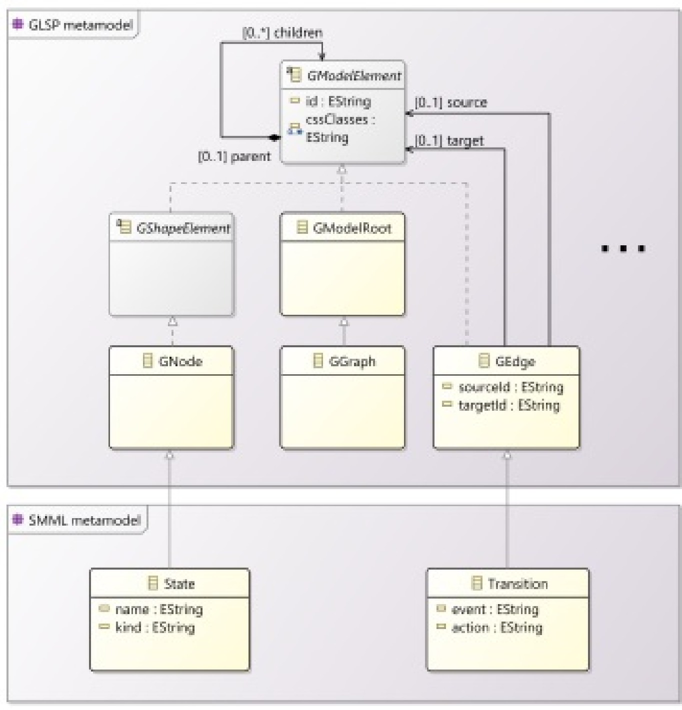

Figure 6.1: The SMML metamodel extending the GLSP metamodel.

Furthermore, we use the client framework of the GLS platform and the underlying diagram framework Sprotty to define the concrete syntax of the SMML. The GLSP diagram editor will be extended to fit the requirements for the SMML. Figure 6.2 shows

[1] https://github.com/eclipse-glsp/glsp-server/blob/master/plugins/org.eclipse.glsp.graph/model/glsp-graph.ecore

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

the result of integrating the newly defined SMML into the GLS platform. The tool palette in the diagram editor consists of three nodes: simple state, initial state, and final state. A transition can be used to connect the states. The shown example model constructed with the SMML illustrates a Bank Automated Teller Machine (ATM) top-level machine. Based on the transition conditions, the state machine changes between states. The Bank ATM is initially turned off. After the machine is turned on, the ATM performs a *startup* action and enters the *Self Test* state. If the test fails, the ATM goes into *OutOfService* state, otherwise, there is a triggerless transition to the *Idle* state. In this state, the ATM waits for customer interaction. The ATM state changes from *Idle* state to *ServingCustomer* state when the customer inserts a banking card in the ATM card reader. The transition from *ServingCustomer* state back to the *Idle* state could be triggered by a *cancel* event as the customer could cancel the transaction at any time. The same applies to the transition from *ServingCustomer* state to the *OutOfService* state that is triggered by the *failure* event. When the ATM requires maintenance the transition from the *Idle* state to the *Maintenance* state could be triggered by the *service* event. If the maintenance fails the ATM state changes in the *OutOfService* state, otherwise another triggerless transition leads the ATM into the *Self Test* state. Finally, the *turn off* event can be triggered in the *Idle* state and the *OutOfService* state to shut down the ATM and change to the *Off* state. The *ServingCustomer* state is a composite state with the sub-state machine shown in Figure 6.3. The sub-state machine consists of the sequential states *CustomerAuthentication*, *SelectingTransaction*, and *Transaction*. Those states have a triggerless transition to change into the next state. After the transaction is finished another triggerless transition changes the ATM state back to the *Idle* state.

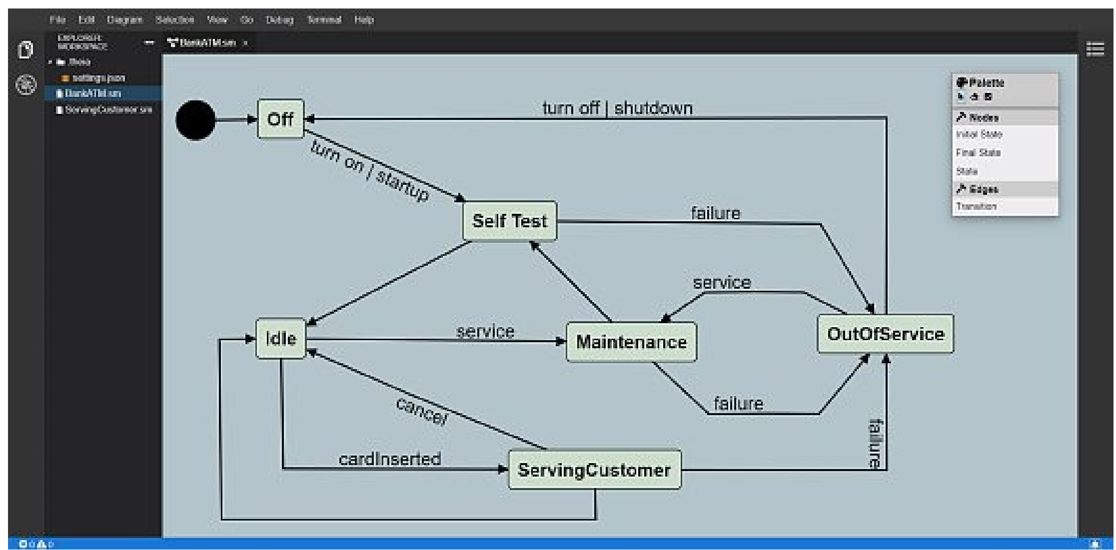

Figure 6.2: The SMML diagram editor integrated into the Eclipse Theia framework.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

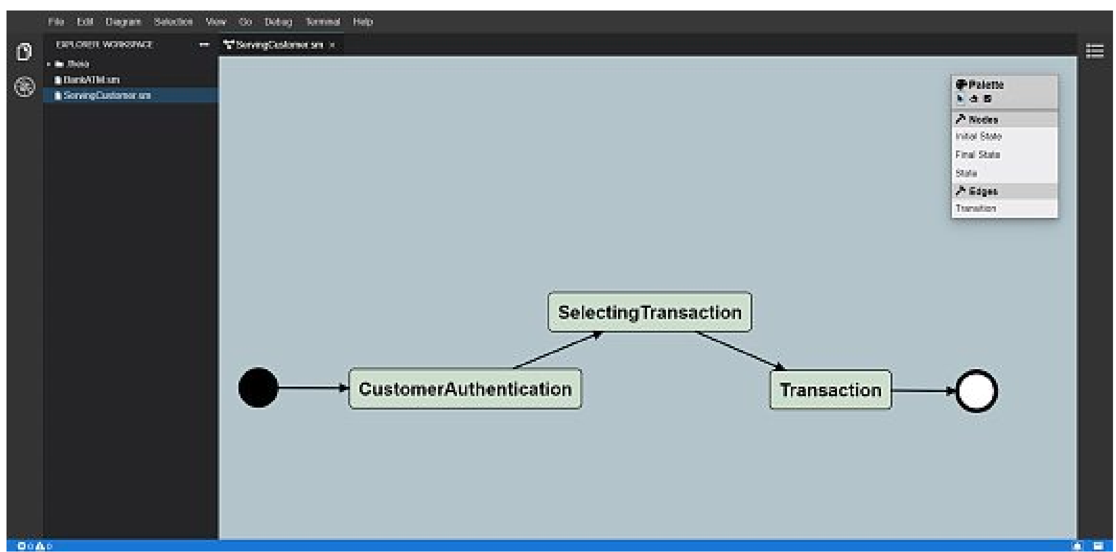

Figure 6.3: The SMML sub-state machine for the *ServingCustomer* state.

6.1.1 Requirements for Debugging the State Machine Language

In Section 4.2, we derived the requirements for a DSML debugger based on the debugging concepts we discussed in Section 2.3 and implemented the WFML debugger. In a further step, we take a look at how the developed framework can be reused from the WFML debugger, to implement another debugger for the SMML.

Before we derive the requirements for debugging the SMML, we would like to discuss the main differences between the WFML and the SMML. The WFML differs from the SMML mainly because the transition from one element to the next is not done through simple edges, but by event-driven transitions. To control these transitions, the debugger GUI needs an input field to interact with the user. The further execution of the SMML debugger then takes place via the event control of the user. Furthermore, the states of the SMML are unique within the diagram, and multiple usages, as known from the task nodes of the WFML, are no longer possible. However, this does not affect the debugger, since we use the ID of the model element and the path to the diagram in which the model element is located for the unique identification. Finally, compared to the WFML, the SMML has a single initial state within a state machine, which is intended as the starting point for the execution.This means that the debugger does not have to look for the model element that does not have an incoming edge, but can use the state of the type "initialState". The final state means an endpoint within a sub-state machine. The debugger no longer has to check whether the state has an outgoing edge, but knows from the state type that it is a "finalState".

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Similar to the WFML debugger, we also want to provide the user of the SMML with a debugger that has the basic features of a modern debugger. Therefore, we discuss the defined requirements introduced for the WFML debugger in Chapter 4 and consider how the SMML affects these requirements. In the following, we analyze the existing components of the WFML debugger, whether they can be reused and how the new requirements can be implemented.

Execution Modes

The user needs functionalities to start, pause, and stop the execution of the model. Therefore, the user requires a graphical user interface that can be used to operate the individual debugging features and to output relevant information to the user. By defining another language, we do not expect any modifications of the generic graphical user interface provided by the Theia-debug extension. However, a separate debug adapter for the SMML is required, but the existing logic of the WFML debug adapter can be reused. The SMML debugger must be able to parse, execute, and interpret the new language constructs accordingly. In the following, we analyze how the developed components for the execution modes in the WFML debugger can be applied to the SMML.

> **Applying the WFML debugger to the SMML:** Like the WFML debugger, the debug session is started in the development tool, which in turn, sends the file path of the model source via the debug adapter to the SMML debugger. The only difference between the WFML and the SMML is the different file extension of the model source. Since the protocol transfers the path to the file in a String data format, the only thing to deal with is the changed file format at the debugger-side. To pause or stop the debugger, the protocol, and the components of the Theia-debug extension, use the thread ID of the debug session. Thus, an entirely language-independent concept for starting a debug session is available.

Steps

The user should be able to step through the model and execute one model element at a time. The WFML uses simple connections between model elements whereas a state in the SMML has several transitions that are triggered through transition events. During the debugging process, the user wants to trigger these events by manually selecting them to determine the further course of the execution. Thus, the user requires a dialog input field that lists the events available for the current state and further allows the user to select one of the listed events. During the transition from one state to another state, actions might be executed. Hierarchy structures are also supported by the SMML, which must be handled by the SMML debugger. Finally, the user wants to be informed about the course of all transitions that were already performed. For this purpose, the debug window should be extended by another view where the current event flow (e.g. *turn on* event triggering the transition from the *Idle* state to the *Self Test* state) is displayed. In the following, we discuss the steps necessary for implementing the requirements mentioned above.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

1. **Language-specific debug request data:** The DAP does not provide any existing requests that can be used to transfer the available state events between the debug adapter and the development tool. However, as with the *GLSPBreakpoints*, new custom requests and events between the Theia-debug extension and the debug adapter can be introduced. It is necessary to introduce a new protocol event that transmits all transitions available for the current state to the development tool. Before this, the model execution must be interrupted using a stop event. After receiving the information, the development tool is in charge of opening the input field, presenting the received transition events, and waiting for the user to select one of them. As soon as the user confirms the input, a request which sets the event should be sent from the development tool to the debug adapter and further forward the selected event to the SMML debugger. Using the obtained information, the SMML debugger should continue the execution and send back the new event flow to the development tool's event window.

2. **Integration of an event window into the Theia-debug extension:** For the implementation of the transition events, new components must be integrated into the existing debugging framework. A component is required, where information about the events possible for the next transition received from the SMML debugger can be stored, and that can be used to list the events in the input dialog window. For this, a class *GLSPEvent* should be implemented, which consists of attributes to store a unique ID, the model element for which the event is available, the event name, and the path to the diagram source. Furthermore, the Theia-debug extension must be extended to integrate the event window. Using dependency injection, additional views can be injected into the existing debug view window. Thus, the event flow of the current debug session received from the SMML debugger can be listed.

Breakpoints

Like textual languages, where breakpoints can be set in the source code, the user should also be able to set a breakpoint on a specific element in the SMML diagram editor. We expect no differences concerning breakpoints between the two languages WFML and SMML since breakpoints are stored language-independent based on the ID of the model elements. In the following, we analyze how the developed components for breakpoints in the WFML debugger can be applied to the SMML.

Applying the WFML debugger to the SMML: The *GLSPBreakpoints* were defined independently of the language in use since only the ID of the model element is stored. The components for managing, displaying, and transmitting the breakpoints to the debug adapter were also implemented independently of the language in use. The request for transferring the *GLSPBreakpoints* from the development tool to the debug adapter only stores the model element's ID and the path to the diagram file. The actions and commands defined within the client

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

framework of the GLS platform execute the operations based on the ID of the model element. The graphical animation of the breakpoint within the diagram is based on the implemented breakpoint feature. Any model element in a new language can support this feature. To describe the style of the model elements and the representation of the breakpoint we use CSS. In case the graphical representations of a new language do not have frames, breakpoints can still be easily integrated. Therefore, it makes no difference how the syntax of the element was defined.

Stack Trace

The user wants to view the procedure calls that are currently on the stack. Therefore, a call stack window is required that shows the order in which the nodes (e.g. states, transitions) of the SMML are getting called. Both WFML and SMML support hierarchy structures that can be displayed via the call stack view. In the following, we analyze how the components for stack traces in the WFML debugger can be applied to the SMML.

Applying the WFML debugger to the SMML: The representation of the current stack frame in the call stack view also takes place via a String data type. The SMML debugger determines the current stack frame and transmits the ID of the model element as a String data type to the debug adapter. The debug adapter forwards the received information via the DAP to the development tool. Thus, using a String data type to display and transfer the stack frame we are independent of the language in use.

Runtime Variable I/O

When the model's execution is paused, the user requires a view, where the variable values of the current stack frame are shown. The model elements of the SMML have different variables than the model elements of the WFML. Thus, a transition has the two variables event and action, which must be represented accordingly. In the following, we analyze how the developed components for variables in the WFML debugger can be applied to the SMML.

Applying the WFML debugger to the SMML: The components for variables window were developed independently of the language used and can be reused for the SMML debugger. The variable attributes (e. g. name, value, type) are transferred between the debug adapter and the development tool as a String data type and are therefore kept very generic. This makes the exchange independent of the language used. The debugger is responsible for storing the variables of the elements in the corresponding data types during parsing. Whenever a variable request is executed, the debugger converts the attributes for the respective variable into Strings and transmits all existing variables to the debug adapter. The debug adapter then redirects the variables to the development tool.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Exceptions

The user wants to be informed if an exception occurs during the execution of the model and to what cause this is due. These must be presented to the user, who usually only has knowledge of the modeling language in an understandable form. In the following, we analyze how the developed components for exceptions in the WFML debugger can be applied to the SMML.

> **Applying the WFML debugger to the SMML:** The implementation of the Theia-debug extension as well as the WFML debug adapter implementation can be reused for the SMML. Exceptions that occur during execution must be adapted to the SMML so that users who are only familiar with this modeling language understand the information received from the debugger. Therefore, new exception handlers must be implemented at the SMML debugger to handle errors specific to the SMML language.

Requirements Overview

Table 6.1 gives an overview of the requirements for debugging the SMML. The cells colored in green represent the existing components we can reuse, whereas the cells colored in red represent the components we have to implement/adapt. The shortcut "NA" stands for "No Action", i.e., no implementation or adaption is required. In the first column, the requirements mentioned above are listed. The second column shows which components we can reuse from the Theia-debug extension. In this table, we already expected to reuse the components developed for the WFML, and therefore some requirements are highlighted in green. The third column shows the requests and events that can be used from the DAP. The fourth column represents the modification using the GLSP to integrate graphical animations. The fifth column shows the implementations necessary for the SMML debug adapter. We expect to reuse the components developed for the WFML debugger. The last column lists all requirements for the debugger implementation. Since the language constructs of the SMML are different from the WFML, this part requires significant adaption and is the most affordable.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Requirement	Theia-debug extension (GUI)	DAP	GLSP	Debug Adapter (WFML debugger)	Debugger
Start	command view	launch request	open diagram	connect to debugger, send model	parse, execute, and interpret model
Pause	command view	pause request	NA	forward pause request	pause execution
Stop	command view	disconnect request	NA	forward stop request	stop execution
Step	command view	next request	NA	forward step request	get next element, send element
Step In	command view	stepIn request	open child diagram	forward stepIn request	step into child model
Step Out	command view	stepOut request	open parent diagram	forward stepOut request	step out of child model
Breakpoints	Extending the breakpoint view for GLSPBreakpoints	setGLSPBreakpoints request	add/remove & enable/disable breakpoints	send GLSP-Breakpoints	stop execution on breakpoint
Events	Adding event view for GLSPEvents, dialog input field	setGLSPEvent request, eventFlow request, GLSPEvents event	NA	send GLSPEvent, response GLSPEvents, response event flow	stop execution on event
Stack Trace	call stack view	stackTrace request	highlight current stack frame	response stack trace	send stack trace
Variable	variable view	variables request	NA	response variables	send variables
Exceptions	debug console	output event	NA	response exception	handle exception

Table 6.1: Requirements for Debugging the SMML.

6.1.2 Implementation

We implemented the SMML debugger based on the requirements described previously. In the following, we shortly describe the SMML debugger features by showing illustrations of the debugging process within the web-based development environment.

The example diagram in Figure 6.4 shows the BankATM model introduced in the previous section. The debug session is already started and paused on the *Self Test* state marked through the green frame of the state's representation. On the left side of the development tool, the debug view is displayed, where the views and commands known from the WFML debugger are reused. Again, the thread view shows the current status of the execution. The call stack view displays the current stack frame, and the variables view shows the state variables and the corresponding values. Additionally, the debug view shows the newly developed events view on the bottom, where the steps taken so far are represented. From the initial state, the debugger went on through the *default* event to the next state *Off*. Afterwards, the *turn on* event triggered the transition to the current state *Self Test*.

From this point, the user can click on the step command, and a new dialog window appears. The opened dialog window shown in Figure 6.5, asks the user to select one of the listed events to trigger the next transition. As long as the window is opened, the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

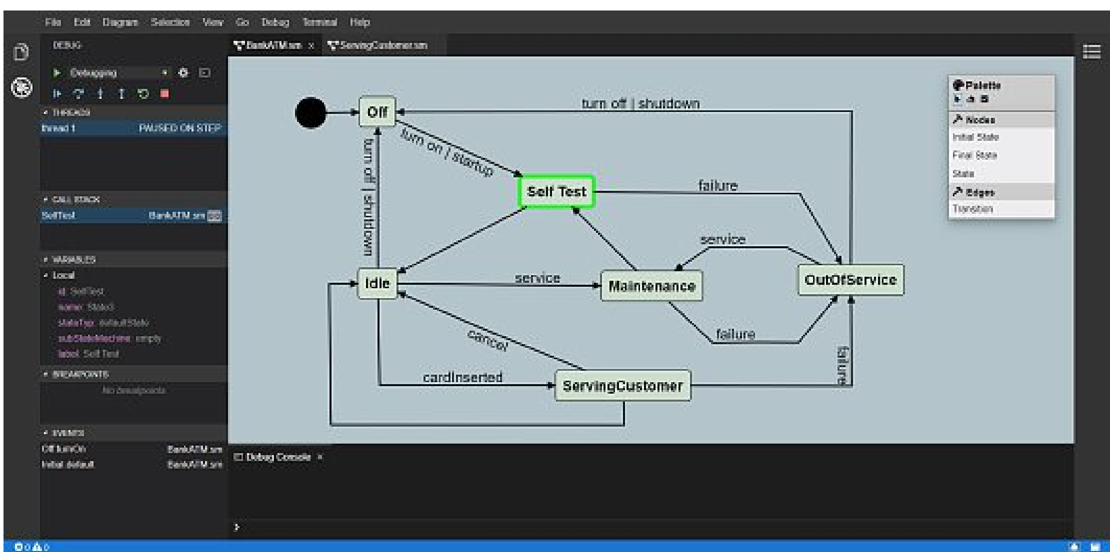

Figure 6.4: The SMML debugger is paused at the state *Self Test* and presents the current stack frame information.

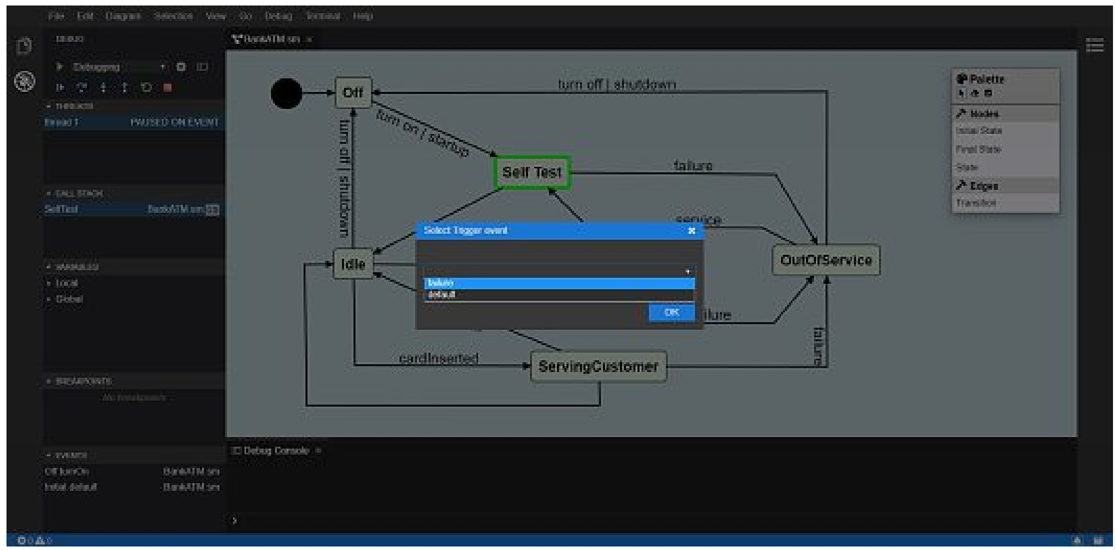

Figure 6.5: The SMML debugger is paused at the *Self Test* state, waiting for the user to select a trigger event for the next transition.

execution is paused as represented through the threads view. When the user confirms the selected event, the execution is continued, and the SMML debugger moves on to the next state. In Figure 6.6, the thread is paused at the state *Idle*, and within the events view, the last selected event *default*.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

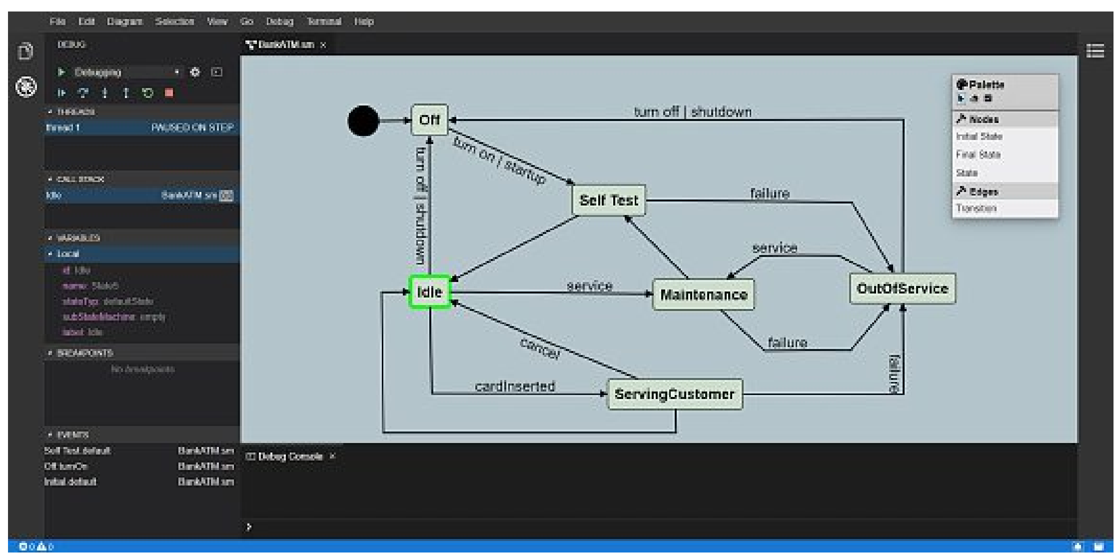

Figure 6.6: The SMML debugger is paused at the *Idle* state after executing the default transition from the state *SelfTest*.

Like the WFML, the SMML also supports hierarchical structures where the user can step into sub-state machines. To demonstrate this use case, the *ServingCustomer* state represents a composite state. When the debugging process is paused at the *ServingCustomer* state, the user can use the step-in command to open the corresponding sub-state machine. The sub-state machine for the *ServingCustomer* state consists of one initial state, one final state, and several simple states in-between. In Figure 6.7, the SMML debugger opened the diagram for the *ServingCustomer* state machine, and the execution is paused at the *CustomerAuthentication* state. In the debug view, the changed call stack is shown, where the *CustomerAuthentication* lies on top of the *ServingCustomer* frame. The user can jump between the diagrams by clicking on the diagram source within the call stack view. In the events view, the steps taken within the sub-state machine, including the new source, are illustrated.

An important point to consider with hierarchical structures in the context of the SMML is the inheritance of the transitions from the composite state to the sub-state machine. The inheritance in case of the *ServingCustomer* composite state is shown in Figure 6.8. In addition to the *default* event to go from the *CustomerAuthentication* state to the *SelectingTransition* state within the sub-state machine, both the *cancel* event and the *failure* event of the *ServingCustomer* state are listed in the input dialog window. This inheritance affects all state transitions of the sub-state machine.

To get back to the parent state machine, the user can either use the step-out command or step to the end of the sub-state machine until the final state has been reached. Using another step command, the user gets back into the parent state machine.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

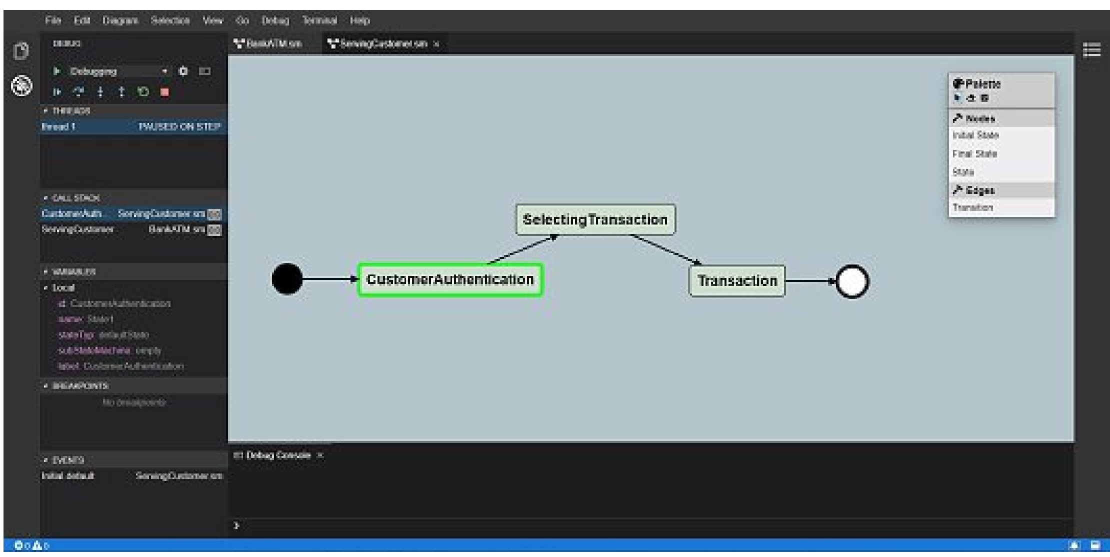

Figure 6.7: The step in command forces the SMML debugger to step into the sub-state machine.

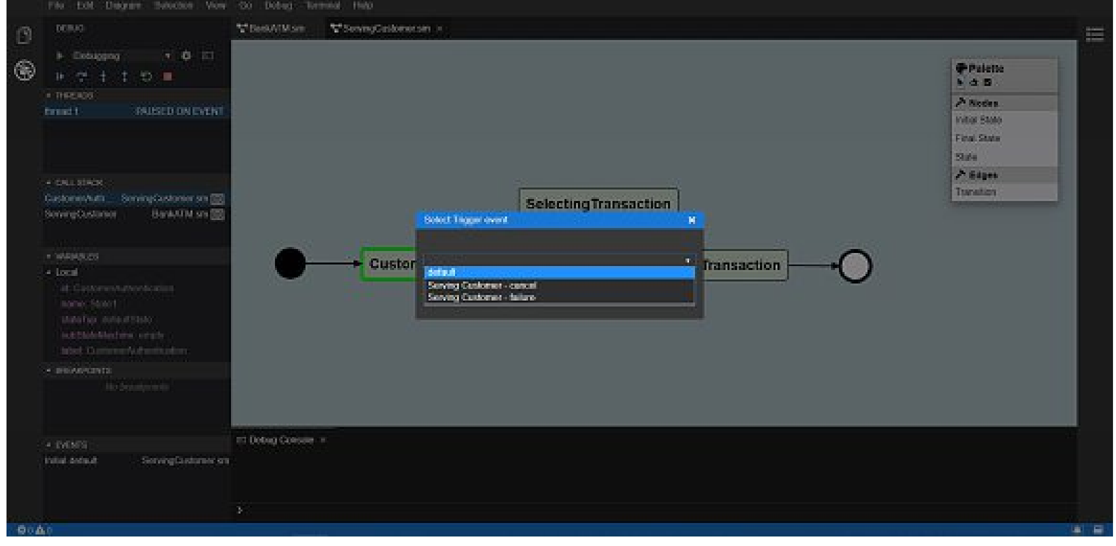

Figure 6.8: An SMML sub-state machine inherits the transitions of the calling composite state.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

As described in the previous section's requirements, *GLSPBreakpoints* can also be set using the SMML debugger. Figure 6.9 shows three breakpoints within the BankATM diagram. Again, in the breakpoints view on the left, all breakpoints within the project are listed. The known features from the WFML debugger to enable, disable, and remove breakpoints are also available with the SMML debugger.

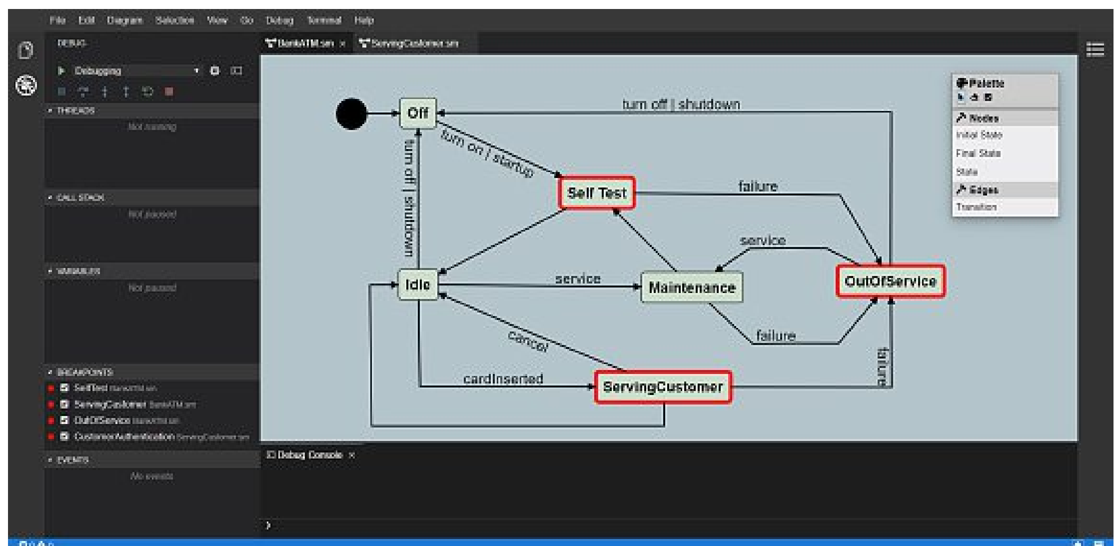

Figure 6.9: SMML states are marked as breakpoints and listed in the breakpoint window.

Finally, Figure 6.10 shows a use case where an exception is thrown during the runtime. In this case, the target ID of the transition triggered through the *turn off* event was changed. This could be caused by a model transformation error or a parsing error. When the SMML debugger tries to execute the transition, the target element with the ID *ThrowException* cannot be found. The corresponding exception is displayed in the debug console in a form the modeler can understand. In the debug view, the thread is paused due to an exception, and all necessary information like the current stack frame, the variables, and the event flow is shown. The implementation of the SMML debug-adapter[2] and the SMML debugger[3] is available on GitHub.

[2]https://github.com/EderH/graphicalLSP
[3]https://github.com/EderH/gml-interpreter

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

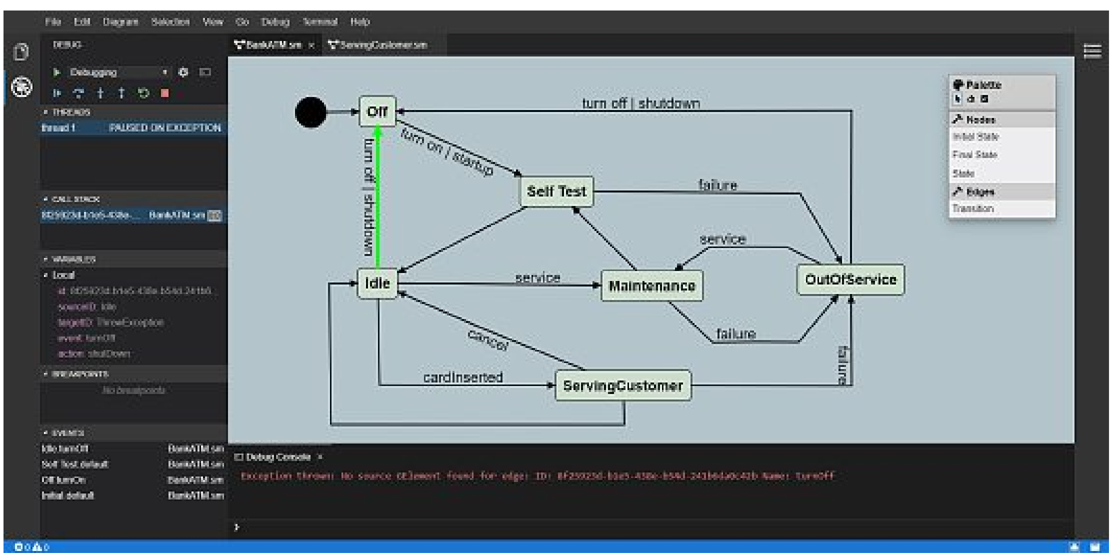

Figure 6.10: The SMML debugger is paused due to an exception and presents the information about the current stack frame and the thrown exception.

6.2 Open Problems

This section discusses the problems we faced during this thesis and that have not been solved. First, the extension of the DAP regarding the newly defined protocol messages and second the communication between the debug adapter and the debugger.

6.2.1 Extension of the DAP

As a result of the implementation of the *GLSPBreakpoints* and the *GLSPEvents*, we defined custom requests to send the breakpoints and events from the development tool to the debug adapter. However, the data transmission does not take place based on a defined standard, as known from other DAP messages. In this work, however, we were only able to identify the request for setting *GLSPBreakpoints* as a language-independent request. Both the WFML and the SMML can use this request. The request for *GLSPEvents* is not usable in the context of the WFML but could be reused in other languages because we do not use language-specific attributes. Graphical modeling languages that require some form of user-interaction could use this request and control the further execution. An example of this would be a language for a game where the user interacts with a game character and instructs the character in which direction to move. Nevertheless, most of the DAP protocol messages can be used to implement a modern debugger for graphical modeling languages. From our point of view, it makes no sense at this stage to define a new protocol to standardize the exchange of *GLSPBreakpoints*. However, one might consider whether to make the DAP more generic regarding breakpoints. An interface could be defined which uses optional attributes. The different types of breakpoints then only use the attributes that they need to transmit the breakpoints between the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

development tool and the debug adapter. It should be noted, however, that this solution still provides a well-structured overview of the individual breakpoint types.

Future experiments should implement further graphical modeling languages as well as additional debugging features (e.g. multiple threads, modify variable). If further custom requests had to be defined during the work, a new evaluation of the DAP should take place and, if necessary, it should be discussed whether a separate protocol for debugging graphical modeling languages should be defined.

6.2.2 Communication between Debug Adapter and Debugger Server

In this work, we have implemented a custom debugger for both WFML and SMML. When starting the debug session, the debug adapter connects to the debugger and starts exchanging messages. This exchange between the debug adapter and the DSML debugger does not take place according to a standardized schema. This is because the graphical modeling languages in use are self-defined and therefore do not have a generally known standard as a basis. For the purposes of this work, however, it was not necessary to establish a separate protocol for exchanging messages between the debug adapter and the debugger. The focus of this work was mainly on the communication between the development tool and debug adapter via the DAP and the generic graphical user interfaces of the development tool.

If the graphical modeling languages and the debuggers implemented are made accessible to a broader community, one can consider whether a separate protocol between the debug adapter and the debugger should be defined.

6.3 Interpretation of the Results

In this section the research questions posed in Section 1.3 are answered based on the evaluation results presented in Section 6.1.

RQ1: What should a debug adapter protocol (DAP) look like to meet the requirements of graphical modeling languages?

To answer this research questions, we performed the following analysis steps: we translated the debugging concepts for debugging textual languages into the context of debugging graphical modeling languages. Furthermore, we derived the requirements for debugging the two case study DSMLs and thus gained an understanding of how the DAP should look like to support graphical languages. The main requirements for a debug protocol for graphical modeling languages are the following:

- **Configuration Process.** Before the debug session is started, protocol messages must be exchanged for the launch configuration, which contains debug adapter specific, and hence language-specific, information. The exchanged information is independent of the representation of the language i.e. whether it is a textual language or graphical modeling language.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- **Execution Modes.** The DAP should provide messages to start, pause, and stop the execution of the model. These commands are responsible for controlling the debugging process in the language-specific debugger. After the start command is executed, the source code or model is parsed via the debugger. For the DAP, it is not relevant whether source code or a model is read in. The pausing and stopping of the debugging process are done based on the currently executed thread. The thread has an ID that is transferred between the development tool and the debug adapter to identify the running debugging process.

- **Steps.** Besides, we require a feature to run the model stepwise, which makes an identification of the current execution position necessary. Compared to textual languages, where the line and column identify the position, the current position in graphical modeling languages is identified by the model elements, which are usually identified by IDs. In case the DSML supports hierarchical structures, facilities to step into the child model or step back out into the parent model are required. The commands step in and step out force the debugging process to change to another hierachy level. However, the debug adapter only needs the thread ID of the current debugging process and forwards it to the debugger. The language-specific debugger is responsible for further execution based on the hierarchy structures of the language used, be it a textual language or a graphical modeling language.

- **Breakpoints.** The protocol should have messages to exchange the breakpoints using the model element ID and the path to the model for which the breakpoint was set. The DAP has different types of breakpoints (e. g. *SourceBreakpoint*, *FunctionBreakpoint*, *DataBreakpoint*, *ExceptionBreakpoint*), which have been defined for the requirements of textual languages. In graphical modeling languages, we identify a breakpoint using the ID and diagram location of the model element. None of the existing breakpoint types of the DAP have the necessary attributes to transmit the above information. That's why we defined a new breakpoint type with the *GLSPBreakpoint*.

- **Stack Trace.** While the execution of the model is paused, the stack trace should be transferred over the protocol to the development tool. For the stack trace, the individual stack frames are transmitted to the development tool. However, the stack frames are not independent of whether a text language or a graphical modeling language is used. The reason for this is the use of the line and column to identify a stack frame within the source code. The two attributes are not optional and therefore a value must be assigned. For graphical modeling languages, we only need the corresponding ID to identify a model element. Therefore, we cannot assign specific values to these two attributes. However, if a stack frame has no source (i.e. path to a source file) in text languages, value "0" is assigned to these two attributes. The development tool recognizes this and ignores the line and column of a stack frame.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- **Runtime Variable I/O.** While the execution of the model is paused, the variables should be transferred over the protocol to the development tool. For the transfer of variables we did not identified any difference between textual languages and graphical modeling languages. The information about the variable is converted into a String data type by the debugger. The response of the variables to the development tool is very generic in that the name, the value, and the type of the variable is transmitted as a String data type. The representation in the variable view is also done as a String data type.

- **Exceptions.** If an error occurs during the execution of the model, the DAP requires messages to transfer the program's status and the appropriate error information to the development tool. Since the exception is transferred using the DAP as a String data type and is also represented as a String in the debug console it does not matter if we use a textual language or a graphical modeling language.

RQ2: Can the DAP for textual languages be applied to graphical model debugging, and if not, what needs to be extended or changed?

For the implementation of the WFML and the SMML debugger, we used the standardized DAP features[4] to communicate the debugging data between the development tool and the debug adapter. We were able to meet a large part of the debugger requirements with the DAP as is. However, there were individual situations where an extension of the DAP was necessary. First, we could not use the existing requests to transfer breakpoints between the development tool and the debug adapter due to missing attributes. We had to define a custom request to transfer the *GLSPBreakpoints* to the debug adapter. The two most important attributes of the *GLSPBreakpoints* are the model element ID stored as a String data type and the path to the model. We could not find this composition in any of the existing requests. Second, analyzing the requirements for the SMML, we were not able to transmit the transition events available for a state to the development tool with the existing protocol messages of the DAP. In the context of textual languages, there is no comparable debugging concept that presents the user with a selection of available input events, from which the user must select an event for further execution. Thus, we defined a new DAP event for the transmission of the *GLSPEvents* to the user interface. Third, there was no request available to send the selected event back to the debug adapter. Therefore, we defined another custom request to transfer the selected *GLSPEvent*. In summary, we conclude that the DAP can be used for graphical modeling languages with a few modifications. Nevertheless, we do not consider it useful to introduce a separate protocol for graphical modeling languages for these two requests. When conducting further work on this topic, such as reusing the developed debugging framework for other languages and debugger features, it has to be re-evaluated whether the definition of a new protocol should be forced.

[4] https://microsoft.github.io/debug-adapter-protocol/specification

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

RQ3: What generic client-frameworks and user interface/debug adapter client components exist for such a debug protocol that can be reused across debugging use cases for multiple graphical domain-specific languages?

The debugger's implementation showed that the existing Theia-debug extension, which supports the DAP, could be reused and extended to fit the requirements of debugging graphical modeling languages. Therefore, we were able to reuse the graphical user interface components of the theia-debug extension like the debug view, threads view, call stack view, breakpoints view, and the variables view. Concerning the breakpoints view, we were able to inject the *GLSPBreakpoints* as a further data source. Besides, the Theia-debug extension offers various contribution points to adapt the debugger user interface to the custom requirements. This allowed us to integrate custom components for managing the debug session and breakpoints. Furthermore, we were able to integrate an additional view for displaying the event flows of state machines into the user interface. Besides, the Theia-debug extension allows easy integration of the debug adapters we have implemented for the two languages WFML and SMML. For the implementation of the debug adapter, we were able to use the existing npm-modules vscode-debugprotocol and vscode-debugadapter. The latter served as the basis for the debug adapter implementation and was extended to meet the requirements of graphical modeling languages. On the one hand, the debug adapter had to be able to handle *setGLSPBreakpoints* requests and *setGLSPEvent* requests. On the other hand, a new event was introduced, which transmits the transition events to the development tool. Moreover, the debug adapter had to be extended to connect to the external DSML debugger and thus be able to exchange messages between them. Furthermore, we used the GLS platform to create custom DSMLs and the corresponding diagram editors. We used the client framework of the GLS platform for the graphical manipulation of the model. On the one hand, we were able to define language-independent actions to visualize the current stack frame within the diagram and, on the other hand, to establish further actions that allow the setting of breakpoints within the diagram. Besides, we were able to define actions to highlight the model elements as breakpoints. Finally, we can say that the developed debugger can be reused to integrate multiple graphical modeling language debuggers into the same IDE due to the debugger framework's language independence. With the frameworks used, we were able to implement all of our previously defined requirements. The Theia-debug extension has been developed quite specifically for textual languages. However, we were able to add custom implementations via the contribution points and thus solve this discrepancy.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

CHAPTER 7

Related Work

A lot of related work deals with the debugging of DMSLs that have a textual concrete syntax or with models that are first transformed into GPL code and then debugged using existing GPL debuggers. Furthermore, the GLSP was only recently introduced, and the DAP became more accessible to a broader audience in the course of VSCode in 2015. At the time of writing, we could not find any literature on the exact topic of developing a model debugger using client-server architecture for graphical modeling languages. In the following, we introduce some of the relevant work in the area of model execution and debugging.

7.1 Debugging Domain-Specific Models

Tezel and Kardas presented two debugging approaches [2] for Multi-Agent System (MAS) DSMLs [62,63]. MASs include multiple interacting software agents within an environment, where each agent performs a task. MAS DSMLs support both the static and the dynamic aspects of agent software. Furthermore, they reflect the agents' internal behavior model, interaction with other agents, and the use of other environment entities. The DSMLs provide appropriate IDEs in which both modeling and code generation can be performed. The first approach uses mapping information from the link between the source language (DSML) and the target language (GPL). The designed model is transformed into GPL code through model-to-text transformation. The generated code is checked using a GPL debugger that sends back the debugging results to the DSML debugging perspective.

The second approach allows to interpret the runtime state directly at the model level. This requires additional runtime states and transitions in the metamodel. Furthermore, a model interpreter and debugger must be implemented to interpret and check the model entities. The model is then debugged via a model-to-model transformation in which the transitions serve as steps between the elements. The runtime states are used to indicate whether an element is defined as a breakpoint.

The first approach differs from our work in that the generated model is first transformed into GPL code and then checked for correctness with existing GPL debuggers. The debugger, we developed debugs the generated model already on the model level. However, the work does not provide any clues on how the debugging results are sent back to the IDE. Like our work, the second approach aims to interpret the model elements already at the model level. This makes it possible to implement a generic debugger and thus minimize the effort for integrating other languages into the IDE. As with the first approach, it is unclear how the communication between debugger and IDE should be implemented.

Wu et al. presented a DSL debugger framework [64] for textual languages reusing existing, tried, and familiar debugging facilities (e.g. breakpoints, step, stack trace). The approach defined by Wu et al. enables the hidden and automatic construction of a detailed mapping between model entities and synthesized code. The developed debugger framework uses Eclipse's built-in debugging facilities. Eclipse serves as a tool integration platform that provides numerous extension points for customization through a plug-in architecture. The Eclipse debugging perspective is a framework for building an integrating debuggers. The debugging perspective defines a set of user interfaces that define debugging functionalities (e.g. breakpoints, threads, and variables). The debugger perspective does not provide a concrete debugger implementation, but it offers a basic debugger interface that can be extended and adapted to a particular language's requirements. The user interface listens to debug events from the debug model interface and updates the user interface to show the debugged program's current state. Most debug event listeners are implemented as interfaces that can be extended and adapted to correspond to each debugger's specific behavior. The framework has a debugger re-interpreter that obtains a sequence of debugging commands from the specific debug model interface and queries the underlying command-line debugger.

Using this framework, a tool implementer does not have to implement a new debugger from scratch since the Eclipse debugging perspective is independent of the GPL and can be used with every existing GPL debugger. Using different GPL debuggers, the re-interpreter must be adapted due to the different APIs of the debuggers. For different types of DSLs, the specific part of the framework is the variable mapping component, which is represented as additional semantics in the DSL grammar. The framework's architecture, the step algorithm, and the mapping knowledge base are generic parts and can be reused across different tools and DSLs. However, the current version of the approach can only be used for textual languages and assumes that the generated artifact is code.

The GEMOC Studio[1] was invented by the GEMOC Initiative, an open and international community, which focuses on the coordinated usage of various modeling languages. GEMOC develops frameworks and environments to create, integrate, and automate the processing of modeling languages. The GEMOC Studio [65–68] provides functionalities

[1] http://gemoc.org/studio

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

for language designers and domain experts to build and compose new executable DSMLs. Furthermore, GEMOC Studio offers a mode to create and execute models confirming to such executable DSMLs. Like the GLS platform, the abstract syntax of the model is generated with the Ecore Modeling Framework and integrated into the GEMOC Studio. Using the Sirius Designer[2] for the graphical syntax and Xtext for the textual syntax, editing support for DSMLs is developed. In order to make the DSML executable GEMOC using any Java-based language, such as Java, Xtend[3] or Kermeta[4] [69, 70]. The assembling of the DSML concerns is made consistent utilizing the language workbench Melange[5] [71, 72] to define the execution semantics.

The GEMOC Studio [73] comes with an integrated model debugger to help the DSML designer during the model creation phase. Figure 7.1 illustrates the GEMOC model debugger highlighting the current stack frame of a finite-state machine model. The figure further shows the graphical editor on the left side, where the current state *S2* is highlighted and marked with a green arrow. Additionally, the example further includes a textual notation of the model on the right side next to the graphical representation. Furthermore, on top of the editor's views, the illustration shows the call stack view, variables view, and breakpoints view.

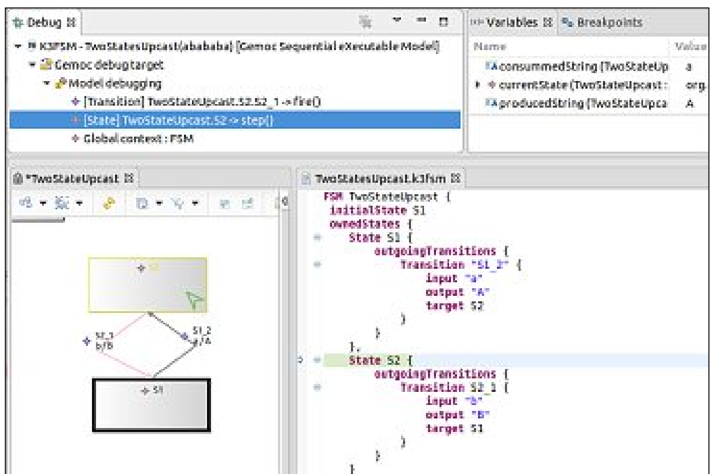

Figure 7.1: Model debugger of the GEMOC Studio [74].

The existing runtime services and model execution tools of the GEMOC Studio can be used for interpreted and compiled DSLs. Therefore, the GEMOC Studio offers a generic execution framework that provides various generic runtime services, such as graphical animation, omniscient debugging, and trace managers. Furthermore, the use of generic tools within GEMOC Studio is supported through the introduction of behavioral

[2]https://obeodesigner.com/en/product/sirius
[3]http://eclipse.org/xtend
[4]http://diverse-project.github.io/k3
[5]http://melange.inria.fr

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

interfaces. A behavioral interface defines a set of events specifying how external tools can interact with models that conform to executable DSLs implementing the interface. This allows the definition of abstract events that can be implemented by similar DSLs, enabling the use of generic tools over DSLs confirming to the same metalanguage. This approach is equivalent to the one we use in our work with the DAP, which consists of different interfaces to standardize the individual requests, events, and responses for debugging. The approach proposed by the GEMOC initiative allows using a reflective event injection GUI in conjugation with the generic debugger already provided by the GEMOC Studio. The extended debugger offers various debugging facilities (e.g. pause execution, use stepping operations, and set breakpoints) [75]. This corresponds to the generic debugger user interface we use in the course of our work. Figure 7.2 gives an overview of the proposed approach by the GEMOC initiative and shows the interaction between the individual entities.

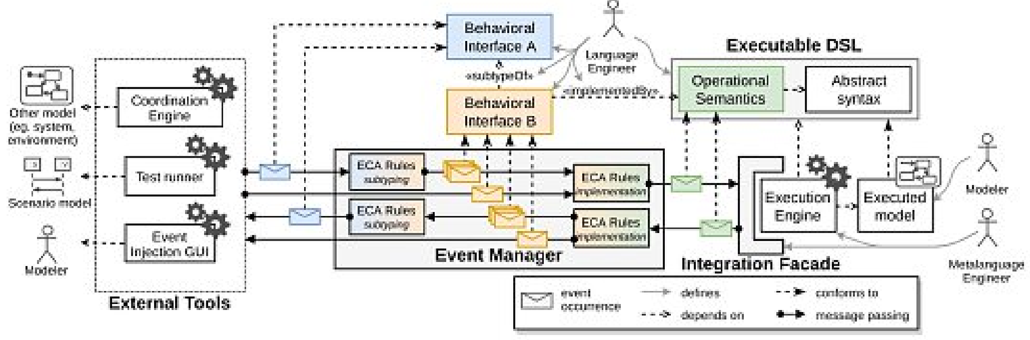

Figure 7.2: Overview of the proposed approach [75].

Furthermore, the approach used by GEMOC Studio includes an event manager. The event manager is responsible for dispatching event occurrences between relationships based on the behavioral interfaces referenced by the relationship. This implementation relationship describes how the xDSL provides the interaction capabilities that are expected a language typed by the behavioral interface. The implementation relationships are realized through Event-Condition-Action (ECA) rules. These rules are triggered when events from one side of the relationship satisfy their associated conditions. The events are then translated into new event occurrences belonging to the other side of the relationship. Finally, an integration façade for the event manager is introduced to act as an intermediary between the execution engine and the event manager. Furthermore, the integration façade is used to define the operational semantics of the xDSL [75]. This is similar to the debug adapter used in our work, which adapts the debugger's domain-specific logic to the generic DAP.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

In the following, we would like to mention further work on existing debugging approaches for other languages: Ladybird debugging tool [76–78] for the DSML Sequencer [79], debugging for the Unified Modeling Language (UML) [80–84], debugging for the parallel DEVS DSL [85], generic standard debugging for executable DSLs [86–88], and domain-specific debugger definition approaches [89–91].

7.2 Web-based DSML Environments

The cloud-based modeling environments AToMPM [92], GenMyModel [93], and WebGME [94] provide client-server architecture via a communication protocol. These applications send an update to the server after each modification of the model. This makes it easier to keep the model consistent across all clients collaborating on the same model. Different DSMLs are integrated via plug-ins into the editor. Additionally, AToMPM offers a release mode and debug mode. In release mode, the input model is sent to a transformation engine lying on a dedicated server. The model is transformed completely on the server, and the resulting model is sent back to the client and then displayed on the canvas. The transformation is animated at the client's canvas and can be executed continuously or step-by-step in debug mode. Furthermore, breakpoints can be set to halt the execution at a specific point in the control flow. The model transformation execution is deployed as a plug-in and can be integrated into the AToMPM framework. It is unclear how the communication between the client and the server is implemented and whether a protocol similar to the DAP is used.

Several stand-alone web-based modeling environments (e.g., [95], [96], [97], [98]) were introduced. These modeling tools provide full editing functionalities for certain domain-specific languages in a monolithic environment. The graphical editor and its modeling operations can be used to design models from graphical modeling languages. All the logic to create or edit the models lies at the client-side, only the latest version of the model is stored on a database server and updated at the end of a session. In contrast, editors based on the GLSP provide only a user interface to execute model edit operations at the client, the logic to create and edit the model is hosted by the server. Hence, only server-side adaption is required to add the language logic to the environment.

7.3 Other Related Work

There already exist several implementations[6] of the Debug Adapter Protocol for textual languages within a client-server architecture. Among them are popular languages like Java, Python, Node, and C/C++, etc. These implementations provide valuable insights into the communication between the development tool and the language-specific debugger. Components such as the npm-modules for the debug-adapter and the debug-adapter protocol that are independent of the concrete syntax can be reused in this work.

[6] https://microsoft.github.io/debug-adapter-protocol/implementors/adapters

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 8

Summary and Conclusion

In this chapter, we first summarize our work on this thesis, and secondly present the main results. Furthermore, we compare our work with the related work presented in Chapter 7. Finally, we discuss the limitations of this work and list some remaining challenges that can be addressed in future work.

8.1 Summary

This thesis aimed at investigating the existing concepts of the DAP defined for debugging textual languages and how these concepts can be transferred to debugging graphical modeling languages implemented based on the GLSP. We analyzed the potential of using existing frameworks and components to reduce the effort of integrating new debuggers for graphical modeling languages into IDEs. For this, we analyzed the debugging process as well as the debugging concepts available in modern code debuggers. Afterward, we translated the concepts for debugging textual languages into the context of graphical modeling languages. Using the existing standardizes protocols GLSP, which is inspired by LSP, and DAP, as well as the web-based frameworks Theia and Sprotty, we developed debuggers for two different graphical modeling languages. The LSP separates the language-specific logic of a textual language from the integration into an editor. Like the LSP, the GLSP follows a similar approach but applies it to graphical modeling languages. Therefore, the GLS platform comes with a client and server framework to build web-based diagram editors. Using the DAP allows implementing one generic debugger interface for a development tool that can communicate with different language debuggers via so-called debug adapters. The debug adapters can be reused across multiple development tools, which significantly reduces the effort to support a new debugger in different tools. From the existing WFML, a graphical modeling language for designing workflows, we derived the requirements to investigate if the DAP fits. Based on the information we gained at analyzing the DAP, we implemented a debugger to ease the integration of debugging support for new graphical modeling languages.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

The following steps were taken to implement the debugger: First, we created a custom web-based IDE using the Eclipse Theia framework. In addition to the basic components of Theia, which are necessary for a basic IDE, we integrated the GLS platform as well as the Theia-debug extension. The GLS platform offers a diagram editor and a graphical language server with which we are able to create and edit WFML diagrams. The Theia-debug extension provides a generic debugger GUI, the logic to manage the individual debugging features, and allows the exchange of DAP messages between a development tool and a debug adapter. Second, we replaced the components of the Theia-debug extension that were not suitable for graphical modeling languages with custom implementations. Third, we implemented a custom debug adapter for the WFML. For this purpose, we used the existing vscode-debugadapter module, which includes a basic implementation of a debug adapter. We extended this debug adapter to meet the requirements of the WFML. Furthermore, the debug adapter connects to a remote server where the WFML debugger is located. Fourth, we developed a debugger for the WFML, which parses the model, is aware of the individual debugging features (e.g. breakpoints, steps, variables), executes the model, and interprets the individual model elements accordingly. Finally, we applied the DAP to the GLSP, in which we implemented custom GLSP actions and graphical representations to enable, among other things, setting a breakpoint in the diagram and highlighting the current stack frame.

The second DSML was defined to investigate the reusability of the developed debugger to integrate new graphical modeling language debuggers into web-based diagram editors. We can conclude that by using the DAP developed for textual languages a language-independent implementation for a large part of the requirements for graphical modeling languages is possible. We have managed to use the generic debugger GUI as well as the custom implementations developed for the WFML also for the newly defined SMML. For the requirements that we could not serve with the DAP, we used custom requests and events. These custom protocol messages (e.g. *GLSPBreakpoint*), which are additionally defined for graphical modeling languages, are recognized by the Theia-debug extension and the custom debug adapters. Furthermore, we were able to use the GLS platform to introduce language-independent GLSP actions to enable graphical animations in the diagram. These actions can be used by both the WFML and the SMML. The GLSP features (e.g. *breakpointFeature*) we have developed can be integrated into the individual model elements. Finally, it can be said that the components implemented in this work can be used as a starting point for the implementation of further graphical modeling languages and their associated tools (e.g. debugger, interpreter).

Furthermore, we discussed open problems and future work on the developed debugger.

8.2 Comparison with Related Work

As mentioned in Chapter 7, we could not find any literature on the exact topic of using a protocol for standardizing the communication between development tools and real debuggers to supports the debugging of graphical modeling languages. Therefore, the main difference between the related work and the work at hand is that related work

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

addresses debuggers deeply integrated into the IDE. At the same time, this thesis follows a generic way of integrating debuggers. Furthermore, most of the related work requires a local installation of the tools. However, this thesis focuses on a client-server architecture to provide a web-based modeling tool, including modern debugging facilities. The few related tools available in a web-based environment offer either no debug mode or deal with the debugging of model transformation. In this work, however, we deal with model interpretation and model debugging.

8.3 Limitations and Future Work

In this section, we discuss limitations of the current approach and present suggestions for future work on the developed debugger. This comprises further debugging features integrated into the existing debugger and extensions to the language constructs of the DSMLs in use.

8.3.1 Extending the Debugger Facilities

In the following, we present features that could be implemented to extend the current debugger.

Support of Multiple Threads. The implemented debuggers currently only support the execution of a single thread. Therefore, models that support parallelism cannot be executed correctly. The WFML used as the running example offers a language structure that supports parallelism in models. For the parallel execution of processes, the two activity nodes fork node and join node were defined in the WFML. Future work could analyze whether the WFML debugger can be extended to support parallel execution of models. For this, the DAP protocol messages for the execution of multiple threads should be used in connection with graphical modeling languages. Furthermore, it has to be analyzed to what extent the client framework of the GLS platform has to be adapted/extended to ensure the correct functioning of the graphical animations during the parallel execution.

Set Variables. Another important point for future work on this topic is the modification of variables during the model's execution. In addition to the *variable* request to receive the variables, the DAP also supports a *set variable* request to modify variable values when the execution is paused. It should be noted that a change of the variable value should be reflected appropriately at every level of abstraction, to ensure continued overall consistency. This affects the variable view within the debug view, the model source, and the graphical representation in the form of the diagram.

Graphical Animations. Finally, the framework developed can be extended by further graphical animations. The Sprotty framework underlying the GLS platform offers several additional features (e.g. visual transition) that can be used to implement the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

debugging concepts. The graphical representations implemented so far only show a small part of what the Sprotty framework would offer in terms of animations.

8.3.2 Utilizing Tools and Frameworks

Concerning RQ3, we used existing tools and frameworks (e.g. Theia, GLS platform, Sprotty, DAP) to implement the debugger. These tools and frameworks were themselves still in the development phase. It might come to compatible issues when upgrading the frameworks to their current version. Further work should, therefore, consider the latest versions of the components used.

8.3.3 Extending the DSMLs

Concerning the results for the RQ1 and RQ2, we would like to point out that we have created only two DSMLs in the course of this work and presented the results based on these two. The results could be theoretically different if we implemented further modeling languages. In the course of future work, further languages and domains should be implemented, and the existing framework, including the DAP, should be analyzed for its reusability. These further experiments should also provide information as to whether other languages also need further protocol messages. It can then be discussed whether a new protocol for graphical modeling languages should be defined.

Besides, defining new DSMLs, the two case study DSMLs used in this thesis can be extended by additional language constructs. In the WFML, instead of using the probability of weighted edges, the activity nodes could also use statements as guards (e.g. [*Temperature ok?*]: *Yes* | *No*). The SMML, in its current version, is an abstraction of what one knows from state chart diagrams [99]. For example, the SMML could be extended by the following actions:

- **Entry action:** This action is executed while entering a new state, regardless of the state transition through which the new state has been reached.
- **Exit action:** This action is executed when the current state is exited, regardless of the state transition over which the current state is exited.
- **Input action:** This action is generated depending on the current state and the input event. Several actions can be assigned to a state.

The entry and exit action do not require any modification of the debugger since it can be handled by the interpreter. Input actions are executed when a certain event occurs within a state. The *GLSPEvents* introduced by us could be used to handle the input events. The debugger must be able to determine whether this event triggers a transition to another state or performs the action within a state.

Regarding these extensions, it is important to analyze whether the existing components which were developed for actions can also deal with the above-defined actions.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Subsequently, additional language constructs such as the activity nodes of the WFML can be added to the SMML, where again, attention should be paid on reusability.

8.3.4 Debugging for Code Generation

For future work it can be considered whether, in addition to the model interpretation approach used in this work, the introduced code generation approach described in Section 2.1, can be implemented. In Section 2.3, we briefly talked about debugging model transformations. Debuggers for code generators could help the user fix errors in the translation process from the model to executable code.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Bibliography

[1] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice: Second Edition*. Morgan and Claypool Publishers, 2nd edition, 2017.

[2] Baris Tekin Tezel and Geylani Kardas. Towards Providing Debugging in the Domain-Specific Modeling Languages for Software Agents. In *Proceedings of the Second International Workshop on Debugging in Model-Driven Engineering (MDEbug 2018) co-located with ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018)*. CEUR Workshop Proceedings, 2018.

[3] Roberto Rodriguez-Echeverria, Javier L. C. Izquierdo, Manuel Wimmer, and Jordi Cabot. Towards a Language Server Protocol Infrastructure for Graphical Modeling. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, MODELS '18, pages 370–380. ACM, 2018.

[4] Microsoft. Debug Adapter Protocol documentation. `https://microsoft.github.io/debug-adapter-protocol`, Accessed: 2020-06.

[5] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design Science in Information Systems Research. In *Management Information Systems Quarterly*, volume 28, pages 75–105. Society for Information Management and The Management Information Systems Research Center, 2004.

[6] John Venable, Jan Pries-Heje, and Richard Baskerville. A Comprehensive Framework for Evaluation in Design Science Research. In *Proceedings of the 7th International Conference on Design Science Research in Information Systems. Advances in Theory and Practice*, pages 423–438. Springer, 2012.

[7] Ken Peffers, Marcus Rothenberger, Tuure Tuunanen, and Reza Vaezi. Design Science Research Evaluation. In *Design Science Research in Information Systems. Advances in Theory and Practice*, pages 398–410. Springer, 2012.

[8] Robert K. Yin. *Case Study Research: Design and Methods (Applied Social Research Methods)*. Sage Publications, 4th edition, 2008.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Bibliothek Your knowledge hub

[9] Sami Beydeda, Matthias Book, and Volker Gruhn. *Model-Driven Software Development*. Springer, 2005.

[10] Bran Selic. The Pragmatics of Model-Driven Development. In *Software, IEEE*, volume 20, pages 19–25. IEEE Computer Society Press, 2003.

[11] Douglas C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. In *Computer*, volume 39, pages 25–31. IEEE Computer Society Press, 2006.

[12] M. Voelter and I. Groher. Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In *11th International Software Product Line Conference (SPLC 2007)*, pages 233–242. IEEE Computer Society Press, 2007.

[13] Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley and Sons, Inc., 2006.

[14] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.

[15] Erwan Breton and Jean Bézivin. Towards an Understanding of Model Executability. In *Proceedings of the International Conference on Formal Ontology in Information Systems - Volume 2001*, FOIS '01, pages 70–80. ACM, 2001.

[16] Gustavo C. M. Sousa, Fábio M. Costa, Peter J. Clarke, and Andrew A. Allen. Model-Driven Development of DSML Execution Engines. In *Proceedings of the 7th Workshop on Models@run.Time*, ser. MRT '12, pages 10–15. ACM, 2012.

[17] K. A. Morris, J. Wei, P. J. Clarke, and F. M. Costa. Towards Adaptable Middleware to Support Service Delivery Validation in i-DSML Execution Engines. In *2012 IEEE 14th International Symposium on High-Assurance Systems Engineering*, pages 82–89. IEEE Computer Society Press, 2012.

[18] Mark Allison, Peter J. Clarke, and Xudong He. A Generic Model of Execution for Synthesizing Interpreted Domain-Specific Models. In *Procedia Computer Science*, volume 62, pages 495–504. Elsevier Science Publishers B. V., 2015.

[19] Stefan Bucur, Johannes Kinder, and George Candea. Prototyping Symbolic Execution Engines for Interpreted Languages. In *SIGARCH Comput. Archit. News*, volume 42, pages 239–254. ACM, 2014.

[20] Jack Herrington. *Code Generation in Action*. Manning Publications Co., 2003.

[21] M. Puschel, J. M. F. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, Jianxin Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code Generation for DSP Transforms. In *Proceedings of the IEEE*, volume 93, pages 232–275. IEEE Computer Society Press, 2005.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[22] Valentin Besnard, Matthias Brun, Philippe Dhaussy, Frédéric Jouault, David Olivier, and Ciprian Teodorov. Towards One Model Interpreter for Both Design and Deployment. In *Proceedings of the 3rd International Workshop on Executable Modeling (EXE 2017) colocated at MODELS 2017*. CEUR Workshop Proceedings, 2017.

[23] Tanja Mayerhofer and Philip Langer. A Runtime Model for fUML. In *Proceedings of the 7th Workshop on Models@run.time (MRT) co-located with the 15th International Conference on Model Driven Engineering Languages and Systems (MODELS'12)*, pages 53–58. ACM, 2012.

[24] Yoann Laurent, Reda Bendraou, and Marie-Pierre Gervais. Executing and Debugging UML Models: An FUML Extension. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1095–1102. ACM, 2013.

[25] Daniel A. Sadilek and Guido Wachsmuth. Prototyping Visual Interpreters and Debuggers for Domain-Specific Modelling Languages. In *Proceedings of the 4th European Conference on Model Driven Architecture: Foundations and Applications*, ECMDA-FA '08, pages 63–78. Springer, 2008.

[26] Erwan Bousse, Jonathan Corley, Benoit Combemale, Jeff Gray, and Benoit Baudry. Supporting Efficient and Advanced Omniscient Debugging for xDSMLs. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2015, pages 137–148. ACM, 2015.

[27] Martin Fowler. *Domain Specific Languages*. Addison-Wesley, 1st edition, 2010.

[28] Tomaz Kosar, Nuno Oliveira, Marjan Mernik, Maria João Varanda Pereira, Matej Crepinsek, Daniela Carneiro da Cruz, and Pedro Rangel Henriques. Comparing General-Purpose and Domain-Specific Languages: An Empirical Study. In *Computer Science and Information Systems*, volume 7, pages 247–264. ComSIS Consortium, 2010.

[29] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society Press, 2008.

[30] Mark von Rosing, Stephen White, Fred Cummins, and Henk Man. *Business Process Model and Notation (BPMN)*, volume 1, pages 429–453. Elsevier Science Publishers B. V. - Morgan Kaufmann, 2015.

[31] Stefano Ceri, Maristella Matera, Francesca Rizzo, and Vera Demaldé. Designing Data-Intensive Web Applications for Content Accessibility Using Web Marts. In *Commun. ACM*, volume 50, pages 55–61. ACM, 2007.

[32] Colin Atkinson and Thomas Kühne. Model-Driven Development: A Metamodeling Foundation. In *IEEE Softw.*, volume 20, pages 36–41. IEEE Computer Society Press, 2003.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

[33] Mark Strembeck and Uwe Zdun. An Approach for the Systematic Development of Domain-Specific Languages. In *Softw. Pract. Exper.*, volume 39, pages 1253–1292. John Wiley and Sons, Inc., 2009.

[34] Thomas Kühne. Matters of (Meta-) Modeling. In *Software and Systems Modeling*, volume 5, pages 369–385. Springer, 2006.

[35] Anneke Kleppe. Software Language Engineering: Creating Domain-Specific Languages Using Metamodels. Addison-Wesley, 2008.

[36] Business Informatics Group. Graphical Modeling Languages [PowerPoint slides]. Retrieved during the Model Engineering Course 2018. https://tiss.tuwien.ac.at/course/educationDetails.xhtml?courseNr=188923&semester=2018W&dswid=2551&dsrid=665, Technische Universität Wien, Accessed: 2020-06.

[37] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Published by Schmidt, David A., 1997.

[38] Shane Sendall and Wojtek Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. In *IEEE Softw.*, volume 20, pages 42–45. IEEE Computer Society Press, 2003.

[39] Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel, and Andrew Wood. Transformation: The Missing Link of MDA. In *Graph Transformation*, pages 90–105. Springer, 2002.

[40] Tom Mens and Pieter [Van Gorp]. A Taxonomy of Model Transformation. In *Electronic Notes in Theoretical Computer Science*, volume 152, pages 125–142. Elsevier Science Publishers B. V., 2006.

[41] K. Czarnecki and S. Helsen. Feature-Based Survey of Model Transformation Approaches. In *IBM Syst. J.*, volume 45, pages 621–645. IBM Corp., 2006.

[42] Raphael Mannadiar and Hans Vangheluwe. Debugging in Domain-Specific Modelling. In *Software Language Engineering*, pages 276–285. Springer, 2011.

[43] Jonathan Corley, Brian P. Eddy, Eugene Syriani, and Jeff Gray. Efficient and Scalable Omniscient Debugging for Model Transformations. In *Software Quality Journal*, volume 25, pages 7–48. Kluwer Academic Publishers, 2017.

[44] Ian Sommerville. *Software Engineering*. Pearson, 10th edition, 2015.

[45] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., 2nd edition, 2009.

[46] Educba. Introduction to Debugging. https://www.educba.com/what-is-debugging/, Accessed: 2020-06.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Bibliothek
Your knowledge hub

[47] Jack B. Dennis. Petri Nets. In *Encyclopedia of Parallel Computing*, pages 1525–1530. Springer, 2011.

[48] Hendrik Bünder and Herbert Kuchen. Towards Multi-editor Support for Domain-Specific Languages Utilizing the Language Server Protocol. In *Model-Driven Engineering and Software Development*, pages 225–245. Springer, 2020.

[49] Hendrik Bünder. Decoupling Language and Editor - The Impact of the Language Server Protocol on Textual Domain-Specific Languages. In *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development*, MODELSWARD 2019, pages 129–140. SCITEPRESS - Science and Technology Publications, Lda, 2019.

[50] Sven Efftinge and Markus Völter. oAW xText: A framework for textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit*, volume 32. EclipseCon, 2006.

[51] Microsoft. Language Server Protocol documentation. `https://microsoft.github.io/language-server-protocol`, Accessed: 2020-06.

[52] Eclipse Source. Graphical Language Server Protocol documentation. `https://www.eclipse.org/glsp`, Accessed: 2020-06.

[53] Eclipse Sprotty. Eclipse Sprotty documentation. `https://github.com/eclipse/sprotty/wiki`, Accessed: 2020-06.

[54] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, 2nd edition, 2008.

[55] Anatolii Bazko. Implementing the Debug Adapter Protocol for Eclipse Theia and Eclipse Che. `https://che.eclipse.org/implementing-the-debug-adapter-protocol-for-eclipse-theia-d55cac38c59d`, Accessed: 2020-08.

[56] Remo H. Jansen. InversifyJS documentation. `https://github.com/inversify/InversifyJS`, Accessed: 2020-06.

[57] Microsoft. Visual Studio Code Debugger Extension. `https://code.visualstudio.com/api/extension-guides/debugger-extension`, Accessed: 2020-06.

[58] James H. Hill and Aniruddha S. Gokhale. Using Generative Programming to Enhance Reuse in Visitor Pattern-based DSML Model Interpreters. In *IEEE Trans. SP*. Institute for Software Integrated Systems, Vanderbilt University, 2007.

[59] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

[60] Elisabeth Freeman, Eric Freeman, Bert Bates, and Kathy Sierra. Head First Design Patterns. O' Reilly and Associates, Inc., 2004.

[61] Refactoring Guru. Visitor Pattern. `https://refactoring.guru/design-patterns/visitor`, Accessed: 2020-06.

[62] Christian Hahn. A Domain Specific Modeling Language for Multiagent Systems. In *In Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems*, volume 1, pages 233–240. AAMAS 2008, 2008.

[63] Federico Bergenti, Eleonora Iotti, Stefania Monica, and Agostino Poggi. Agent-Oriented Model-Driven Development for JADE with the JADEL Programming Language. In *Computer Languages, Systems and Structures*, volume 50, pages 142–158. Elsevier Science Publishers B. V., 2017.

[64] Hui Wu, Jeff Gray, and Marjan Mernik. Grammar-Driven Generation of Domain-Specific Language Debuggers. In *Softw. Pract. Exper.*, volume 38, pages 1073–1103. John Wiley and Sons, Inc., 2008.

[65] B. Combemale, O. Barais, and A. Wortmann. Language Engineering with the GEMOC Studio. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 189–191. IEEE Computer Society Press, 2017.

[66] Benoit Combemale, Julien Deantoni, Olivier Barais, Arnaud Blouin, Erwan Bousse, Cédric Brun, Thomas Degueule, and Didier Vojtisek. A Solution to the TTC'15 Model Execution Case Using the GEMOC Studio. In *8th Transformation Tool Contest*. CEUR Workshop Proceedings, 2015.

[67] Erwan Bousse, Thomas Degueule, Didier Vojtisek, Tanja Mayerhofer, Julien Deantoni, and Benoit Combemale. Execution Framework of the GEMOC Studio (Tool Demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2016, pages 84–89. ACM, 2016.

[68] Dorian Leroy, Erwan Bousse, Manuel Wimmer, Benoit Combemale, and Wieland Schwinger. Create and Play your Pac-Man Game with the GEMOC Studio (Tool Demonstration). In *Proceedings of the 3rd International Workshop on Executable Modeling (EXE 2017) colocated at MODELS 2017*, pages 1–6. CEUR Workshop Proceedings, 2017.

[69] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving Executability into Object-Oriented Meta-languages. In *Model Driven Engineering Languages and Systems*, volume 3713, pages 264–278. Springer, 2005.

[70] Jean-Marc Jézéquel, Olivier Barais, and Franck Fleurey. Model Driven Language Engineering with Kermeta. In *Generative and Transformational Techniques in Software Engineering III: International Summer School, GTTSE 2009, Braga, Portugal, July 6-11, 2009. Revised Papers*, volume 6491, pages 201–221. Springer, 2011.

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

[71] Thomas Degueule, Benoit Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. Melange: A Meta-Language for Modular and Reusable Development of DSLs. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2015, pages 25–36. ACM, 2015.

[72] Thomas Degueule, Benoit Combemale, Arnaud Blouin, and Olivier Barais. Reusing legacy dsls with melange. In *Proceedings of the Workshop on Domain-Specific Modeling*, DSM 2015, pages 45–46. ACM, 2015.

[73] Erwan Bousse, Tanja Mayerhofer, and Manuel Wimmer. Domain-Level Debugging for Compiled DSLs with the GEMOC Studio (Tool Demo), 1rst International Workshop on Debugging in Model-Driven Engineering (MDEbug 2017). In *Proceedings of MODELS 2017 Satellite Event*. CEUR Workshop Proceedings, 2017.

[74] GEMOC Group. GEMOC Studio Documentation. `https://download.eclipse.org/gemoc/docs`, Accessed: 2020-06.

[75] Dorian Leroy, Erwan Bousse, Manuel Wimmer, Tanja Mayerhofer, Benoit Combemale, and Wieland Schwinger. Behavioral interfaces for executable DSLs. In *Software and Systems Modeling*, volume 19, pages 1015–1043. Springer, 2020.

[76] Tomaž Kos, Marjan Mernik, and Tomaž Kosar. A Tool Support for Model-Driven Development: An Industrial Case Study from a Measurement Domain. In *Applied Sciences*, volume 9, page 4553. MDPI, 2019.

[77] Tomaž Kos, Tomaž Kosar, Marjan Mernik, and Jure Knez. Ladybird: Debugging Support in the Sequencer. In *Proceedings of the 2011 American Conference on Applied Mathematics and the 5th WSEAS International Conference on Computer Engineering and Applications*, AMERICAN-MATH'11/CEA'11, pages 135–139. WSEAS, 2011.

[78] Tomaž Kosar, Marjan Mernik, Jeff Gray, and Tomaž Kos. Debugging Measurement Systems using a Domain-Specific Modeling Language. In *Computers in Industry*, volume 65, pages 622–635. Elsevier Science Publishers B. V., 2014.

[79] Tomaz Kos, Tomaž Kosar, Jure Knez, and Marjan Mernik. Improving End-User Productivity in Measurement Systems with a Domain-Specific (Modeling) Language Sequencer. In *ADBIS*, volume 639, pages 61–76. CEUR Workshop Proceedings, 2010.

[80] Lidia Fuentes, Jorge Manrique, and Pablo Sánchez. PóPulo: A Tool for Debugging UML Models. In *Companion of the 30th International Conference on Software Engineering*, ICSE Companion '08, pages 955–956. ACM, 2008.

[81] Michelle L. Crane and Juergen Dingel. Towards a UML Virtual Machine: Implementing an Interpreter for UML 2 Actions and Activities. In *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, CASCON '08. ACM, 2008.

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

[82] Andrei Kirshin, Dolev Dotan, and Alan Hartman. A UML Simulator Based on a Generic Model Execution Engine. In *Proceedings of the 2006 International Conference on Models in Software Engineering*, MoDELS'06, pages 324–326. Springer, 2006.

[83] Dirk Riehle, Steven Fraleigh, Dirk Bucka-Lassen, and Nosa Omorogbe. The Architecture of a UML Virtual Machine. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, pages 327–341. ACM, 2001.

[84] Dolev Dotan and Andrei Kirshin. Debugging and Testing Behavioral UML Models. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*, OOPSLA '07, pages 838–839. ACM, 2007.

[85] Simon Van Mierlo, Yentl Van Tendeloo, and Hans Vangheluwe. Debugging Parallel DEVS. In *Simulation*, volume 93, pages 285–306. Society for Computer Simulation International, 2017.

[86] Benoît Combemale, Xavier Crégut, Jean-Pierre Giacometti, Pierre Michel, and Marc Pantel. Introducing Simulation and Model Animation in the MDE Topcased Toolkit. European congress on Embedded Real Time Software and Systems (ERTS), 2008.

[87] I. Rath, D. Vago, and D. Varro. Design-time simulation of domain-specific models by incremental pattern matching. In *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 219–222. IEEE Computer Society Press, 2008.

[88] Nils Bandener, Christian Soltenborn, and Gregor Engels. Extending DMM Behavior Specifications for Visual Execution and Debugging. SLE'10, pages 357–376. Springer, 2010.

[89] Ricky T. Lindeman, Lennart C.L. Kats, and Eelco Visser. Declaratively Defining Domain-Specific Language Debuggers. In *SIGPLAN Not.*, volume 47, pages 127–136. ACM, 2011.

[90] Andrei Chiş, Tudor Gîrba, and Oscar Nierstrasz. The Moldable Debugger: A Framework for Developing Domain-Specific Debuggers. In *Software Language Engineering*, pages 102–121. Springer, 2014.

[91] Andrei Chiş, Marcus Denker, Tudor Gîrba, and Oscar Nierstrasz. Practical Domain-Specific Debuggers Using the Moldable Debugger Framework. In *Comput. Lang. Syst. Struct.*, volume 44, pages 89–113. Elsevier Science Publishers B. V., 2015.

[92] Eugene Syriani, Hans Vangheluwe, Raphael Mannadiar, Conner Hansen, Simon Van Mierlo, and Hüseyin Ergin. AToMPM: A Web-based Modeling Environment. In *Demos/Posters/StudentResearch@MoDELS*. CEUR Workshop Proceedings, 2013.

[93] Michel Dirix, Alexis Muller, and Vincent Aranega. GenMyModel : An Online UML Case Tool. European Conference on Object-Oriented Programming, 2013.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[94] Miklós Maróti, Tamás Kecskés, Róbert Kereskényi, Brian Broll, Péter Völgyesi, László Jurácz, Tihamer Levendovszky, and Ákos Lédeczi. Next generation (Meta)modeling: Web- and cloud-based collaborative tool infrastructure. In *MPM@MoDELS*, volume 1237, pages 41–60. CEUR Workshop Proceedings, 2014.

[95] Louis M. Rose, Dimitrios S. Kolovos, and Richard F. Paige. EuGENia Live: A Flexible Graphical Modelling Tool. In *Proceedings of the 2012 Extreme Modeling Workshop*, XM '12, pages 15–20. ACM, 2012.

[96] José P. Leal, Helder Correia, and José C. Paiva. Eshu: An Extensible Web Editor for Diagrammatic Languages. In *5th Symposium on Languages, Applications and Technologies (SLATE'16)*, volume 51 of *OpenAccess Series in Informatics (OASIcs)*, pages 12:1–12:13. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016.

[97] Christian Thum, Michael Schwind, and Martin Schader. SLIM–A Lightweight Environment for Synchronous Collaborative Modeling. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, volume 5795 of *MODELS '09*, pages 137–151. Springer, 2009.

[98] Shuhei Hiya, Kenji Hisazumi, Akira Fukuda, and Tsuneo Nakanishi. Clooca: Web based tool for Domain Specific Modeling. In *Proceedings of Demos/Posters/StudentResearch@ MoDELS*, volume 1115. CEUR Workshop Proceedings, 2013.

[99] José A. Cruz-Lemus, Ann Maes, Marcela Genero, Geert Poels, and Mario Piattini. The Impact of Structural Complexity on the Understandability of UML Statechart Diagrams. In *Inf. Sci.*, volume 180, pages 2209–2220. Elsevier Science Publishers B. V., 2010.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub