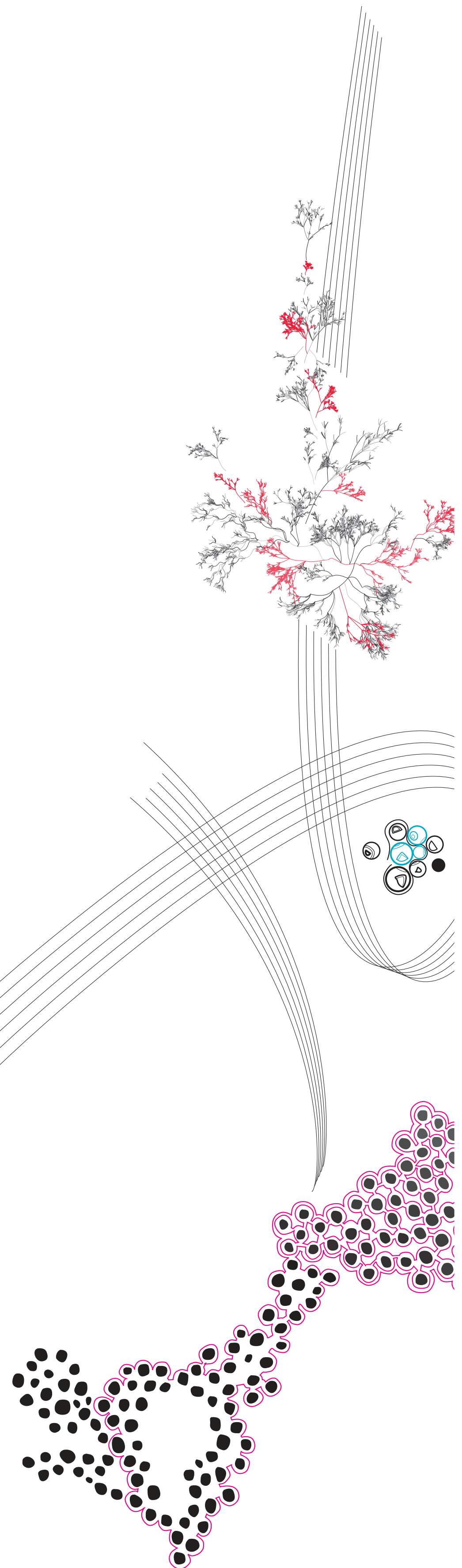

MSc in Computer Science
Final Project

Extensibility Of Domain-Specific Languages: A Case Study of an Industrial DSL

Naum Tomov

Supervisors:
Vadim Zaytsev &
Nhat Bui

July 6, 2024

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics, and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Contents

Glossary and Abbreviations 1

1 Introduction 3
- 1.1 Graduation Project Task Background 4
- 1.2 Problem Statement 4
- 1.3 Contribution 5
- 1.4 Outline 5

2 Background 6
- 2.1 Relevant DSL Terminology 7
- 2.2 Software Language Extensibility 8

3 Research Questions 9
- 3.1 RQ1: What are factors that influence the extensibility of a DSL? 10
- 3.2 RQ2: What are the current challenges in extending APS? 10
- 3.3 RQ3: How can existing software language design and extensibility guidelines be used to effectively evaluate the state of a DSL? 10
- 3.4 RQ4: What are effective and actionable guidelines for DSL extensions to ensure maintainability? 11

4 Related Literature 12
- 4.1 Meta-Literature 13
- 4.2 Domain-Specific Language Design 15

5 RQ1. Extensibility Factors of DSLs 17
- 5.1 Methodology 18
- 5.2 Results 18
  - 5.2.1 DSL Construction as Software Engineering 19
  - 5.2.2 Language Workbenches (LWs) 21
  - 5.2.3 Software Language Grammars 23
  - 5.2.4 DSL Usability 24
- 5.3 Discussion 24

6 RQ2. Current Challenges in Extending APS 26
- 6.1 Methodology 27
- 6.2 Results 27
  - 6.2.1 The Challenges in the Development of APS 27
- 6.3 Discussion 28

7 RQ3. Extraction and Application of Modern DSL Design Guidelines from Academia 31
7.1 Methodology . . . 32
7.2 Results . . . 33
7.3 Discussion . . . 40

8 RQ4. DSL Extension Guidelines 41
8.1 Methodology . . . 42
8.2 Results . . . 43
8.2.1 Extension guidelines for APS . . . 43
8.2.2 Validation of Guidelines via Retroactive Application . . . 43
8.3 Discussion . . . 47

9 Conclusion 48
9.1 Summary of Findings . . . 49
9.2 Contributions to the Field . . . 49
9.3 Threats to Validity . . . 52
9.3.1 Internal Validity . . . 53
9.3.2 External Validity . . . 53
9.4 Future Work . . . 53

A Internship Task Description 62
A.1 Background Information . . . 62
A.2 Assignment . . . 62

B Developer Interviews 63
B.1 Consent Brochure for Interview . . . 63
B.2 Questions for Interview . . . 65
B.3 Interview Answers . . . 66

Abstract

Domain-specific languages (DSLs) are software languages made for a certain domain. They provide an interface for domain experts to write expressive, yet readable code. One major disadvantage of designing and developing DSLs is the cost of maintenance and evolution of the language. A DSL needs to evolve to reflect changes in the domain it represents, to provide new functionality, and to address end-user demands. Software language engineering is a developing discipline and systematic techniques as well as measurements are presently lacking. This thesis aims to outline and address this gap, starting with the need for more objective ways to evaluate languages and arriving at a set of extension guidelines for an industrial DSL. A case study is performed on a DSL from the industry to learn about the challenges faced in practice while developing domain-specific languages and to provide tangible examples of how to procure relevant extension guidelines for domain-specific languages.

Keywords: domain-specific languages, software language design, software language engineering, model-driven engineering, DSL evolution, DSL extensibility, DSL evaluation, DSL design guidelines, DSL challenges

Glossary and Abbreviations

Glossary

- Abstract Syntax Tree
  An abstract syntax tree (AST) is a data structure used to represent the structure of a program or code snippet, generally as a product of parsing.

- Back End
  In the context of this thesis, the back end refers to the back end of the domain-specific language. It is responsible for converting an abstract syntax tree into the desired output for the project. In this case study, the back end was mainly responsible for generating C++ code.

- Front End
  The front end of a code generator is responsible for creating an abstract syntax tree for a given input. This usually involves lexing, parsing and some form of data representation optimisation.

- Grammarware
  Grammarware comprises grammars and all grammar-dependent software, i.e., software artefacts that directly involve grammar knowledge.

- Language Workbench
  To facilitate the development of DSLs and to avoid needing to re-invent the wheel each time a software language construction project is undertaken, language workbenches (LWs) have been created. LW is a term popularised by Martin Fowler in 2005 [21]. Various tools out there achieve the goal of streamlining the process of DSL development in varying degrees. In 2013, S. Erdweg et al. defined 34 features for language workbenches [18]. To qualify as a language workbench, a tool must satisfy a sensible, even if limited, amount of these features. The usage of LWs when building languages is considered a good practice, as their purpose is to make the development of new languages affordable and efficient [13] [18].

- Metaphrasing
  Metaphrasing is the change in the interpretation of existing language rules, processing old expressions in new ways. This inherently requires work on the language processor and requires significant development effort. This terminology is borrowed from the paper "Extensibility in programming languages" [62] by Thomas Standish.

- Orthophrasing
  Orthophrasing comes from "orthogonal", in that orthogonal features are introduced into the language. Those are features that the existing language structure cannot capture, thus introducing them requires working on the language processor itself.

This terminology is borrowed from the paper "Extensibility in programming languages" [62] by Thomas Standish.

- Paraphrasing
  Paraphrasing is the language extension method that modern programmers are most familiar with as it involves using language features to define new ones. This is enabled by meta-programming features of languages that allow the developer to define types and operators as well as create macros and extend control structures. These types of extensions are the most common and easiest to implement, but they are limited to a pre-determined scope of what the language allows. This terminology is borrowed from the paper "Extensibility in programming languages" [62] by Thomas Standish.

- Usability
  The "extent to which a system, product or service can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use" [29] as per the International Organisation for Standardisation.

List of Abbreviations

- APS — The ASML Parameter Specification language
- ASML — The company which suggested this graduation project. They specialise in the production of lithography machines for the semiconductor sector. The abbreviation does not stand for anything.
- AST — Abstract Syntax Tree
- CI — Continuous Integration
- DSL — Domain-Specific Language
- GPL — General Purpose Language
- LW — Language Workbench
- SLE — Software Language Engineering
- SLR — Systematic Literature Review
- SMS — Systematic Mapping Study

Chapter 1

Introduction

Software has proven to be incredibly versatile and useful in a plethora of fields, such as mathematics, biology, physics, commerce, etc. While software can enable experts from different fields to take advantage of the processing power of computers, these specialists are rarely also apt computer scientists. Thus, software engineers have to work with domain experts and try to bridge the gap between abstract domain expertise and low-level programming implementation details. This challenge makes it difficult for both parties to ensure that desired functionalities work as intended by the abstract domain model. Developers lack the knowledge to understand changes within the domain model and experts have a hard time ensuring that all of their abstract knowledge has been translated correctly.

Domain-specific languages (DSLs) [22] are built to describe the concepts of a certain domain with expressive and semantically rich notation. DSL developers aim to narrow the gap between the level of abstraction in a certain domain and the one provided by the software language. They enable, even enforce, code reuse in a similar fashion as a programming library would. In the context of this research, the focus is to help enable the evolution of a DSL, leveraging existing language design patterns and best practices. The thesis project is a task proposed by ASML as a graduation project.

1.1 Graduation Project Task Background

This project is an industrial case study of a DSL developed and maintained by ASML, called ASML Parameter Specification (APS). The language serves as an interface to provide lithography machines with input settings. A code generator then generates C++ code for the machines. This language must continuously evolve, as more types of input settings become available for the machines. It uses the language workbench Xtext [19].

The task, as provided by ASML, is, among other things, to devise guidelines for the extension of the language. It was created in 2019 and has been under iterative development ever since, with iterations being delivered every quarter. APS has grown substantially since then. They have started to see an opportunity for it to be used outside of the team which originally developed it. This has created the necessity for formal extension guidelines and an examination of the impact of the language on its surrounding interfaces. A strong emphasis is placed on backward compatibility, due to its existing usage and processes' dependence on it. For the details of the assignment, refer to the Internship Task Description in Appendix A.

1.2 Problem Statement

Software language evolution studies the nature of the change in programming languages. This change is made to expand the functionality, increasing the expressiveness and ensuring the consistency and correctness of the language. Languages such as Java and C++ have undergone significant changes since their initial releases [10], adding support for a plethora of features that enable new functionality within these languages.

DSLs, as software languages, are also subject to evolution. New features can be introduced, based on user demand, or changes in the represented domain. However, such changes ideally should not have an impact on already existing instances of that language. If there are such oversights, it would lead to needing a significant overhaul of code bases, or, more likely, the lack of adoption for the new version of the language. As an example, the migration of the Python language from version 2 to 3, which was not backwards-compatible, has had an impact on adoptability more than 10 years later [46]. Some developers expend resources on supporting two versions of their libraries and are locked out of using new, exclusive features in the new version. Furthermore, extensions to the language should not hinder future work, as the evolution of languages is continuous and without a final destination. This creates the need for extension guidelines, which ensure both backward and forward compatibility, as well as minimal impact on the language core code base.

The main challenge to be tackled is the lack of language design benchmarks. No systematic framework can be followed or applied to language grammars or semantic concepts. The state-of-the-art in DSL design will be researched, to find a way to systematically evaluate the APS language. Moreover, software language evaluation frameworks are also lacking, literature on extensibility is scarce and the term "extensibility" itself is ambiguous.

In this thesis, extensibility will be related to the design of the language and explored through multiple lenses. Usability and learnability play a role in the language's ability to extend its potential user base. Whereas its adherence to design patterns and development guidelines can indicate its ability to be syntactically and semantically enriched.

Finally, a modern formulation of the extensibility challenge can be found in "The Software Language Extension Problem" by M. Leduc et al. from 2020 [44]. This paper formulates the software language extension problem using the old "expression problem" [52]. That is a now-classical problem in programming languages referring to the difficulty of

writing data abstractions that can be easily extended with both new operations and new data variants. This shows that the challenge of this thesis, the extensibility of languages, is still an open problem with no systematic, research-based solution.

1.3 Contribution

This thesis makes several contributions to the field of DSLs, particularly in the context of extensibility. Firstly, it addresses the ambiguity surrounding the definition of "extensibility" by proposing a clear and comprehensive interpretation that can be used in both academic and industrial settings. This interpretation includes various facets such as usability, learnability, and the ability to syntactically and semantically enrich the language.

Secondly, the thesis highlights the lack thereof and proposes a novel, objective evaluation method for assessing the extensibility of DSLs. This method leverages best practices from DSL language engineering and reduces them to relevant criteria for a language's design. By applying this method to the ASML Parameter Specification language, the thesis provides a practical example of how to evaluate and improve a real-world industrial DSL.

Thirdly, the research introduces a set of guidelines for extending DSLs, ensuring that these guidelines are both actionable and effective. These guidelines were derived through consulting literature and using the industrial DSL in the case study. They are a stride towards a more systematic approach to evolution-enabling design.

Finally, the thesis contributes to the currently limited library of knowledge on vertical, embedded DSLs. By examining APS, an industrial DSL with specific use cases and constraints, the research provides insights that can apply to other similar DSLs. This case study highlights the challenges and successes of DSL extensibility and offers practical solutions that other DSL developers can adopt.

1.4 Outline

This thesis is organised into several chapters, each focusing on different aspects of the research and its findings. After this introductory chapter, there is a Background chapter with additional preliminary information surrounding this thesis. This includes definitions of key terms and concepts related to DSLs and their extensibility.

Chapter 3 presents the research questions that guide this thesis. These questions are designed to explore the factors influencing DSL extensibility, the challenges faced in extending APS, the applicability of existing design guidelines, and the development of new, actionable guidelines for DSL extension.

This is followed by a Related Literature chapter to help contextualise the following chapters, which answer the research questions. Each Research Question chapter lays out the methodology used, the result obtained, and a reflective discussion about the results and their relation to the rest of the thesis. Finally, there is a Conclusion chapter, which summarises the findings, includes the final list of DSL extension guidelines and proposes ideas for future work.

Chapter 2

Background

This chapter contains essential foundational knowledge required to understand the subsequent discussions and analyses in this thesis. Critical concepts and terminology surrounding DSLs are defined, setting the stage for a deeper exploration of extensibility and evolution in software language engineering. The background information presented here is also useful for contextualising the industrial case study included. Understanding the key terms and theoretical underpinnings enables the readers to grasp the complexities and nuances of this thesis.

2.1 Relevant DSL Terminology

Defining the right vocabulary to meaningfully examine domain-specific languages is imperative. It enables the recognition and categorisation of key aspects of DSLs and their evolution. The "Dynamic Language Embedding With Homogeneous Tool Support" [58] paper provides useful terminology to categorise DSLs as:

- **Internal** [58] — DSLs built within a host language as a fluent application programming interface [22]. This means they piggyback off the host language's implementation and rely on its parser and compiler/interpreter.
- **External** [58] — DSLs built from scratch, with their own parser, lexer, and accompanying compilation technologies. They are a lot more independent and flexible than internal DSLs but require a lot more design and development work.
- **Embedded** [58] — DSLs built into an environment meant for extension, such as Xtext [19]. These environments, called language workbenches, allow for the easy development of languages.

The language in this case study is embedded, though embedded and external languages are similar in more ways than not.

Furthermore, DSLs fall into different camps based on their form of notation — "textual", "graphical", "tabular" and "symbolic". This, of course, reflects the way that the end-user can define instances of that language. APS is a textual DSL, which means that language instances are text files, reminiscent of popular programming languages. Nonetheless, they are still not programs, as APS is not a programming language. It is a specification language, used for specifying options, without control-flow logic.

Other relevant terms about DSLs include "vertical" versus "horizontal" languages [39]. Vertical languages are focused on narrow use cases and are also known as business languages. Horizontal languages are focused on a broader domain with more use cases. With the given terminology, APS can be defined as a vertical, embedded DSL. A relevant quote from the paper "A reflection on the lack of adoption of domain-specific languages" [66], which uses the same terminology:

> "Vertical external DSLs directed to non-developers can be transformative for an organisation. They require the deployment of a significant effort and need the support of the whole organisation or a large unit, such as a department. They are typically multi-year projects with far-reaching consequences in terms of productivity. They require adequate planning as they need to be supported in the long term. This kind of DSL is probably the least well-known by a broader audience."

This is an important consideration for APS, as it shows the lack of awareness around these types of language. It also indicates that it is unlikely to be able to find examples of similar languages to draw from.

To facilitate the development of DSLs and to avoid needing to re-invent the wheel each time a software language construction project is undertaken, language workbenches (LWs) have been created. LW is a term popularised by Martin Fowler in 2005 [21]. Various tools out there achieve the goal of streamlining the process of DSL development in varying degrees. In 2013, S. Erdweg et al. defined 34 features for language workbenches [18]. To qualify as a language workbench, a tool must satisfy a sensible, even if limited, amount of these features. The usage of LWs when building languages is considered a good practice, as their purpose is to make the development of new languages affordable and efficient [18] [13].

2.2 Software Language Extensibility

It is important to note a central theme in this thesis — "extensibility" is overloaded and potentially misleading in the context of software languages. The paper "Extensibility in programming language design" [62] by Thomas Standish in 1975 gives us a framework to define different types of extensibility, predating modern programming languages. The paper's age notwithstanding, it is readily applicable to this discourse at the time of writing this thesis. Three types of extension techniques are defined, which may be relevant for this project: *paraphrasing*, *orthophrasing* and *metaphrasing*. The terms are defined in the Glossary. For this thesis, extensibility is explored under the guide of *orthophrasing* alone, which is the addition of features to the language. A software language getting orthogonal extensions over time is commonly referred to as language evolution [15]. Hence, enabling extensibility means enabling the language to evolve.

A lot of literature [17] [16] [33] focuses on the *paraphrasing* aspect of extensibility under Standish's terms [62], which is the **user's ability to extend the language**. This type of extensibility is not directly related to this research, therefore literature regarding it is only tangential to this thesis. This differs from how extensibility is normally understood in software engineering, which refers to the quality of being **designed to allow the addition of new capabilities** or functionality [31]. In the context of this thesis, the former, user-oriented definition is discarded, while the latter is expanded in the following capacity — the "extensibility" of a DSL, to be evaluated and prioritised in the extension guidelines, comprises of:

- Enriching the language vocabulary and syntax. This is the most intuitive understanding of extending a language and will be discussed in depth in this project.
- Increasing the tool support of the language, e.g. plugins for it, debug support, etc. This is an important aspect of language development but not the main focus of this project.
- Extending the language's end-user base, This is a more liberal interpretation of the term "extensibility", but in the context of the project, one aim is to procure guidelines that define extensions which enable the language to be adopted.

Sebastian Erdweg et al. explore this from a more theoretical perspective in the paper "Language Composition Untangled" [15], which gives a classification for different types of language composition, one of which is language extension. These extensions depend on the base language and their composition over time constitutes a language's evolution, such as the addition of generics and 'foreach' loops in Java, which were not present in the initial language. Noteworthy is the so-called incremental extension, which is the composition of language extensions one after another, as is seen in the evolution of languages.

Later, in 2013 S. Erdweg et al. developed A Framework for Extensible Languages [17], which aims to transform non-extensible languages into extensible ones, allowing for customisable syntax, static analysis, and editor support. The framework only needs the base language's grammar, import statement syntax, and base-language compilation method. The generality of the framework is demonstrated through its application to several general-purpose languages, including Java and Haskell. This paper refers to extensibility as the user's ability to introduce new features and is thus tangential.

Chapter 3

Research Questions

This thesis is carried out under a Research Question framework [56], where research questions (RQs) are defined, such that their answers shed light on the central topic of the thesis — the evaluation and extensibility of DSLs. The RQs aim to encapsulate the task provided by ASML (see appendix A) into questions which can be answered systematically.

3.1 RQ1: What are factors that influence the extensibility of a DSL?

This is the starting point of the thesis and serves to provide a direction for the subsequent research questions. To answer this question, this thesis will lean on the literature about DSLs. The answer is crucial for providing research focus of the thesis and enables a more thorough investigation of the latter portions of the research.

3.2 RQ2: What are the current challenges in extending APS?

This question delves into identifying specific challenges encountered in the process of extending APS. It aims to identify these challenges through interviews, to provide an understanding of what should be the focus of the extensibility guidelines. This will allow for the guidelines to be tailored for the specific use case of the language in this case study, as well as serve as a reference point for future readers who want to apply the extension guidelines to their language. If the challenges identified in this case study are not experienced in the development of another language, then the guidelines may also have limited usefulness.

3.3 RQ3: How can existing software language design and extensibility guidelines be used to effectively evaluate the state of a DSL?

This question explores the applicability of existing language design guidelines in assessing a DSL's state. It seeks to understand whether they can provide a robust framework for evaluating the DSL's design, structure, and extensibility potential. Carrying it out requires both identifying the state-of-the-art design patterns and then applying them to the language. This will involve a literature review, validation with the APS developers regarding the relevance of the collected design guidelines and finally the manual identification of APS' compliance with the state-of-the-art design recommendation in academia. Because this is a multi-faceted question, answering it will require answering the following sub-questions:

RQ3.1: Are there established, systematic ways to evaluate a DSL holistically?

Since this RQ aims to synthesise an evaluation method based on existing guidelines, it is worth exploring what are the existing evaluation methods for DSLs. This sheds light on whether or not this approach has merit, or if other established methods can be used for DSL evaluation instead.

RQ3.2: What are the existing software language design guidelines?

Examining the state-of-the-art in academia on the topic of software language design should produce a set of relevant guidelines which could be retroactively applied to a language to assess its state.

RQ3.3: What are the existing software language extensibility guidelines?

Finding the agreed-upon extensibility guidelines in the literature is essential for this research, as a central focus is placed on the notion of extensible language design.

RQ3.4: How can the answers of RQ3.1, RQ3.2, and RQ3.3 be leveraged to evaluate the state of a DSL?

Once the guidelines and assessment methods are obtained, RQ3 must be answered — how can they be applied, if at all, to assess the state of a DSL? If no suitable evaluation methods are identified, the design guidelines can be turned into an evaluation framework, wherein adherence to design standards is scrutinised.

Finding the answer to these sub-questions will provide the necessary insight into DSL construction literature to answer the main RQ. To further assist with these questions, the industrial DSL from this case study will be used as an example of the evaluation.

3.4 RQ4: What are effective and actionable guidelines for DSL extensions to ensure maintainability?

This question is the culmination of this thesis and aims to devise a set of practical guidelines that can be applied when extending a DSL. The generality of these guidelines to DSLs outside this case study is not guaranteed and an evaluation will be carried out to assess their external validity.

The focus is on actionable strategies that ensure the DSL's evolution is manageable, maintainable, and user-friendly. "Actionable" refers to them being specific enough to allow for actions to be taken as a direct consequence of the guidelines. For them to have a net positive effect on the process, would classify them as "effective". Answering the question will involve collating the answers of the previous RQs to formulate a set of evidence-backed extension guidelines. Finally, they will be validated by retroactively applying them on past extensions and exploring what impact they would have had, if any. They can then be adjusted to be more effective and to ensure their relevance.

Chapter 4

Related Literature

Domain-specific languages have existed and been a part of computer science academia's consciousness for over half a century [48]. However, they have been called different names, such as "application-oriented", "specialised", "task-specific", etc. The term DSL was solidified and properly defined by L. Walton in 1996, according to L. Renggli [58]. Regardless of the definition, there is still no solid line distinguishing a DSL from a general-purpose language. If a DSL expands and adopts new features, it can eventually evolve into a GPL, however, this line is unclear. In fact, A. Kleppe asserts in her book "Software language engineering: creating domain-specific languages using metamodels" that

> "... I must conclude that no specific characteristics make DSLs different from any other software language." [39]

Thus, literature regarding generic software language engineering, design, and evaluation is also related and applicable to DSLs.

Over this period, a significant body of literature has been built up, both about software language engineering as a whole and DSLs in particular. The literature is sufficient for the relatively recent emergence of systematic literature reviews (SLRs), as well as systematic mapping studies (SMS) [7] [13] [25] [30] [43] [55] [64]; due to this abundance of research, DSL development can be classified as a mature research field. Diving into this literature gives the reader more context necessary to understand the rest of this thesis.

4.1 Meta-Literature

This section is dedicated to the aforementioned literature reviews and mapping studies — the literature about other literature on the topic (i.e. meta-literature).

P. Gabriel et al.: Do software language engineers evaluate their languages? 2011 [25]

Although not self-prescribed as an SMS or SLR, this paper delves into SLE literature to answer the question in its title. Due to the relatively old age of this paper, the results are not necessarily applicable nowadays, but it is an important stepping stone in the field of software language evaluation. The authors found that only 2 out of 36 DSL papers reported an industrial-level assessment of the language. Similarly, 29 of the papers had no usability evaluation reported. This led the authors to call for the SLE to include evaluation as standard practice in the development process and concluded that the software industry was not investing much in the evaluation of DSLs at the time.

A. Barisic et al.: Domain-specific language domain analysis and evaluation: a systematic literature review, 2015 [7]

This SLR focuses on graphical DSLs, thus the findings are not necessarily generalisable to all DSLs, nor to the specific DSL of this case study. However, it makes the case for the importance of usability in DSLs and reports on an increase in the usability evaluation of papers as opposed to the previous paper by P. Gabriel et al. However, it still does not define or outline a systematic approach for usability evaluation as a part of the software language development process.

T. Kosar et al.: Domain-Specific Languages: A Systematic Mapping Study, 2016 [43]

This SMS digs into the research space in the field of DSLs and explores the trends of the DSL literature. It makes a distinction between domain-specific languages and domain-specific modelling languages (DSMLs). This is an important discussion, especially due to the modelling community's tendency to use the terms interchangeably, adding to the confusion around the topic. In the paper, DSMLs are seen as a subset of software language engineering (SLE) and remain excluded, as the paper focuses on grammar-based DSLs and DSMLs tend to be graphical. The paper also recognises SLE as a young engineering discipline, which can be one of the reasons for the lack of systematic approaches in designing and evaluating languages. Furthermore, the SMS reports a "clear lack of evaluation research" and suggests a need for more empirical evaluation in software modelling. This reinforces the difficulty of evaluating the state and extensibility of a DSL, which is the topic of this thesis.

I. Poltronieri et al.: Usability evaluation of domain-specific languages: a systematic literature review, 2017 [55]

This SLR employed the Kitchenham protocol [38] and analysed 12 papers. All the papers included some form of usability consideration and a plethora of different evaluation techniques were used, from which a taxonomy for DSL evaluation was extracted. Finally, it also found no reported major issues with usability of DSLs, on the contrary, studies

presented advantages of DSLs over GPLs. This study also paved the way for the Usa-DSL [54] framework for a systematic usability evaluation of DSLs, which will be relevant in this thesis.

J. Thanhofer-Pilisch et al.: A Systematic Mapping Study on DSL Evolution, 2017 [64]

This SMS has found 34 papers relevant to DSL evolution in 2017. It concludes that DSL evolution is a trending topic and that there is a low amount of cross-referencing in the identified literature, indicating low awareness of existing papers. DSL evolution is a central topic to this thesis, as DSL extensibility is defined as enabling the evolution of a language. The study finds plenty of different approaches for DSL evolution have been proposed, but there is no identified consensus.

A. Iung et al.: Systematic mapping study on domain-specific language development tools, 2020 [30]

This SMS covers the different DSL development tools, also known as language workbenches, discussed extensively in the homonymous section 5.2.2. These tools support graphical, textual, symbolic and tabular DSL development, with some supporting multiple representations. Another noteworthy finding is that these tools tend to be predominantly non-commercially licensed and their application domains — varied. This indicates that DSL construction is a mature research area. Finally, it points out the lack of an established mechanism to transform meta-metamodels, making the migration from one such tool to another cumbersome.

G. Czech et al.: A systematic mapping study on best practices for domain-specific modelling, 2020 [13]

Finally, this SMS covers a central topic of this thesis, which is best practices for DSL development. This includes various guidelines, patterns and design recommendations. Out of 143 studies analysed, a majority (72%) had a semi-formal style. The most cited source of the best practices was industrial projects, thus indicating a relationship between academia and industry in this field. Furthermore, they found the following success factors which were commonly identified:

- Usability (The ease of use of the language notation and associated tools and methods): 28 contributing practices
- Development cost (Reduction of overall development costs through increased automation): 16 contributing practices
- Reliability (Increased reliability through automation, e.g., generation of source code): 15 contributing practices
- Expressiveness (Expressiveness of a language w.r.t. the domain): 12 contributing practices
- Reusability (Increase reuse through DSL by reuse on the level of models): 12 contributing practices
- Learnability (Effort to learn a new language): 11 contributing practices

This indicates a consensus that usability is a pivotal success factor. This study also found no major contradiction between best practices in the 143 papers, which helps validate the practices and shows that they build off each other and have some internal consistency.

Finally, this SMS contributes to this thesis's critically important list of the most commonly cited best practices for DSL development, which can be used to evaluate the state of a DSL based on its adherence to these practices.

4.2 Domain-Specific Language Design

There is a concrete definition [22] of DSLs, as well as an understanding of the advantages and the drawbacks, that cause the hesitance of DSL adoption in the industry [66]. Regardless, there is a common sentiment in academia that "language design is largely an art, not a science" [71]. Martin Fowler also stated in 2010 that there are no clear features of a good DSL design [22]. This outlines that field experts have identified the difficulty of evaluating a language's design. Note that while development patterns exist, no heuristics have been defined that can be applied to an existing language's design.

This is an area of ongoing research [51], as programming language engineers continue to try to create actionable patterns for creation and criteria for the evaluation of languages. The early 2000s saw a formalisation of design patterns in DSLs, encapsulating common solutions to recurring problems in domain-specific language design. The document titled "Notable Design Patterns for Domain-Specific Languages" [61] from 2001 provides an early foundational framework for DSL patterns. It categorises patterns into creational, behavioural, and structural, offering a structured approach to addressing DSL design issues. The authors provide names, UML diagrams, classifications, and implementation guidelines for each pattern, aiming to foster clarity and efficiency in DSL development. These are the building blocks used to define DSL's good practices and development guidelines. They could be used directly for evaluating the extensibility of a language. Moreover, these patterns are among the first ways to conceptualise proper DSL design and modern guidelines have built upon this work.

The 2005 paper "When and How to Develop Domain-Specific Languages" [48] highlighted the nuanced needs of DSLs compared to general-purpose languages, underlining the importance of executability and the specific domains in which DSLs are used. This paper emphasises the complexity of DSL constructs and how they differ from general-purpose languages, reflecting an evolving understanding of the tailored nature of DSLs. The document indicates a growing emphasis on the practicalities and specificities of DSLs, rather than just their theoretical underpinnings. The authors define three types of patterns: decision, analysis, and design patterns. Decision patterns are used to make key choices about the DSL design and implementation, to evaluate the usefulness and put it against the upfront development cost. Analysis patterns focus on understanding the problem domain of the language, which is an essential part of DSL development. Finally, design patterns in DSL development pertain to the actual construction of the language. This includes grammar definition, parser development, compiler construction, etc. Among these patterns, the design patterns are the ones that have a use-case for this project, as the language already exists and is to be evaluated. While they cannot be applied to language development, their existence, or lack thereof, can be identified and discussed.

The "Dynamic Language Embedding With Homogeneous Tool Support" [58] paper's innovative approach to embedding DSLs within a host language through "Language Boxes" serves as a practical embodiment of language extensibility principles. This technique demonstrates a form of language extension where new functionalities are seamlessly in-

tegrated into the host language, resonating with the extensibility definitions and goals highlighted by seminal works in the field. By enabling the embedding of DSLs without altering the base language's implementation, it aligns with the modern understanding of language extension, focusing on enhancing language capabilities while ensuring compatibility with existing development tools. This method underscores the significance of extensible language design, offering a direct application of theoretical extensibility concepts to address the challenges of DSL integration in software development. While this thesis does not aim to integrate DSLs into host languages, the paper [58] provides useful terminology and insight into "sustainable" language extensions.

In 2014, "DSL Design Guidelines" [34] offered a more comprehensive and detailed set of guidelines for DSL development, reflecting the accumulated experience and knowledge in the field. The guidelines are categorised according to different aspects of language development, including purpose, realisation, content, and syntax (both abstract and concrete). This document represents a maturation of the field, providing a detailed road map for DSL designers and indicating the importance of considering a wide range of factors in DSL development. Similar to the design patterns from the paper "When and How to Develop Domain-Specific Languages" [48], these guidelines cannot be applied directly, but their presence, or lack thereof, in a DSL can act as a benchmark of the state of the language and the quality of its design.

In more recent times, fewer and fewer design guidelines are being published [13]. However, there are efforts to continue moving SLE into a proper engineering discipline. R. Gupta et al. published a paper "Towards a Systematic Engineering of Industrial Domain-Specific Languages" [26] in 2021. The paper acknowledges the scarcity of systematic methodologies in SLE and proposes a novel approach to graphical DSL development. Leveraging the power of reusability, DSL Building Blocks are used to support industrial language engineers in developing better DSLs. While this is limited to graphical DSLs, it highlights a consensus on the lack of objective and systematic methodologies in SLE.

A recent evolution in the discussion is presented in "Tara: Streamlining DSL Development through Syntactic Patterns" [51] from 2023. This paper introduces a mother language, Tara, designed to underpin DSLs, advocating for a unified approach that leverages syntactic patterns across a wide range of domains. This represents a significant evolution in DSL design thinking, aiming to simplify the development process and enhance consistency among DSLs. The framework includes a comprehensive analytical structure and compiler architecture, indicating a move towards more sophisticated, integrated systems in DSL development. While this is not directly applicable to this project, it is an insight into novel approaches to DSL design and shows that this field is still active.

All of these papers focus on DSL design and development. They show how academia has been active over decades on this topic. Due to the existence of this literature, as well as the meta-literature based on it, the research questions in this thesis can be answered and substantiated.

Chapter 5

RQ1. Extensibility Factors of DSLs

The opening research question of the thesis — "What are factors that influence the extensibility of a DSL? — paves the way for this research, by providing focus points for the latter research questions. The aim is not exhaustively listing **all** such factors, but rather provide research direction through discussion. Due to previously discussed (see Section 2.2) confusion around the term "extensibility", the fact it is also called "modifiability" and is closely linked to "maintainability", little literature exists around it, much less so in the context of SLE. Therefore, this chapter explores this topic to identify extensibility factors and justify their importance. This exploratory research question aims to knit together academic knowledge and contextualise this topic in a way conducive to answering the subsequent research questions.

5.1 Methodology

Answering RQ1 requires consulting different pieces of literature, from various fields, to find extensibility factors of software languages and software projects. The methodology consisted of the following steps:

1. Literature Review:
   A review of existing literature on DSLs was conducted. This leads to the finding of the inconsistent definition of the term "extensibility" and broadens the search to include the keywords "maintainability" and "modifiability. Consequently, the scarcity of specific literature relating these terms to DSLs leads to borrowing knowledge from the general field of SLE, and further SE.

2. Selection Criteria:
   Papers were selected based on their relevance to the general topic of this thesis — DSL extensibility. Standardisation documents and papers were consulted to define the surrounding terminology. Only peer-reviewed articles were considered to ensure the inclusion of high-quality research.

3. Data Extraction:
   From the reviewed papers, relevant data was extracted, if it was found to impact extensibility. This involved grouping similar ideas to form coherent themes, which were then analysed to create a list of relevant factors.

4. Validation Through Expert Feedback:
   The identified factors were discussed with experts in the SLE field, namely this thesis' supervisors, as well as with the developers of the industrial DSL of this case study. Through informal discussions, these factors were validated and properly formulated, as their impact was contextualised.

The literature review includes consulting the SLE field, as well as SE for maintainability standards and standardisation documents for the definitions of the terms "maintainability", "extensibility", "modifiability", etc. As discussed in a later section, DSL development is a branch of software engineering, thus best practices can be borrowed from the broader field. The proximity of the term "extensibility" to "maintainability" and its close relationship to "software evolution" in the context of software language engineering are leveraged to broaden the inclusion of relevant literature. As an exploratory RQ, much reflection is required to accompany potential identified factors.

5.2 Results

Extensibility [31], as defined in this thesis is equivalent to the modifiability aspect of maintainability defined by the ISO [28]. Thus, maintainability standards can be applied to achieve extensibility as well. Many quality attributes relate to maintainability — module coupling, component independence, unit complexity, etc. More generally, this research article about the impact of DSL tools on maintainability of language implementations [41] explores this subject by applying the principle that language implementations are software engineering projects and generalising dimensions of software system maintainability to DSL projects: Volume, Structural complexity, and Duplication.

While these factors are undoubtedly important, in this case study, there is no frame of reference to make use of them. Thus, delving into them is not fruitful for this research.

Nevertheless, this article shows that language implementations can be explored as software engineering projects and that their maintainability has been studied.

The following are identified factors in DSL extensibility:

- Implementation Volume [41] — from software system maintainability
- Implementation Structural Complexity [41] — from software system maintainability
- Implementation Duplication [41] — from software system maintainability
- **Documentation Quality** [47] — from software system maintainability
- **Regression Testing** [23] — from software system maintainability
- **Choice of Language Workbench** [13] — specific to DSL development
- **Language Grammar** [12] — specific to textual DSL development
- **DSL Usability** [13] — specific to DSL development

The factors denoted with bold text are discussed in detail below.

5.2.1 DSL Construction as Software Engineering

The notion that constructing software languages is a form of software engineering is not novel; P. Klint et al. popularised the term "grammarware" in 2005 to promote the engineering discipline of building grammar-involving software [40]. This notion has been adopted, as in 2016 M. Voelter et al., in the paper "Automated testing of DSL implementations: experiences from building mbeddr" [57] states that:

> "In a sense, they [language workbenches] move language development from the domain of computer science into the domain of software engineering." [57]

By automating the parsing, editor environment creation, and other complex tasks, DSL construction tools have solidified SLE as a form of software engineering. They have made it possible for developers without expertise in SLE to develop languages actively. Therefore, it is reasonable to expect that common best practices from SE will also apply to DSL projects, including the industrial DSL from this case study. Since the terminology is a bit inconsistent, literature regarding software extensibility is lacking. However, there is a large degree of overlap between maintainability and extensibility, with extensibility being a subset of what comprises maintainability. The aforementioned attributes such as Volume, Structural Complexity and Duplication play a significant role, but they are not useful in this case study. Ergo, the role of higher-level and more process-oriented practices — documentation and continuous integration are discussed, as they are more abstract and widely applicable than cyclomatic complexity. Namely, documentation quality and continuous integration are discussed, in the context of their importance in extensibility.

Documentation Quality

Not without its criticisms [2] [47], documentation in software development is an industry-wide adopted practice. Software becomes more usable and maintainable through different types of documentation. There are multiple dimensions to documentation and each affects extensibility in some manner.

- User Documentation:
  User documentation targets DSL end-users providing comprehensive guidance on effectively utilising the language. This can include tutorials, workshops, exercises, and reference manuals to elucidate the syntax, semantics, and usage conventions of the language. This can greatly boost its learnability and the consistency of APS specifications across different end-users.

- Developer Documentation:
  There is plenty of discussion to be had about the value of developer documentation in general. Arguments can be made that in practice, it can clutter the code readability, it fails to stay up to date with changes and does not justify its cost in development and review times. However, this is outside the scope of this thesis. As the project in this case study is developed with the Xtend [14] framework, it supports Javadoc — a widely-used code documentation generator which enables the generation of HTML pages with code documentation. It has been regarded as highly beneficial, especially in the context of public APIs [47]. If used in a project, it should be used consistently, conforming to a project-wide standard and included wherever applicable. The quality of developer documentation can affect how engineers interact with portions of the code base that are new to them. Moreover, it can make it easier for novice developers to use a new code base. The trade-off is that it increases development time, but it has an overall positive effect on extensibility

- Tooling Documentation:
  Tooling documentation focuses on the tools and utilities associated with the DSL. Namely, the language plugin for development with the DSL is essential, as its environment may require external knowledge to be utilised effectively. Similarly to user documentation, tooling documentation enables higher learnability of the end product and can — if up to date — assist end users with adopting new features, thereby positively affecting extensibility.

High-quality documentation on all levels will lead to a language that is easier to adopt, use and develop, improving the experience of both end-users and developers. Documentation extensions need to accompany all language implementation extensions correspondingly.

Continuous Integration

Continuous Integration (CI) and regression testing are established practices in modern software development, known for enhancing the extensibility and maintainability of software projects [23]. They are popular in the industry with over 93% of large enterprises involved in some form of DevOps [60]. By automating build processes, detecting regressions early, and ensuring code stability, CI and regression testing contribute significantly to the scalability and agility of software systems. Furthermore, iterative development itself is recognised as an important practice within the DSL construction community [13].

In SLE, reliability and regression monitoring are especially important. Existing versions of the language in production have instances written for them, and developers need assurance that updates do not cause regression. Employing CI tools such as Jenkins [35], as done in the development of APS, provides developers with confidence about the reliability of their projects. This means that extending the language requires less manual effort. This heavily impacts extensibility and should receive plenty of attention in SLE projects.

Co-evolution of DSL and its environment

Software co-evolution or coupled evolution refers to the phenomenon of two software artefacts needing to accommodate each other's changes when they have a dependency. Vertical DSLs are often situated within a larger system and are only responsible for a portion of its functionality, as is the case with APS. This creates a co-evolution pretext for the DSL and the wider system it inhabits, impacting the extensibility of the language. There may be vulnerable parts in the generated code, which depend on outside calls. Changes to the outer code might have a significant impact on the generator.

Conversely, it is relatively easy not to adjust parts of the generator which have dependencies of their own, but it is also a consideration that needs to be consistently accounted for. These types of considerations are common in software engineering and are just as important (if not more) in the context of DSL engineering. The impact of changes in the code generator on the interfaces enabling the generated code to interact with the entire system is vast. An analysis of this impact has to be performed and it needs to be regularly considered, whenever changes are made. A DSL should have as little dependencies on components likely to change as possible.

5.2.2 Language Workbenches (LWs)

Language workbenches are tools meant for DSL development. They are defined in detail in the Glossary and as previously discussed, their existence moves DSL construction into an engineering discipline [57]. Such a tool was indeed used in the development of APS. The LW employed is Xtext [19], which was recognised as a very mature tool by A. Iung et al. in this systematic mapping study of LWs [30]. It has existed since 2006 and has been popular in the field for almost 2 decades.

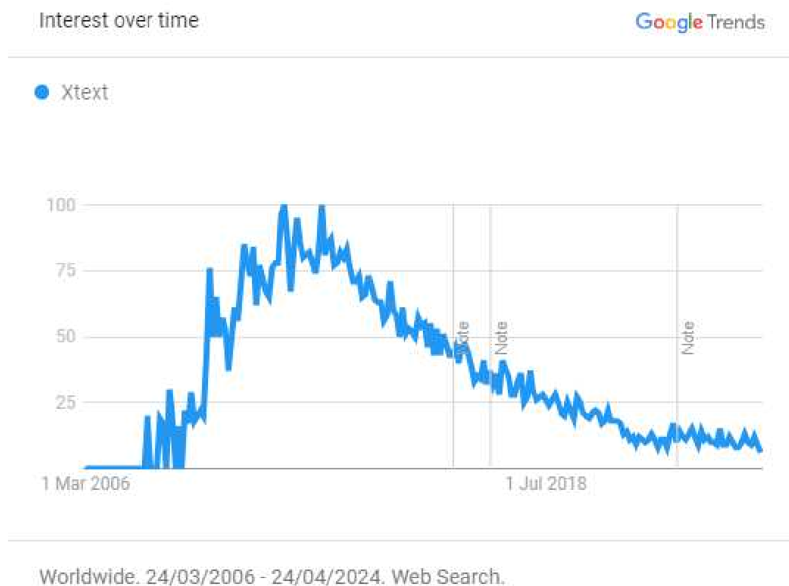

FIGURE 5.1: Daily Google searches of the term 'Xtext' 2006-2024, worldwide (Sourced from Google Trends)

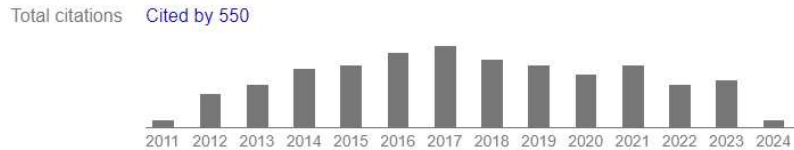

FIGURE 5.2: Yearly citations of "Xtext: implement your language faster than the quick and dirty way" [19] in April 2024
(Sourced from Google Scholar)

Impact of Xtext

Choosing a particular language workbench can affect a language over its lifetime. Firstly, different tools have varying functionality and support, as seen in the study by A. Iung et al. [30], which can make certain tools incompatible with the goals of a project, e.g., tools for graphical notation cannot be used for textual DSLs. Another consideration is the evolution of the language and whether or not the tool will be suitable for future extensions. Modularity and reusability are the focus of all LWs. Nonetheless, some lack certain out-of-the-box functionality, which may arise as a requirement as the language evolves. Thus, great care should be put into choosing the correct language construction tool, as recognised in the systematic mapping study by G. Czech et al. on the best practices in domain-specific modelling [13].

Xtext supports 32 of the 34 features defined for language workbenches [30]. It is based on the Eclipse Modelling Framework [63], has a non-commercial license, and has received significant attention in the field, both by researchers and industry [30]. However, the DSL scene is evolving and potentially shifting away from Xtext. The overall interest in the framework is dwindling as seen in Figure 5.1. Note that this does not necessarily represent a relevant decline, as Xtext continues to receive academic interest (Figure 5.2). Still, the development team of Xtext itself has recognised this as an existential threat to the framework and created a call to action [1] on Xtext's GitHub development repository. Relying on open-source software always carries an underlying risk of the project losing steam and going out of support. This can have serious consequences, as updating the underlying development environment becomes more costly and these updates can be critically important. Moreover, there is an LW lock-in, due to the lack of an established way to transform meta-metamodels [30]. There has been some research in model interoperability [37] in the form of M3 bridges. M3 refers to the "meta-meta" level of modelling and proposes a systematic way to migrate from one meta-modelling tool to another. This is however largely incomplete, as it is not automated and still requires expertise and resources to perform, making it economically challenging in an industrial context.

Notwithstanding the current development state of Xtext itself, this section explores its impact on the development and state of APS. The tool provides the DSL with a reasonably user-friendly interface for defining APS files. This is done largely via the Xtext-enforced BNF [42]-derivative grammar, modified to capture semantics and through the interfaces generated by Xtext for defining language server [49] features. It also comes in a package with the Xtend [14] framework — a dialect of Java, itself developed with Xtext, specially tailored for code generation, employing template expressions to support templated code generation [22]. Xtext and Xtend are considered parts of a whole package, which together make up the LW and referrals to "Xtext" in this thesis are used to refer to the entire package. Using Xtext forces developers into patterns and provides a structured way to approach the task of parsing and code generation. This results in structured and main-

tainable code, a strong positive effect of having employed the LW (refer to Section 7.2, to see how many patterns Xtext has contributed to).

Furthermore, the tool support Xtext provides out-of-the-box such as syntax highlighting, code folding and auto-completion have a direct positive impact on the usability of the language. These features are readily available to use inside of the Eclipse IDE, the Xtext environment. Still, they are compatible with other code editors because they are implemented largely through the Language Server Protocol (LSP) [49], which means that the tool support can be extended to other code editors that support LSP with relative ease.

Finally, engineers who have worked on APS have identified Xtext and Xtend as useful (see Chapter 6), specifically on account of reducing the necessary effort to achieve the necessary goals. The language developers expressed gladness that it was not necessary to build the language from scratch, as that would require a lot of extra work. Even if this benefit is only perceived, that is still a substantial benefit, as the perception of the development cost of DSLs remains a major hurdle in their adoption in the industry [66]. Furthermore, most of the engineers working on APS do not have any background in SLE — further solidifying the concept that LWs have shifted language construction to the domain of SE.

Co-evolution of DSL and its LW

As previously mentioned, co-evolution is observed when two pieces of software have a dependency, causing a requirement for change in one, when the other undergoes evolution. In this context, a DSL and a LW need to co-evolve as the DSL exists within the environment of the LW. As previously discussed, using a particular LW often means locking the language into its environment, meaning co-evolution is particularly relevant for the maintainability of a specific language.

Moreover, DSLs — especially vertical, embedded ones like APS — rarely exist in a vacuum. They play a role in a broader system, which is itself a software artefact that undergoes evolution. The DSL has an impact on the system it is a part of and vice versa. This is an area that has already received some attention in the recent 2023 paper "Evaluating Tool Support for Co-Evolution of Modeling Languages, Tools and Models" [65]. It presents a framework for evaluating an LW's capabilities for the co-evolution of graphical modelling languages. This shows that not only do LW have an impact, but also that different ones have varying effects.

5.2.3 Software Language Grammars

A final aspect to naturally touch on when discussing the ability to extend a textual language is its grammar. Textual languages, such as APS, generally have a grammar defined to describe their syntax. Introducing new syntax, therefore, requires modifying the grammar of the language to incorporate it. Depending on the extension, this can lead to grammar ambiguities — cases where the parser has more than one option to parse a given text. This can lead to unpredictable and unwanted behaviour and slow down development.

To address this, academia has proposed numerous extensible, composable, modular, and modifiable grammars [12] [20] [32] [67]. This is a testament to the importance of grammars in language extensibility. Another way to handle this ambiguity is to do away with the grammar altogether — as is done in the language workbench MPS [11], which uses a so-called Projectional Editor, instead of a formal grammar. This allows for adding new syntax without the risk of introducing ambiguities. Still, Xtext uses a classic BNF-

style grammar and this grammar has an impact on the language. Migrating away from it is economically unfeasible, thus considerations have to be made regarding the grammar whenever syntax extensions are made. Grammar ambiguities need to be considered and prevented during the planning phases of extension development. This is something that the extension guidelines in RQ4 need to consider.

5.2.4 DSL Usability

Usability, as defined by the International Organisation for Standardisation (ISO) [29], is the "extent to which a system, product or service can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use". The usability of DSLs is paramount for their successful adoption [13] [55]. Thus, it has to be a key factor to consider when trying to extend a language, as extensions tend to impact usability.

This systematic literature review by Ankica Barisic et al. from 2015 [7] has highlighted the importance of evaluating the usability of DSLs, suggesting that a systematic approach must be developed and employed. The SLR emphasises the importance of involving end-users in evaluating DSLs for usability and quality in use. However, it notes a general lack of systematic approaches, guidelines, and comprehensive tool sets for DSL evaluation, indicating a gap in the current (at the time of the paper's writing) practice. It is important to note that this review only focused on visual domain-specific languages, however, there is no reason to believe that graphical DSLs differ in the importance of usability.

As a first step in addressing this gap of usability evaluation of DSLs the Usa-DSL framework [54] was created. It comprises four phases: Planning, Execution, Analysis, and Results (PEAR). Each phase consists of specific steps, such as defining evaluator profiles, ethical and legal responsibilities, data types, empirical study methods, and evaluation metrics. In summary, the Usa-DSL framework marks a beginning in the systematic evaluation of DSLs, integrating diverse methodologies and insights from previous studies.

Usability plays a pivotal role in the maintainability of a DSL [4], especially the adoptability. End-user demands can dictate extensions to the language and if usability is not considered throughout the development cycle, it can increase the amount of work required for extending the language. This is why usability in DSLs is still a trending topic in academia [53].

Like other aspects discussed in this chapter, this is deeply relevant to vertical DSLs, especially if the end users are not software developers. The usability of a DSL artefact includes the ease with which it can be understood, edited and interacted with. To achieve its goal, a DSL needs to have a tangible productivity boost and usability is how this can be achieved. Ergo, these considerations need to be made a part of the development cycle and make their way into DSL extension guidelines.

5.3 Discussion

This chapter has highlighted areas important in DSL construction regarding extensibility. Some factors are derived from the general maintainability standards for software engineering. Others are unique to DSL development. Some of the factors were discussed more in-depth, due to their relevance to the case study in this thesis. The subsequent research questions make use of these insights, by focusing on the effects of the discussed factors. Evolution in DSLs is very common in practice with 86% of surveyed DSL practitioners reporting at least one case of evolution in a recent survey of established DSL practices [8].

There is, however, a noteworthy sparseness of literature on this topic. A systematic mapping study on the subject of DSL evolution in 2017 [64] found only 34 papers relating to DSL evolution. Furthermore, it found a low amount of cross-citations within those 34 papers. This can be speculatively attributed to the inconsistency of the language used in the field. The evolution of the language, e.g., the repeated extensions to the language, is enabled by a maintainable language implementation. This can be viewed as the extensibility of the language under the classical definition of the term in the context of software engineering [31]. However, in the DSL domain, extensibility carries often a different meaning, relating to languages with a focus on the user's ability to extend the language. Moreover, the confusion around DSL and DSML further causes difficulty in finding relevant papers [43]. The SMS on DSL evolution concludes that general software evolution practices may be worth examining in the context of DSL, corroborating the discussion on SLE as SE.

Language workbenches successfully achieve their task of making DSL development easier and the resulting language implementations more maintainable [41]. It has been shown to successfully aid the transition of SLE into a proper engineering discipline and they've enabled language development to become a branch of the broader field of software engineering. This means that LWs have had a profound impact on DSLs and their extensibility. Both directly, as they make languages a subject for co-evolution and indirectly, by enabling language development in an industrial context. This motivates the subsequent research to assess the impact Xtext has had on APS, to help validate the identified research.

The identified extensibility factors mark a new development in the field, as they are the first (to the knowledge of the author) such collection to date. Furthermore, in the context of this thesis, they enable a more comprehensive approach to answering the subsequent RQs. Firstly, the questions for the interview performed for RQ2 (see B.2) to obtain the developers' perspectives on these topics were influenced by these factors. Secondly, guidelines identified in RQ3.1, consider these factors and their impact on relevancy. Lastly, the extension guidelines — the culmination of this thesis — take these factors into account, to ensure that extensions do not negatively impact them.

Chapter 6

RQ2. Current Challenges in Extending APS

In this chapter, the second research question will be addressed — "What are the current challenges in extending APS?". That includes the methodology undertaken, as well as the obtained results and their relation to the rest of this research. The results are described in detail, in this chapter, while a summary can be found in Appendix B.1. These results guide the succeeding questions by providing a research focus.

6.1 Methodology

The APS developers were directly queried to understand the challenges in the iterative development of the DSL. Seven domain expert interviews were performed, as a reliable way of obtaining relevant information, especially in this context, where the research question pertains, at least partially, to the subjective experience of the developers. To ensure the respect, privacy and safety of all participants in the research, before each interview, the interviewee was informed about the purposes of the interview, the way their data will be handled and their consent was obtained formally via the ethics-board-approved consent brochure, attached in Appendix B.1. An ethics proposal for conducting the interviews was submitted to the University of Twente's Ethics Board and received approval thereafter. The interview was conducted under a semi-structured format and the questions asked can be found in Appendix B.2. These questions have been chosen specifically to fit the scope of this research, and thus have not been borrowed from other papers. They focus on answering RQ2 by understanding where the main development effort is spent, what the priorities are and what issues have occurred hitherto. However, several questions also serve to simply contextualise the project and assist the other research questions — by querying about the impact of the meta tooling employed, as well as the impact of regression testing — a common software engineering practice. They have been added, due to them being identified as important factors in extensibility in RQ1.

6.2 Results

6.2.1 The Challenges in the Development of APS

A table with the aggregation of interview answers can be found in Appendix B.3. After conducting seven interviews with the developers important insights were obtained into the challenges associated with the DSL development, as well as into the positive elements of their development process. Since APS is a relatively large and mature project, having undergone five years of iterative development, the complexity of the project is high and thus the answers were varied. Nonetheless, there was a level of internal consistency, which is a sign of the alignment of the goals of all the interviewed developers.

Four out of seven interviewees expressed that they spend more time reading specifications than writing them, with two spending more time writing and one seeing them as equally time-consuming. Nevertheless, each participant claimed to value readability in the language highly.

Furthermore, while the front end and back end both receive attention during extensions, the main development efforts, everyone agrees, are spent on the back end. Another area in which all developers arrived at the same answers was the focus on extensions — always feature-driven. The main focus of additions to the language is to include new functionality, introducing keywords and adding use cases. An emphasis is placed on value for money, meaning that each extension must justify the development efforts with productivity boosts. This means that to undertake an extension, the extension has to be seen as value-adding from an economic standpoint

The APS developers also all agreed on the usefulness of regression testing and continuous integration, reinforcing that classic software engineering best practices are also applicable to modern software language engineering projects. The frameworks used in the project, Xtext and Xtend, received significant praise from each of the interviewees. The economic viability of the project without such a tool was rated dubious at best and completely infeasible at worst.

The last unanimous consensus amongst the developers was the lack of a formal feedback loop mechanism. Everyone recognised defining a formal process for obtaining, processing and integrating user feedback into the development cycle is imperative. This is a high priority, as it should exist before the language roll-out to external users.

There were, however, some discrepancies in the answers. There were no contradictions per se, but rather different perspectives on some topics, stemming from the different roles and responsibilities of the interviewees. On the topic of usability, everyone considered it a consideration in the development cycle, with three people expressing it as an "indirect" consideration. Nevertheless, every respondent had a different interpretation of how these considerations of usability manifested in the development of the language. Notwithstanding the differences in the answers, all participants mentioned that great care is put into choosing keywords. The domain space of the end-users is considered and great attention is put into avoiding using overly specific constructs, which does indeed display a concern for usability. Regardless, there were no mentions of systematic measurements of the usability of the language and all considerations made concerning usability are intuition-based.

Interviewees were also not unanimous in their views of failed past extensions. This is again attributed to their difference in experience and roles. Two developers believed that due to the strong software impact analysis performed prior to development, extensions never "miss the mark". However, more veteran developers had examples of needing to remove keywords, change where they are placed within the language, and shift from implicit behaviours to explicit specifications. Structural challenges arise in the development process, often due to a lack of specific knowledge. An example named by three interviewees was the introduction of legacy support. Due to the criticality of the project's reliability and backward/forward compatibility between the different versions of the language and the different machines, a lot of legacy considerations need to be made.

As a consequence of the project's size and the domain's particular nature, challenges were also named by two developers regarding maintaining consistency across the language with the numerous extensions, which are sometimes developed in parallel. This is also manifested in extensions which are developed from the engineer's perspective with highly specialised knowledge of the back end and the underlying processes. These concerns relate to the language's usability and learnability and thus should be addressed systematically.

Overall, the interviews gave an insight into the specific challenges encountered as well as the inner workings of the company structure and how it contributes to the development process. The insights highlight both the positive current practices that must be maintained as well as the potential pitfalls and missing elements.

6.3 Discussion

The results show that the development effort is focused on the back end. That is both a testament to Xtext's efficiency in the definition of a front end, as well as to the design of the front end, which enables them to insert keywords and constructs easily whenever necessary. This indicates that the extensibility of the grammar, e.g. the front end, is in a good state and that extension guidelines should aim to maintain the current structure of the front end and improve the back end's modifiability.

Furthermore, it becomes clear, both from past and future extension focus, that new functionality is the top priority. Value-for-money is a unique consideration to make, only relevant in the industrial context. It shows that there is a particular economic challenge in the development of the language — chosen extensions must be directly economically viable and justifiable before implementation. This means that in the industry, extension

guidelines cannot merely focus on some internal measurement of quality; extensions also need to be "useful" from a corporate standpoint.

The different answers on the topic of usability can be attributed to the broad definition of usability and people's interpretation of it. However, people with more managerial roles tend to believe an emphasis **is** placed on usability, whereas people with more hands-on development experience seem to believe it is indirect. This is a common separator of opinions in this study and it exemplifies that the proper processes are followed and people perform their respective roles. This is further reinforced when you look at the difference in answers concerning the prominence of technical and structural challenges. This is another example of the structure within the development group working as intended, with people having different focuses and views on the overall development.

The interviews showed that, while a consideration, usability is not measured systematically. This is not necessarily an incorrect approach, but it does indicate that developer bias can negatively impact usability. This was also recognised by several of the interviewees. Thus, maintaining objectivity in this context is also challenging in the development process.

As for the named challenges — legacy features seemed particularly trouble-inducing, due to a common challenge surrounding legacy — a lack of understanding of the current employees. This further reinforces that modern compilers can be treated as software projects when discussing maintainability.

Overall, these interviews have shed light on the practical challenges encountered in the development of a DSL in the industry. Moreover, they have also highlighted the aspects of the projects that are going smoothly and should be maintained. They have reinforced key points in this research, such as highlighting the importance of language workbenches and also indicating that DSL construction projects benefit from classic software engineering best practices, such as regression testing and continuous integration. As pointed out by the developers themselves, their bias with respect to usability needs to be addressed by relying on objective measurements of these qualities.

Although this research question focuses on the challenges, some positive practices were observed from the interviewees' answers. The good practices indicate that general software evolution techniques successfully apply to DSL development as well. This answers a question proposed by an SMS on DSL evolution [64]. These practices were identified as beneficial to the development of the DSL and should be preserved:

- The proper distribution of different responsibilities amongst the different roles of the developers
- The usage of a meta-modelling tool, namely Xtext
  - This has partly contributed to the good, easily extendable Xtext grammar
- The emphasis placed on the readability of the specifications
- The rigorous impact analysis performed before the implementation of extensions
- The beneficial regression test suite

As for the challenges in the development of APS, these were identified from the interviews:

- Implementation-related challenges:
  - Implementing legacy extensions

  - Performing restructurings of the language
- Usability-related challenges:
  - Maintaining objectivity when evaluating usability as developers
  - Preserving internal language consistency across numerous extensions and use cases
  - Ensuring decoupling between the specialised knowledge of the language developers and the language constructs themselves
    * Choosing the correct keywords with considerations for the domain of the end-users
- General challenges:
  - Ensuring extensions are future-proof
  - The lack of a formal mechanism for obtaining, analysing, and integrating feedback into the development process
  - Ensuring the economic viability of an extension

Answering the second research question of this thesis is important because it positions the researcher to procure relevant guidelines for the DSL. Thus, the following work in this thesis focuses on these challenges. This increases both the internal and the external validity of the thesis. The internal validity — via ensuring that the guidelines in the answer to the final research question apply to the language. And the external — by indicating the circumstances under which the guidelines are pertinent. They may be less relevant to the extension of a DSL with different obstacles in its development.

Chapter 7

RQ3. Extraction and Application of Modern DSL Design Guidelines from Academia

This chapter delves into the best practices as recognised by academia in DSL development and aims to translate theoretical DSL design insights into a tailored set of guidelines for APS. The goal is to answer RQ3 "How can existing language design and extensibility guidelines be used to effectively evaluate the state of a DSL?" This is done by extracting a set of relevant design guidelines, extensibility standards, and evaluation methods from the literature and scrutinising APS against them. There is a significant amount of literature on the topic, as explored in Chapter 4, however, a lot of it may not be relevant to a given DSL.

7.1 Methodology

Due to the wide scope of this research question, it was split up into four sub-questions. The first three sub-questions are explanatory research questions, as they entail a **literature review** to synthesise a summary of the existing body of knowledge. The final sub-question is exploratory and it involves using the answers from the other ones to propose a novel, curated method of DSL evaluation.

RQ3.1: Are there established, systematic ways to evaluate a DSL holistically?

Despite being a yes-or-no question, this research sub-question prompts a non-trivial probe into the academic literature. This topic, the evaluation of DSLs, has been touched upon in the Related Literature Chapter 4. The term "holistically" in the question refers to a complete evaluation of all the language's attributes, not just aspects of it. As mentioned before, usability evaluation within the DSL community is a trending topic [53] as of 2021, but that is not a holistic approach to DSL assessment. It has also been discussed in Chapter 4 that software language engineers did not report evaluations of their languages as of 2011 [25]. Moreover, while ad-hoc evaluations have reported productivity boosts, there is still no external validity to such claims, due to the lack of systematic measurements [6]. Yet still, an SMS of DSLs in 2016 [43] cites a "clear lack of evaluation research" in the DSL field. There is a lack of more modern meta-literature on the topic to lean on, thus an exhaustive analysis has to be performed to answer RQ3.1. The following query was executed in the Google Scholar search engine:

> allintitle: (evaluation OR assessment) "domain-specific language" OR "domain-specific languages" -usability

It specifically seeks these terms in the paper's title, as otherwise, too many false positive results are returned, because "assessment" and "evaluation" are ubiquitous words in literature. There is a need to limit the results so that they can feasibly be exhaustively analysed. Furthermore, "usability" was excluded from search results to find holistic evaluation frameworks. Lastly, the time frame of the query was set to be the last decade, to obtain modern, relevant results. With all these constraints, the query returns 24 results (as of early 2024). After removing duplicates, the number becomes 15. Of these, five are false positives not concerned with DSL evaluation, but rather employ DSLs to evaluate something else. The remaining 10 contain six case studies, which do not report a method for general DSL evaluation, and hence are excluded. The last four are discussed in the results section of this chapter.

RQ3.2: What are the existing domain-specific language design guidelines?

Although DSL design has been categorised as an art, rather than a science in the past [50], there have been strides towards moving the field into an engineering discipline [39] [40]. Systematic methods need to be defined to shift from the domain of art to science, therefore plenty of objective guidelines have been devised for domain-specific language design in the past 30 years. To identify them, the following search query was employed in the Google Scholar search engine:

> ("domain-specific" OR "domain specific" OR "DSL" OR "DSLs") AND ("language" OR "languages" OR "modelling") AND ("design patterns" OR "best practices" OR "guidelines")

The query was further tightened to only retrieve papers from the year 2000 to the present day (early 2024). With this constraint, the query yielded 18,200 results. A large portion of the surveyed ones were irrelevant, but performing an exhaustive, systematic literature review on this scale is outside of the scope of this thesis. From the first 20 matches, the papers with a direct focus (expressed in the title or abstract) on design guidelines, patterns or best practices were selected [24] [34] [36] [48] [61] [68]. Given their volume, they provide a sufficient library of best practices to choose from for this sub-question. While search engines are imperfect and miss relevant papers, the results were enriched via "snowballing" — selecting relevant papers and then, for each one:

- Scanning the references bibliography
- Scanning papers which have used it as a reference

This is the methodology employed in discovering the existing DSL design guidelines.

RQ3.3: What are the existing software language extensibility guidelines?

The answer to the antecedent sub-question alludes to some sort of extensibility considerations in academia ('Design for language evolution' [13]); regardless, none of the guidelines instructs **how** to achieve this evolution-enabling design. Therefore, the literature has been probed to look for an answer. The following search query was employed in the Google Scholar search engine, the results of which are discussed in the dedicated section:

> ("domain-specific" OR "software" OR "domain specific" OR "DSL" OR "DSLs") AND ("language" OR "languages" OR "modelling") AND ("extensibility standards" OR "extensibility guidelines" OR "evolution guidelines" OR "extensibility-centered design" OR "evolution-centered design")

RQ3.4: How can the answers of RQ3.1, RQ3.2, and RQ3.3 be leveraged to evaluate the state of a DSL?

Using the results from the previous sub-questions, RQ3.4 is answered empirically, by evaluating the industrial DSL from this case study. This evaluation involves careful examination of the language implementation, leaning on discussions with the developers themselves. The evaluation is systematic and objective and serves as an answer for the overarching RQ3 — "How can existing language design and extensibility guidelines be used to effectively evaluate the state of a DSL?"

7.2 Results

RQ3.1: Are there established, systematic ways to evaluate a DSL holistically?

The four papers identified to potentially answer the RQ are the following:

- Domain-specific language domain analysis and evaluation: a systematic literature review [7] — previously explored in the Related Literature Chapter 4, considers only visual DSLs and does not propose a systematic way of DSL evaluation
- A framework for qualitative assessment of domain-specific languages [33] — a viable candidate for a systematic framework for DSL evaluation, which will be discussed in-depth below.

- AgentDSM-Eval: A framework for the evaluation of domain-specific modeling languages for multi-agent systems [3] — focuses on a narrow scope and thus does not apply to this research sub-question.
- Heuristic evaluation checklist for domain-specific languages [59] — concerns usability evaluation only, despite not being mentioned in the title, therefore it is also not applicable

From the above findings, a single paper was found to satisfy the requirements for a **holistic** evaluation method — the framework for qualitative assessment of DSLs [33] (FQAD). The framework describes the quality characteristics of a DSL, with accompanying sub-characteristics that should each be evaluated to obtain a comprehensive evaluation. The paper instructs the evaluator to prioritise the quality characteristics most important to them if they need to narrow the scope. However, the paper provides no information on how the evaluation should be performed and is ultimately left to the performer. The authors acknowledge a valid criticism of the framework — that it constitutes late feedback. As discovered in RQ2, economic viability is a consideration, especially for industrial DSLs. From this perspective, investing in ways to guide a software language engineer in the decision-making process pre-development is far more sensible. This is also reflected in the large volume of guidelines identified in academia, which specifically address the need for careful analysis and decision-making prior to implementation, as opposed to the scarce literature on DSL evaluation.

In conclusion, while the FQAD paper makes a significant step in the right direction by defining the relevant quality attributes for a DSL, the answer to the sub-question is no. At least for inexperienced evaluators, as FQAD leans on the experience of the evaluator to have them make the decision. None of the papers identified describe a systematic, objective way of evaluating a domain-specific language. This suggests that finding and using the state-of-the-art DSL guidelines for a language evaluation is a valid approach, due to the lack of other established, objective assessment methods.

RQ3.2: What are the existing domain-specific language design guidelines?

Via the employed methodology, a particularly relevant study on the best practices in domain-specific modelling [13] was found. It is a systematic mapping study carried out on this precise topic in 2020. This is ideal for this sub-question because it already includes all the hitherto identified papers, out of a total of 143 studies. This SMS itself answers this sub-question by providing the 10 most cited guidelines, which is a great starting point for a list of guidelines. Some literature has come out since on the matter (i.e., [27] [69]), but none that pushes forward novel design guidelines. Consequently, this sub-question leans heavily on this SMS, which notes that the majority (70%) of guidelines were defined before 2011, exemplifying a slowdown in the field. It compiles a list of the ten most cited good practices for DSL development. These are the consensus in the field of DSL construction, so adherence to them indicates a language's quality. Some best practices listed are generic and inapplicable post or during development. This reflects the overall state of the literature, where a significant portion of the guidelines are general considerations to make before development, rather than practical advice. This is expected — after all, they are written for prospective DSL engineers and are not intended as an evaluation benchmark.

Ultimately, this sub-question aims to condense a list of the most relevant DSL guidelines. While the ten found in the SMS are a good starting point, they can be expanded, especially with more specific development patterns, to ensure that the final list of guidelines is relevant in an industrial context. There is a considerable amount of papers with

a set of guidelines which were reviewed [22] [24] [34] [36] [48] [61] [68] [70], which have considerable overlap and were already included in the SMS. To supplement the first ten most-cited guidelines practical patterns need to be added. Martin Fowler's pivotal work on DSLs [22] has been chosen for its uniquely specific implementation patterns. None of them are in the most cited ten best DSL practices, reinforcing their uniqueness. Checking if a DSL employs these patterns is insightful as some are mutually exclusive, such as model aware and model ignorant code generation. Thus no DSL exists that does not employ any of them. The book includes 46 DSL patterns, which provide more practical insight into how DSLs should be developed. 15 of them are meant for internal DSLs and consequently are excluded. Seven are also concerned with the parser and lexer development, thus are likewise omitted, as APS is developed with the Xtext [19] tool.

Lastly, a set of applicable and specific best practices was included, emanating from the paper by G. Karsai et al. on design guidelines for DSLs [34]. It is chosen over others because its guidelines are exhaustive, it directly cites other high-quality papers from the list and provides examples of where and how these guidelines can be applied. Moreover, unlike others on the list [68] [70], it is not based solely on the author's experience, so it complements the patterns of Fowler nicely, ensuring the final list is representative of the DSL community, rather than a couple of persons. Five out of the ten most cited practices appear in the 26 provided by this paper.

The guidelines are compiled from the top ten most cited best practices from the SMS by G. Czech et al. [13], Martin Fowler's "Domain-Specific Languages" [22], and the design guidelines from G. Karsai et al. [13] The following are **omitted**, grouped by the reason for omission, along with the list they are from:

- Omitted, due to targeting internal DSLs:
  → From Martin Fowler's [22] patterns:
  - Expression Builder, Function Sequence, Nested Function, Method Chaining, Object Scoping, Closure, Nested Closure, Literal List, Literal Map, Dynamic Reception, Annotation, Class Symbol Table, Parse Tree Manipulation, Textual Polishing, Literal Extension
- Omitted, due to targeting parsing/lexing techniques, which are handled by Xtext:
  → From Martin Fowler's [22] patterns:
  - Recursive Descent Parser, Parser Combinator, Delimiter-Directed Translation, Syntax-Directed Translation, Embedded Interpretation, Alternative Tokenisation, Context Variable, Symbol Table, Nested Operator Expression
- Omitted, due to referring to the planning phase of DSL development:
  → From G. Karsai et al.'s design guidelines [34]:
  - Identify Language Uses Early, Ask Questions, Make your language consistent, Decide carefully whether to use graphical or textual realisation, Compose existing languages where possible, Reuse existing language definitions, Reuse existing type systems

  → From the SMS by G. Czech et al. [13]:
  - Carefully choose the form of notation
- Omitted, due to APS developers' choice regarding relevancy:
  → From the SMS on the best practices in domain-specific modelling [13]:

  – Reuse language definitions, Balance genericity and specialisation, Design must have a purpose

→ From Martin Fowler's [22] patterns:

  – Decision Table, State Machine, Production Rule System, Macro

→ From G. Karsai et al.'s design guidelines [34]:

  – Keep it simple, Use syntactic sugar appropriately, Use descriptive notations

This results in the following answer to RQ3.2 — design guidelines, curated for APS, grouped into:

- Language Layout — Guidelines concerned with the vocabulary, syntax and layout of the DSL instances

  1. Adopt existing domain notations [13] [34]
  2. Foreign Code [22] — Embed some foreign code into an external DSL to provide more elaborate behaviour than can be specified in the DSL
  3. Syntactic Indentation [22] — Recognising the level of indentation as a part of the syntax
  4. Adaptive Model [22] — Arrange blocks of code in a way that captures data structure to implement an alternative (non-imperative) computational model
  5. Introduce interfaces [34]
  6. Prefer layout which does not affect translation from concrete to abstract syntax [34]
  7. Align abstract and concrete syntax [34]

- Development Process — Meta guidelines which direct how the development should be carried out

  8. Importance of meta-tooling [13]
  9. Iterative development [13]
  10. Domain engineering team [13]

- Design — Guidelines regarding the design of the Language itself from a high level

  11. Design for language evolution [13]
  12. Use the same style everywhere [34]
  13. Identify usage conventions [34]
  14. Balance compactness and comprehensibility [34]
  15. Provide organisational structures for models [34]
  16. Avoid conceptual redundancy [34]
  17. Avoid inefficient language elements [34]
  18. Enable modularity [34]

- Back end — Guidelines centred around code generation

19. Transformer Generation [22] — Generate code by writing a transformer that navigates the input model and produces output
20. Templated Generation [22] — Generate output by handwriting an output file and placing template callouts to generate variable portions
21. Embedment Helper [22] — minimise embedded code in templating systems by providing the necessary functionality as helper functions, which are invoked in the templates
22. Model-Aware Generation [22] — Generate code with an explicit simulacrum of the semantic model of the DSL, meaning that in the target environment, scaffolding exists to support the generation of more understandable code
23. Model Ignorant Generation [22] — Hardcode all logic into the generated code so that there is an explicit representation of the Semantic Model
24. Generation Gap [22] — Separate generated code from non-generated code by inheritance

- Front end — Guidelines for the translation of DSL instances into abstract syntax trees

25. BNF [22] — Formally define the grammar of your language in Backus-Naur Form
26. Parser Generator [22] — Build a parser driven by a grammar file
27. Tree Construction [22] — The parser creates and returns a syntax tree representation of the source text that is manipulated later by tree-walking code
28. Notification [22] — Collects errors and other messages to report back to the caller

- User-centric — Guidelines that primarily aim to enhance the end-user experience of the DSL

29. Importance of DSL tooling [13]
30. Limit the number of language elements [34]
31. Make elements distinguishable [34]
32. Permit comments [34]

RQ3.3: What are the existing software language extensibility guidelines?

The search employed regrettably only returns 125 results, none of which are relevant, notwithstanding efforts to expand the breadth of the query by including plenty of logical disjunctions, including the general term "software" to find literature about "software languages". This shows that there is little in the way of explicit extensibility standards for software languages, let alone for DSLs, not to mention vertical, embedded DSLs such as APS. This indicates that extensibility in DSLs is an area that lacks research and justifies the contributions of this thesis as a viable contribution to the field.

As discussed in the Related Literature Chapter 4, an SMS has been performed on the topic of DSL evolution. It has found 34 relevant papers — a relatively scarce library to begin with. Furthermore, these papers are classified into the following categories:

- DSL creation [64]

- DSL grammar [64]
- Challenges in DSL evolution [64]
- Co-evolution of domain vs. DSL [64]
- Tool for DSL evolution [64]
- General approach for DSL evolution [64]
- Practical example of DSL evolution [64]
- References to GPLs [64]

Thus, this SMS has also not found papers which contain guidelines or design patterns to enable DSL evolution. The category "General approach for DSL evolution" could potentially contain such papers, but there are none. This corroborates the findings from the earlier query and indicates that the central topic and final research question of this thesis — extensibility guidelines for DSLs — is a novel contribution to the field of DSL engineering.

RQ3.4: How can the answers of RQ3.1, RQ3.2, and RQ3.3 be leveraged to evaluate the state of a DSL?

The answers to the previous sub-questions show that despite DSL construction being a mature research field, evaluation research is lacking (see RQ3.1, RQ3.3). Nonetheless, the value of the ability to evaluate the state of a DSL is undisputed. Since guidelines favour the iterative development of languages, having an objective measure of DSL quality would be hugely beneficial for regression monitoring. However, the challenge here requires more than mere effort, as it is conceptual. Consequently, such an objective manner of language assessment must be established. To evaluate the state of a DSL, one can examine the language's adherence to a set of relevant guidelines (see RQ3.2). The higher the adherence, the better that reflects on the DSL. Lack of adherence to guidelines prompts a need for investigation, such that gaps and areas of improvement can be identified. To show this, the procured guidelines from RQ3.2 will be used on APS for this exact purpose. In Table 7.1 APS is scrutinised under the identified guidelines. They are colour-coded, with blue indicating the guideline was followed, red indicating it was not and yellow for partially applied guidelines. Justifications are provided for the decision, sometimes accompanied by a quick reflection on why the guideline was (not) followed.

3/7	Language Layout	
A	Adopt Existing Domain Notations	The developers are closely linked with the domain itself. Moreover, the interviews from RQ2 show great care is put into choosing correct keywords, respecting the domain of the end-users.
A	Prefer Layout without Impact on Parsing	The APS parser gives no significance to whitespace characters, which means that language instances have no enforced consistent formatting.
A*	Align Abstract and Concrete Syntax	The AST does not undergo transformations. This is partly because of Xtext binding syntax and semantics on the grammar level.
NA	Foreign Code	APS does not allow any embedding of foreign code. This is, however, something that can be beneficial and reduce the amount of work required after code generation.
NA	Syntactic Indentation	As this contradicts the guideline for a layout without an impact on parsing, the same justification applies here.
NA	Adaptive Model	Due to the language's nature, no computational models are necessary in its context.
NA	Introduce Interfaces	There is no necessity for paraphrasing features in APS, because it is a specification language.
2.5/3	**Development Process**	
A*	Importance of Meta-Tooling	The language uses a mature LW that is suitable for the use case.
A	Iterative Development	The company employs a scrum methodology for its projects, including APS.
PA	Domain Engineering Team	Despite all manner of engineers working on the language, they all come from the same team within the company. They do, however, actively consider the perspective of the domain of the end-users
3/8	**Design**	
A	Avoid conceptual redundancy	The language of APS does not provide multiple options for the same use case.
PA	Use the Same Style Everywhere	This is intended by the language designers, however the developer interviews revealed that it has been violated when different parts of the language used different syntax for boolean operators. This can be addressed by extracting grammar commonalities, rather than reliance on ad-hoc intervention.
PA	Design for Language Evolution	Both its presence in APS and lack thereof is evident in the project proposal for this thesis.
PA	Balance compactness and comprehensibility	A difficult-to-quantify guideline, due to its genericity. However, the interviews performed for RQ2 have shown that the language designers have put consideration into both aspects, seeing them as complementary attributes.
PA	Enable modularity	In APS, complex specifications can be broken down into multiple files, as APS allows for cross-file references, there is no explicit import mechanism, which limits the modularity.
NA	Identify Usage Conventions	Due to very limited user base, comprised only of the language developers, there are no standard usage conventions defined. Employing them could be useful for distribution to end-users.
NA	Provide organisational structures for models	Multi-file specifications are possible in APS, but there is no explicit 'import' keyword or any equivalent, therefore also no mechanism to refer to other directories.
NA	Avoid inefficient language elements	The performance of the generated code has never been expressed as a focus of the development team. This does not imply that the generated code is inefficient, but there no visible optimisation yet.
4/6	**Back end**	
A	Embedment Helper	To aid the readability and simplicity of the template expressions in Xtend, helper functions are defined and called from within the templates.
A*	Templated Generation	Enforced by Xtend, the template paradigm for code generation is heavily employed in APS.
A	Model Ignorant Generation	As per the previous row, circumstances impose the usage of this pattern.
A	Generation Gap	The current state of the generator allows for only trivial specifications to be fully generated. For the other use cases, interfaces are generated, to be implemented by the developers. This is a proper separation of generated and hand-written code using inheritance.
NA	Transformer Generation	Due to relative closeness of the concrete and abstract syntax, as well as the tools provided by Xtext and Xtend, the code generation does not require tree transformations,
NA	Model-Aware Generation	Owing to a lack of an ontological analysis, the underlying semantic model in APS is undefined. Hence, no simulacrum of the model can be defined to enable model-aware generation.
4/4	**Front End**	
A*	BNF	As enforced by Xtext, the grammar of the language is defined in a BNF-derivative form.
A*	Parser Generator	This is done by virtue of Xtext, which itself employs ANTLR to generate a parser for APS.
A*	Tree Construction	Likewise enforced by the usage of Xtext.
A*	Notification	Due to Xtext's interface for validation, errors are collected and reported together, as per this pattern.
3.5/4	**User-centric**	
A*	Importance of DSL Tooling	The usage of Xtext enables out-of-the-box DSL tooling that enables good functionality to develop APS instances.
A	Limit the Number of Language Elements	The language is focused on providing a specification interface. Thus, it does not attempt to provide other functionality to the end-user.
A	Permit Comments	C-like comments, both single- and multi-line, are included in APS.
PA	Make Elements Distinguishable	At slight odds with the "Use the Same Style Everywhere" guideline, here the elements struggle to be unique, as the language elements follow the same naming conventions, affecting distinguishability

TABLE 7.1: Evaluation of APS.
This table is organised into groups, denoted by the grey rows, which further state how many guidelines from this group were applied. The rows are colour-coded, to indicate application status, as follows:

- Applied (A), denoted by blue; A* signifies tie to Xtext
- Partially Applied (PA), denoted by yellow; counts as 0.5 in grey row totals
- Not Applied (NA), denoted by red

7.3 Discussion

Answering this question has resulted in a set of relevant to APS design guidelines gathered from existing literature. These design patterns were then applied to the language as a form of evaluation, due to the lack of established methods. It has been shown that there is neither an academic consensus on how to achieve extensible software language design, nor systematic ways to evaluate DSLs. Consequently, a method for objective, specifically tailored evaluation has been proposed and employed — the usage of existing best practices as an objective benchmark of the state of a DSL.

There is abundant literature on best practices in DSL development, which was utilised in this chapter to find a set of guidelines, relevant to this thesis. Relevancy was judged based on objective measures (whether it applies to the correct type of DSL — embedded, vertical), alongside subjective decisions made by the language developers. Afterwards, from the set of 32 relevant guidelines, APS was found to utilise, partially or fully, 23 of them. It is important to note that there's three pairs of mutually exclusive guidelines, and a two that aren't beneficial to APS, meaning that 27 is the highest number APS could cover. This indicates a healthy state of the language. It is also a further testament to the efficacy of employing language workbenches and how they enable the development of software languages without the need for a team specialising in compiler construction.

Two of the best practice categories can be seen as problematic, e.g. less than half of the guidelines being applied (counting partially applied ones as half an application). They are the "Design" and "Language Layout" categories. Some of these practices simply do not fit the context of the language, i.e., they are concerned with computational models, whereas APS has no computational functionality at all. Other not applied practices are due to the genericity making judgement difficult. Regardless of these reasons, unused best practices reveal places for improvement in the language over time. The extension guidelines in the final research question specifically aim to address these areas of the language.

Simultaneously, the categories "Development Process", "Front End", and "User-centric" employ, partially or fully, all of the identified guidelines. This shows that there is a reasonable emphasis placed on achieving a robust front end, which satisfies end-user needs, through proven development processes. Furthermore, the "Back End" ones, which are all from Martin Fowler [22], are also all employed, if you exclude mutually exclusive pairs — Transformer vs. Template generation and Model-Aware vs. Model Ignorant Generation. Their full application is a testament to a well-implemented back end.

The implications for the general topic of this thesis are that the language is already in a good state from a high-level perspective. Currently understood best practices are employed to a reasonable degree and this has a positive effect on the extensibility of the language. Moreover, future extensions need to be done with the overall design and layout of the language in consideration, to achieve a higher degree of adherence to the relevant guidelines.

Chapter 8

RQ4. DSL Extension Guidelines

This chapter is the culmination of the thesis, answering the question "What are effective and actionable guidelines for DSL extensions to ensure maintainability?" — a contribution to the field of DSL engineering in the form of extension guidelines for iterative language development. These guidelines are devised for APS — an embedded, vertical DSL, which has been scrutinised in this case study. They are based on the findings from all the previous research questions. Despite the common terminology with the development **guidelines** (design patterns) of the previous research question, in RQ4 the guidelines are **novel**, designed for directing the extension of an already developed language and, moreover, curated specifically for APS.

8.1 Methodology

To answer the final research question, all the knowledge acquired from the former RQs is utilised. Hitherto, DSL extensibility factors, challenges (and positive practices) in APS development have been identified. Furthermore, the DSL has been scrutinised under the best practices from academia, highlighting objective strengths and weaknesses in the language. Combining all these findings will result in a list of extension guidelines — each of which will be related to at least one of the research questions, as substantiation for their inclusion and source for their relevance. The format for the answer to this RQ will be a set of decision-guiding soft rules, providing a direction for the software development processes of the language iterations. They are separated into 3 groups, based on the phase in which they are relevant — iteration planning (pre-implementation), development (implementation), and retrospective analysis (post-implementation). The steps included in devising the guidelines are:

1. Synthesis of Previous Findings:
   The previous findings are summarised to leverage the acquired knowledge from the prior research. Then, overarching themes are identified by collating:
   - The identified factors influencing DSL extensibility
   - The challenges of APS development
   - The low-friction areas of APS development
   - The adherence to language design patterns
2. Formulation of Actionable Guidelines:
   The synthesis of the previous findings was integrated to create guidelines, which address identified gaps in the language (challenges from RQ2 and missed design pattern applications from RQ3). Extensibility factors from RQ1 were emphasised, and the continuation of good practices was included. Here, the guidelines were also separated into categories, regarding when they are relevant to an extension's development.
3. Expert Validation:
   Experts were consulted to ensure the relevancy of the devised guidelines. As the aim of this research question is to arrive at effective and actionable guidelines, the supervisors of the thesis helped to ensure the correctness and effectiveness of the guidelines, whereas the language developers were queried so that the final guidelines are indeed relevant and actionable.
4. Retroactive Application and Impact Analysis:
   Further validation is done by retroactively applying the guidelines on a past language extension. To retroactively apply the guidelines, the extension proposal was examined with a lead APS developer to reflect on the **pre-implementation** guidelines. The code was then analysed, comparing the difference before and after the extension, to validate the **implementation** guidelines. The **post-implementation** guidelines could be directly applied, however, they require an end user to contact, which was not available at the time of writing the thesis. Still, they were partially applied and reflected upon. This process requires high-level abstraction and analysis of how the guidelines could have impacted important factors such as development time, usability, and maintainability. Based on the expected impact, adjustments are made to lower the intrusiveness and increase the desired impacts of the guidelines.

8.2 Results

8.2.1 Extension guidelines for APS

Firstly, to summarise the findings of the thesis, there are 4 categories of relevant results:

- Extensibility factors for DSLs (RQ1)
  - Documentation quality, **regression testing**, **LW impact**, language grammar, **DSL usability**
- Challenges in APS development (RQ2)
  - Legacy extensions, language restructurings, **maintaining objectivity concerning usability**, preserving language consistency, proper domain knowledge context, designing for evolution, lack of feedback integration, the economic viability of extensions
- Low-friction areas in APS development (RQ2)
  - Proper responsibility distribution, **usage of LW**, emphasis on readability, impact analysis of extensions, **regression testing**
- State of APS under scrutiny of academic development guidelines (RQ3)
  - The front end, back end, development process and user-centric features are all up to standard, following academic convention. However, the overall design and language layout are potential points for improvement

The bold entries are ones which appear more than once, reinforcing their validity and helping to draw common themes. Using these findings, guidelines are created to preserve the good areas of the language, simultaneously emphasising improving identified issues and relating them to the extensibility factors and challenges.

As established in (RQ3), iterative development is considered good practice in DSL development [13], and is furthermore employed in the development of APS. Thus, the extension guidelines are tailored to this process, with a categorisation of guidelines concerning planning (pre-implementation), development (implementation) and the retrospective period (post-implementation). Each guideline is supported by findings of at least one prior RQ. They aim to supplement the development process, helping to address potential weak points of the language and ensuring the continuation of good practices.

For the sake of brevity and to avoid ambiguity, the tentative guidelines developed, which were refined in the next section are **not included**. Instead, the final guidelines can be found in the conclusion chapter, as the main contribution of this thesis.

8.2.2 Validation of Guidelines via Retroactive Application

After collating all the findings of the previous research questions, a tentative list of guidelines was devised. Though substantiated and detailed, their effectiveness and impact were uncertain. To validate and refine them for future use, they were retroactively applied to a previous extension of the language.

This requires abstraction and guessing — ultimately, dealing with a hypothetical always does. The extension chosen for the validation was a recent one, which focused on adding functionality to the language, the most common type of extension as per developer

interviews in RQ2. Namely, 4 constructs in the language received an additional optional specification block, which enabled the usage of the Generation Gap pattern [22].

This validation led to the following refinement of the guidelines:

- Addition of a pre-implementation guideline about ensuring backward compatibility:
  Looking at a real-world extension the language went through, a missing consideration was identified. Though the focus of the guidelines and this thesis is evolution-first design, no emphasis was placed directly on backward compatibility. Though the extension **was** backwards compatible, as it added an optional specification block to elements of the language, it highlighted that such additions **need** to be optional. Otherwise, all language instances in production would need to be updated, massively impacting the extension cost.

- Removal of pre-implementation guideline due to lack of specificity:
  One of the tentative pre-implementation guides was "Treat conceptual problems like you would in other software engineering contexts (from RQ1 [40] [57])." While sound advice, this guideline does not fall under the umbrella of "actionable", which is the aim of this RQ. This remains solid advice to follow during extensions but was found to fail in delivering an action plan.

- Adjustment of pre-implementation guideline regarding using patterns from RQ3.4:
  The extension used was found to apply a pattern from the previous research question, namely Generation Gap [22]. This led to a change to the pre-implementation guideline advising developers to consider using unapplied patterns from the previous research question if they have an opportunity. The aim was to improve the state of the language from that lens. However, patterns often can be re-applied, thus one must consider them all, rather than focusing only on the ones which had not been used. Thus the word "un-applied" was removed from the guideline

- Adjustment of the explanation of the implementation guideline about the language grammar construction:
  This guideline was found to be too vague and was elaborated upon to increase the specificity. To make the guidelines actionable, there needs to be an explanation of how they can be directly used to enrich the development process.

- Specifying rules for documentation maintenance during implementation:
  To facilitate a proper state of the user and developer documentation, rigid rules were added, which would make it clear which parts of the documentation to monitor. This was added as a result of inconsistencies in the documentation being observed, which would not necessarily be addressed by the guideline without more rigid rules.

- Introduction of a usability questionnaire template for the Formal Feedback Loop guideline:
  After an extension is complete, a usability questionnaire is advised to be sent out to end users. However, this is not specific enough to facilitate smoother development, especially considering the gap in DSL usability evaluation methods. Thus, a template was found to be necessary to feasibly apply this guideline.

Overall, out of 13 guidelines, 6 were found to have been at least partially carried in this extension's development, with two unused ones needing end users that simply were not available at this stage of the language's development. Namely, the extensions concerning impact analysis, employing DSL design patterns, leveraging continuous integration,

language grammar construction, maintaining documentation, and performance monitoring were a part of the extension process. However, without a rigid process, there is no guarantee they will be followed in other extensions. Thus, the collection of these guidelines helps to ensure consistency in adherence to best practices.

After the refinement, each guideline is both actionable and specific. The final result of this RQ can be found in the Conclusion for the sake of emphasising on the guidelines as the main contribution of this thesis to the field of DSL engineering.

Usability Questionnaire

As established during the refinement, to be actionable the formal feedback loop post-implementation guideline needs to be accompanied by a usability questionnaire template for new language extensions. While usability in DSLs is a researched topic [53], no DSL usability questionnaire has been formally defined yet. However, DSLs can be treated as user interfaces to apply common methodologies for usability evaluation. Two frameworks have already done this, USE-ME [5] and Usa-DSL [54], and in a similar fashion, here we repurpose the widely adopted and validated System Usability Scale (SUS) [9]. We adapt it for language extensions and extend it to include questions regarding DSL quality attributes (readability, expressiveness). The SUS contains a collection of usability-related statements with a Likert scale [45] for the user to express their stance. Hence the following usability questionnaire template was developed. The first 10 statements are adapted by substituting 'system' for 'feature', to make the scale applicable to language extensions. The latter two statements are added on top, to query specifically about readability and expressiveness — important DSL-specific quality attributes. This is a template, which can of course be adjusted should the circumstances require it, e.g. if a language extension did not introduce a new feature but rather constituted other changes.

	Strongly disagree	Disagree	Neutral	Agree	Strongly Agree
I think that I would like to use this feature frequently.	○	○	○	○	○
I found the feature unnecessarily complex.	○	○	○	○	○
I thought the feature was easy to use.	○	○	○	○	○
I think that I would need the support of a technical person to be able to use this feature.	○	○	○	○	○
I found the feature well integrated into the language.	○	○	○	○	○
I thought this feature introduced inconsistency in the language.	○	○	○	○	○
I would imagine that most people would learn to use this feature very quickly.	○	○	○	○	○
I found the feature very cumbersome to use.	○	○	○	○	○
I felt very confident using the feature.	○	○	○	○	○
I needed to learn a lot of things before I could get going with this feature.	○	○	○	○	○
I found this feature negatively affected the readability of my specifications.	○	○	○	○	○
I found this feature sufficiently expressive for the language.	○	○	○	○	○

Table 8.1: Usability Questionnaire for DSL
(adapted SUS [9])

8.3 Discussion

In this RQ, all previous findings were leveraged to develop DSL extension guidelines, which ensure usability and maintainability. While curated for APS, these guidelines nonetheless mark a significant step in DSL extensibility. They were devised in a systematic approach, yielding key insights and general recommendations for DSL development.

Integration of Extensibility Factors

The factors influencing DSL extensibility identified in RQ1 were used as a foundation to understand what contributes to a maintainable DSL. These factors provided a direction for the final extension guidelines and helped substantiate many of them.

Addressing APS Challenges and Maintaining Beneficial Practices

As opposed to the extensibility factors, these challenges and good practices from RQ2 served as a way to make the guidelines more specific to APS. All the identified issues were addressed in at least one guideline. On the other hand, the good practices identified were leveraged as well, to make sure that the strengths are maintained.

Application of Existing Guidelines

The application of design patterns from RQ3 provided a theoretical framework which was validated and refined through practical insights gained from the case study. This approach combined the state-of-the-art in academia with the language from this case study to deliver both a general direction for DSL design, as well as specificity to APS, corroborating both previous RQs. Moreover, it provides a collection of best practices that are relevant to APS, which further serves as a benchmark which can be monitored over time.

Retroactive Application

Applying the guidelines to a past APS extension provided a demonstration of their potential impact. This analysis revealed points for improvement in the guidelines, but also in the development process itself, as intended. This served to show their effectiveness and improve their specificity.

The methodology employed in this RQ successfully synthesised the findings from all previous research questions and delivered practical and effective guidelines for APS extensions. By integrating insights from literature, addressing APS-specific challenges, and validating through retroactive application, the guidelines were ensured to be both robust and relevant. This comprehensive approach can be reused for other industrial languages to develop relevant therein guidelines. The final result is curated to APS, but also leans on academic DSL literature, thus can be made use of for other languages. However, relevancy in that case cannot be guaranteed.

Chapter 9

Conclusion

In this thesis, the extensibility of DSLs has been explored, with a focus on a DSL from the industry. Multiple research questions were devised to tackle the challenge of composing DSL extension guidelines. The extensibility of software languages was researched to find general insights and APS was examined to establish relevancy. In this chapter, we discuss and reflect on the entire process. The extension guidelines are provided, as the main contribution of this thesis. Furthermore, threads to the validity of the research are discussed. Finally, suggestions for future work are given.

9.1 Summary of Findings

The research questions guiding this thesis were systematically addressed through a literature review, interviews with industry developers, and rigorous analysis of the APS language. The key findings from each research question are summarised below:

1. Extensibility Factors of DSLs (RQ1):
   Identified critical factors influencing the extensibility of DSLs, leaning on software engineering practices and software language engineering literature, including DSL-specific papers. It was found that software extensibility factors extended to DSL development and the importance of documentation, regression testing, and usability was highlighted.

2. Challenges in Extending APS (RQ2):
   Uncovered specific challenges faced in the iterative development of APS, such as maintaining language consistency, implementing legacy support and ensuring usability. Furthermore, the successes of the development experience were highlighted. These findings corroborated the previous claims of software engineering practices extending to DSL development and the importance of language workbenches.

3. Existing Design Guidelines and Evaluation Methods (RQ3):
   Outlined a gap in DSL literature in the realm of DSL evaluation. Consequently identified the existing state-of-the-art design guidelines and leveraged them to perform an objective language evaluation. This produced an overview of the state of APS, allowing the final extension guidelines to address any gaps in the language.

4. Devised DSL Extension Guidelines (RQ4):
   All the previous findings culminated in the creation of novel extensibility guidelines, which were further validated and refined via a retroactive application to a past APS extension. The final guidelines can be found below — a collection of actionable and specific directions for extending APS.

9.2 Contributions to the Field

The world of software language engineering is mature but still evolving. Branches like DSL engineering have significant and extensive literature, but also notable gaps and inconsistencies. This thesis highlighted the equivocal definition of "extensibility", outlined gaps in the literature regarding evolution-centric DSL design, and shed light on the state of rarely-mentioned in academia vertical, embedded DSLs. Moreover, a systematic and objective DSL evaluation method was devised and employed, which relies on determining the adherence of a language to the existing design standards in academia.

Finally, the main contribution is a list of DSL extension guidelines, as well as the method for their synthesis. While they are tuned for APS, they can be relevant to other DSLs, especially other vertical, embedded ones. They are split into categories, which indicate when they should be employed.

Each category of guidelines is sorted by priority, denoted by `[H]` for High, `[M]` for Medium and `[L]` for Low. Naturally, all the guidelines are conducive to developing an ideal extension, but time and resource constraints often do not allow for the perfect extension process. Hence, the priorities are meant to guide a developer under such constraints.

Pre-implementation

1. `[H]` Rigorous Impact Analysis (from developer interviews in RQ2)

    - Cost-Benefit Analysis is mandatory before expending resources on an extension. The economic viability of extensions is a key challenge in language development and, therefore should be a first consideration before implementation.
    - Software Impact Analysis before development was identified as beneficial by the developers. It allows for assessing the risk of implementing an extension and provides a moment to ensure backward compatibility of the proposed changes with the older language version. The continuation of this practice is heavily encouraged.
    - Consistency Checks are a low-cost method of avoiding expensive problems. Reaching out and discussing potential extensions with all development teams for language consistency is heavily encouraged.

2. `[H]` Consider Usability Impact of Extensions (from RQ1 [13] [55], developer interviews in RQ2)

    - Usability has been established to be a pivotal aspect of DSLs, enabling their adoption. It is important to **consciously** consider the impact of potential extensions, especially ones that offer new functionality to the user. This appears again in the Formal Feedback Loop post-implementation guideline, where usability data is gathered, which can provide valuable insights for future extensions.

3. `[H]` Preserve backward compatibility and aim for forward compatibility (from developer interviews in RQ2, RQ3 [13])

    - Whenever new constructs are added to the language if they belong to other constructs, they need to be an optional addition, to ensure all previous code remains compilable. Conversely, additions of new constructs must carefully analyse whether some mandatory option will need to be defined, as it can only be done with its addition to ensure forward compatibility. These considerations need to be made with great care, or the language will suffer.

4. `[M]` Define a usage convention (from RQ3 [34])

    - Usage convention refers to the established practices for writing and structuring software language instances. These conventions help make the code more readable, maintainable, and consistent across users and use cases. When adding a new language feature, envision a usage convention for it. As the language evolves, this allows for a language-wide consistent usage convention.

5. `[M]` Co-evolution Planning (from RQ1 [64])

    - As previously discussed, a vertical, embedded DSL, such as APS, has dependencies. When external code changes impact interfaces, this can force changes in the generated code and vice versa. If this is not considered on time, it can cause expensive problems for the developers. Maintaining a knowledge base with interfaces the DSL interacts with is recommended, such that timely updates can be made whenever relevant external changes require it and vice versa.

6. `[L]` Consider using DSL design patterns for large extensions (from the evaluation in RQ3.4)

    - As was found in RQ3, some opportunities for DSL pattern application are missed (Foreign Code, Organisational Structures for Models, Usage Conventions, etc.). Awareness of their existence and consideration is beneficial. When time allows, deeper reflection on the abstractions required to extend the language can lead to an opportunity to apply these patterns and improve the state of the language.

Implementation

1. `[H]` Leverage continuous integration and regression testing (from RQ1 [23], developer interviews in RQ2)

    - Automated testing helps catch issues early and increases the confidence developers can have in the stability of the project. This is supported by the prevalence of this practice within the field of software engineering and by direct accounts of the APS developers.

2. `[H]` Optimise the construction of the language grammar (from RQ1 [32], developer interviews in RQ2, RQ3 [34])

    - One challenge in the language's development found in the interviews in RQ2 was a case where two semantically identical concepts were implemented with two different syntactic structures. This violates two patterns from RQ3 (avoid conceptual redundancy [34], use the same style everywhere [34]). Extracting grammar commonalities into non-terminals for reuse is recommended, which is conducive to avoiding such violations. It furthermore makes maintenance easier, enabling easier changes of the syntax and more readable constructs.
    - The grammar in textual DSL can be a significant bottleneck to language extensions. As explored in RQ3, introducing new syntax involves modifying the grammar, which may lead to unintentional grammar ambiguities. This leads to unpredictable and unwanted behaviour, which can require significant effort to remedy. While this is an eventuality in grammarware, mindfulness of the risks can prevent larger issues from occurring.

3. `[M]` Maintain Comprehensive Documentation (from RQ1 [2] [47])

    - User Documentation: Develop clear and thorough user manuals and reference guides for all changes to the language. This is crucial for improving usability and facilitating user adoption.
    - Developer Documentation: Ensure consistent and pervasive usage of Javadoc (or other applicable documentation tools) to document the code; Integrate developer documentation in the **code review process**, to establish the ubiquity and quality of the documentation. Follow and enforce a consistent documentation convention for the development team.

4. `[L]` Prefer optimal implementations (from developer interviews in RQ2, RQ3 [34])

    - Performance is not a focus of specification languages (such as APS). However, as the language continues to mature, the performance of the code generator increasingly becomes a hurdle in development efforts. The build times for the

tool can affect the productivity of developers. Furthermore, the efficiency of the generated code can become increasingly important as the language develops. Neglecting this aspect can build up a need for significant refactoring, which has been found in the RQ2 interviews to be a major challenge.

Post-implementation

1. [H] Formal Feedback Loop via Usability Questionnaire (from RQ1 [55], developer interviews in RQ2)

    - One week after rolling out an extension to clients, send a usability questionnaire as a feedback form. Some time needs to have passed to allow users to familiarise themselves with the changes to provide unbiased feedback.
    - This provides an opportunity to collect and analyse feedback, a crucial necessity identified by the developers in the interviews for RQ2. This feedback can be integrated into the next iteration planning and usability can be tracked over time.

2. [M] Update Documentation (from RQ1 [2] [47])

    - Update user documentation based on the feedback to address issues and gaps identified. This helps to avoid a documentation backlog, dedicating time to implementing documentation feedback after each extension.

3. [L] Performance Monitoring and Optimisation (from RQ2, RQ3 [34])

    - Monitor the performance metrics of the DSL and the extension's impact. Consider both the performance of the generated code and the code generator.
    - Flag any potential performance pitfalls introduced and prioritise removing them in future iterations.

These guidelines impact the development process in a minimally intrusive manner while achieving a more maintainable code base, conforming to the discovered DSL extensibility standards. For APS extensions examined, this would mean achieving the same practical outcomes, but with more care put into planning and future-proofing the design, as well as defining and sticking to a usage convention. Furthermore, the quality of the documentation can improve, by increasing its ubiquity and consistency. Finally, the guidelines set up the APS development for client distribution, by defining a course of action for rolling out extensions to end clients.

On a higher level, addressing a notable gap in the literature, this thesis provides a systematic way of evaluating and extending a DSL. This furthers our understanding of DSL extensibility and paves the way for the construction and refinement of systematic SLE approaches. Outside the scope of APS, these guidelines represent a first attempt to formalise the process of extending DSLs. Although they may not apply to other languages, this research can serve as a reference point for creating further, more general guidelines.

9.3 Threats to Validity

In this section, any threats to the validity are discussed. By acknowledging these threats and explaining the undertaken measures, this thesis aims to provide a transparent analysis of the extensibility of DSLs, contributing to a genuine advancement of the field.

9.3.1 Internal Validity

1. Interview Bias:

    - **Threat**: The data collected from the interviews in RQ2 may be subject to bias, either from the interviewer's influence or the interviewee's subjective perspectives.
    - **Mitigation**: To mitigate this, a semi-structured format was used to enforce consistency between the interviews. Moreover, multiple developers from different roles and experience levels were interviewed to obtain a balanced perspective.

2. Subjectivity in Qualitative Analysis:

    - **Threat**: The analysis of interview data could introduce subjectivity, influencing the drawn conclusions.
    - **Mitigation**: A systematic approach was employed in the analysis, using expert validation by the thesis supervisors to draw general conclusions from the data

9.3.2 External Validity

1. Generalisability:

    - **Threat**: As acknowledged during the research, the main contribution of the thesis, the extension guidelines, is curated for one specific DSL. Ergo, their external validity (applicability outside of APS) is uncertain.
    - **Mitigation**: To combat this, the research meticulously documented the type of language they were devised for (vertical, embedded, specification language) and also the specific challenges and gaps in the language design, which the guidelines were based on. Therefore, external readers can juxtapose their language with APS, to determine if the guidelines will align.

2. Case Study Limitations:

    - **Threat**: The research is partially based on a single case study, further limiting the generalisability of the findings.
    - **Mitigation**: The method for their derivation, along with the objective evaluation framework has been documented such that it can be reused for other languages. Furthermore, the literature was cross-referenced heavily to corroborate all practical findings.

9.4 Future Work

This research has exposed gaps in the DSL literature, namely terminological inconsistencies and scarcity in DSL evaluation and evolution- and usability-centered design. Therefore, future work to follow up this thesis can include establishing clearer definitions of terms such as "extensibility", "maintainability", "language evolution", "modifiability", and "domain-specific modelling".

Moreover, the evaluation method of measuring adherence to academic guidelines can be used in broader case studies, for different types of DSLs. Likewise, the method for devising fine-tuned extension guidelines can be reused and its merits to general approaches are discussed.

Moreover, the usability questionnaire developed in RQ4 can be validated and more similar templates can be developed. For the sake of the adoption of usability in the field of DSL engineering, methods for quantifying the usability of languages need to be standardised and provided to DSL developers.

Acknowledgements

The author of this thesis extends their gratitude to all persons involved in the research process, for making it possible, frictionless and enjoyable. The cooperation of APS developers helped shed light on an under-researched world of business DSLs. Special thanks to Jarno van der Sanden, whose continuous support and expertise helped bridge the gap between the author and the industrial DSL. Warm appreciation is given to the university supervisors — Vadim Zaytsev and Nhat Bui — without whom this thesis would not have been possible in any capacity. Lastly, many thanks to Edgars Gaisiņš for the feedback and suggestions provided in the writing process.

Bibliography

[1] "Call To Action: Secure the future maintenance of Xtext", 2020. Last Accessed: 2024/03/20. URL: https://github.com/eclipse/xtext/issues/1721.

[2] Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. Software documentation issues unveiled. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1199–1210. IEEE, 2019.

[3] Omer Faruk Alaca, Baris Tekin Tezel, Moharram Challenger, Miguel Goulão, Vasco Amaral, and Geylani Kardas. Agentdsm-eval: A framework for the evaluation of domain-specific modeling languages for multi-agent systems. *Computer Standards & Interfaces*, 76:103513, 2021.

[4] Diego Albuquerque, Bruno Cafeo, Alessandro Garcia, Simone Barbosa, Silvia Abrahão, and António Ribeiro. Quantifying usability of domain-specific languages: An empirical study on software maintenance. *Journal of Systems and Software*, 101:245–259, 2015.

[5] Ankica Barišić, Vasco Amaral, and Miguel Goulão. Usability driven dsl development with use-me. *Computer Languages, Systems & Structures*, 51:118–157, 2018.

[6] Ankica Barišic, Vasco Amaral, Miguel Goulão, and Bruno Barroca. Evaluating the usability of domain-specific languages. In *Software Design and Development: Concepts, Methodologies, Tools, and Applications*, pages 2120–2141. IGI Global, 2014.

[7] Ankica Barisic, Vasco Amaral, and Miguel Goulão. Domain-specific language domain analysis and evaluation: a systematic literature review, 09 2015. doi:10.5281/zenodo.265487.

[8] Holger Stadel Borum and Christoph Seidl. Survey of established practices in the life cycle of domain-specific languages. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*, pages 266–277, 2022.

[9] John Brooke et al. Sus-a quick and dirty usability scale. *Usability evaluation in industry*, 189(194):4–7, 1996.

[10] Tibor Brunner and Zoltán Porkoláb. Programming language history: Experiences based on the evolution of c++. In *Proceedings of the 10th International Conference on Applied Informatics*, pages 63–71, 2017.

[11] Antonio Bucchiarone, Antonio Cicchetti, Federico Ciccozzi, and Alfonso Pierantonio. *Domain-specific languages in practice: with JetBrains MPS*. Springer Nature, 2021.

[12] Luca Cardelli, Florian Matthes, and Martín Abadi. Extensible grammars for language specialization. In *Database Programming Languages (DBPL-4) Proceedings of the Fourth International Workshop on Database Programming Languages—Object Models and Languages, Manhattan, New York City, USA, 30 August–1 September 1993*, pages 11–31. Springer, 1993.

[13] Gerald Czech, Michael Moser, and Josef Pichler. A systematic mapping study on best practices for domain-specific modeling. *Software Quality Journal*, 28(2):663–692, 2020.

[14] Eclipse Foundation. Xtend is a flexible and expressive dialect of Java, which compiles into readable Java 8 compatible source code. https://eclipse.dev/Xtext/xtend/, 2024. Last Accessed: 2024/03/26.

[15] Sebastian Erdweg, Paolo G Giarrusso, and Tillmann Rendel. Language composition untangled. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications*, pages 1–8, 2012.

[16] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. SugarJ: Library-based syntactic language extensibility. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, pages 391–406, 2011.

[17] Sebastian Erdweg and Felix Rieger. A framework for extensible languages. In *Proceedings of the 12th international conference on Generative programming: concepts & experiences*, pages 3–12, 2013.

[18] Sebastian Erdweg, Tijs Van Der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, et al. The state of the art in language workbenches: Conclusions from the language workbench challenge. In *Software Language Engineering: 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings 6*, pages 197–217. Springer, 2013.

[19] Moritz Eysholdt and Heiko Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 307–309, 2010.

[20] Rodney Farrow, Thomas J Marlowe, and Daniel M Yellin. Composable attribute grammars: Support for modularity in translator design and implementation. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 223–234, 1992.

[21] Martin Fowler. Language workbenches: The killer-app for domain specific languages, 2005.

[22] Martin Fowler. *Domain-specific languages*. Pearson Education, 2010.

[23] Martin Fowler and Matthew Foemmel. Continuous integration, 2006.

[24] Ulrich Frank. Domain-specific modeling languages: requirements analysis and design guidelines. *Domain engineering: Product lines, languages, and conceptual models*, pages 133–157, 2013.

[25] Pedro Gabriel, Miguel Goulao, and Vasco Amaral. Do software languages engineers evaluate their languages? *arXiv preprint arXiv:1109.6794*, 2011.

[26] Rohit Gupta, Sieglinde Kranz, Nikolaus Regnat, Bernhard Rumpe, and Andreas Wortmann. Towards a systematic engineering of industrial domain-specific languages. In *2021 IEEE/ACM 8th International Workshop on Software Engineering Research and Industrial Practice (SER&IP)*, pages 49–56. IEEE, 2021.

[27] Rohit Gupta, Sieglinde Kranz, Nikolaus Regnat, Bernhard Rumpe, and Andreas Wortmann. Towards a systematic engineering of industrial domain-specific languages. In *2021 IEEE/ACM 8th International Workshop on Software Engineering Research and Industrial Practice (SER&IP)*, pages 49–56. IEEE, 2021.

[28] International Organization for Standardization. Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. ISO/IEC 25010:2011, 2011. Available online: https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en.

[29] International Organization for Standardization. Ergonomics of human-system interaction – Part 11: Usability: Definitions and concepts. ISO 9241-11:2018, 2018. Available online: https://www.iso.org/obp/ui/#iso:std:iso:9241:-11:ed-2:v1:en.

[30] Aníbal Iung, João Carbonell, Luciano Marchezan, Elder Rodrigues, Maicon Bernardino, Fabio Paulo Basso, and Bruno Medeiros. Systematic mapping study on domain-specific language development tools. *Empirical Software Engineering*, 25:4205–4249, 2020.

[31] Niklas Johansson and Anton Löfgren. Designing for extensibility: An action research study of maximizing extensibility by means of design principles. B.S. thesis, 2009.

[32] Adrian Johnstone, Elizabeth Scott, and Mark van den Brand. Modular grammar specification. *Science of Computer Programming*, 87:23–43, 2014.

[33] Gökhan Kahraman and Semih Bilgen. A framework for qualitative assessment of domain-specific languages. *Software & Systems Modeling*, 14:1505–1526, 2015.

[34] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design guidelines for domain specific languages. *arXiv preprint arXiv:1409.2378*, 2014.

[35] Kohsuke Kawaguchi. https://www.jenkins.io/. Last Accessed: 2024/04/26.

[36] Steven Kelly and Risto Pohjonen. Worst practices for domain-specific modeling. *IEEE software*, 26(4):22–29, 2009.

[37] Heiko Kern. *Model interoperability between meta-modeling environments by using M3-level-based bridges*. PhD thesis, Universität Leipzig, 2016.

[38] Barbara Kitchenham, Stuart Charters, et al. Guidelines for performing systematic literature reviews in software engineering, 2007.

[39] Anneke Kleppe. *Software language engineering: creating domain-specific languages using metamodels*. Pearson Education, 2008.

[40] Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(3):331–380, 2005.

[41] Paul Klint, Tijs Van Der Storm, and Jurgen Vinju. On the impact of dsl tools on the maintainability of language implementations. In *Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications*, pages 1–9, 2010.

[42] Donald E Knuth. Backus normal form vs. backus naur form. *Communications of the ACM*, 7(12):735–736, 1964.

[43] Tomaž Kosar, Sudev Bohra, and Marjan Mernik. Domain-specific languages: A systematic mapping study. *Information and Software Technology*, 71:77–91, 2016.

[44] Manuel Leduc, Thomas Degueule, Eric Van Wyk, and Benoit Combemale. The software language extension problem. *Software and Systems Modeling*, 19(2):263–267, 2020.

[45] Rensis Likert. A technique for the measurement of attitudes. *Archives of psychology*, 1932.

[46] Brian A Malloy and James F Power. An empirical analysis of the transition from python 2 to python 3. *Empirical Software Engineering*, 24:751–778, 2019.

[47] Robert C Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[48] Marjan Mernik, Jan Heering, and Anthony M Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.

[49] Microsoft. Language Server Protocol. https://microsoft.github.io/language-server-protocol/. Accessed: 2024-03-05.

[50] Steven Muchnick. *Advanced compiler design implementation*. Morgan kaufmann, 1997.

[51] José Évora Gómez et al. Octavio Roncal Andrés, José Juan Hernández Cabrera. Tara: Streamlining dsl development through syntactic patterns. December 2023. doi:10.21203/rs.3.rs-3758685/v1.

[52] Bruno C d S Oliveira and William R Cook. Extensibility for the masses: Practical extensibility with object algebras. In *European Conference on Object-Oriented Programming*, pages 2–27. Springer, 2012.

[53] Ildevana Poltronieri, Allan Christopher Pedroso, Avelino Francisco Zorzo, Maicon Bernardino, and Marcia de Borba Campos. Is usability evaluation of dsl still a trending topic? In *Human-Computer Interaction. Theory, Methods and Tools: Thematic Area, HCI 2021, Held as Part of the 23rd HCI International Conference, HCII 2021, Virtual Event, July 24–29, 2021, Proceedings, Part I 23*, pages 299–317. Springer, 2021.

[54] Ildevana Poltronieri, Avelino Francisco Zorzo, Maicon Bernardino, and Marcia de Borba Campos. Usa-dsl: usability evaluation framework for domain-specific languages. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pages 2013–2021, 2018.

[55] Ildevana Poltronieri Rodrigues, Márcia de Borba Campos, and Avelino F Zorzo. Usability evaluation of domain-specific languages: a systematic literature review. In *Human-Computer Interaction. User Interface Design, Development and Multimodality: 19th International Conference, HCI International 2017, Vancouver, BC, Canada, July 9-14, 2017, Proceedings, Part I 19*, pages 522–534. Springer, 2017.

[56] Simmi K Ratan, Tanu Anand, and John Ratan. Formulation of research question - stepwise approach. *J. Indian Assoc. Pediatr. Surg.*, 24(1):15–20, January 2019.

[57] Daniel Ratiu and Markus Voelter. Automated testing of dsl implementations: experiences from building mbeddr. In *Proceedings of the 11th International Workshop on Automation of Software Test*, pages 15–21, 2016.

[58] Lukas Renggli. *Dynamic language embedding with homogeneous tool support*. Lukas Renggli, 2010.

[59] Ildevana Poltronieri Rodrigues, Avelino Francisco Zorzo, Maicon Bernardino da Silveira, Bruno Medeiros, and Marcia de Borba Campos. Heuristic evaluation checklist for domain-specific languages. *HUCAPP, 2021, Brasil.*, 2021.

[60] SlashData. State of Continuous Delivery Report 2023: The Evolution of Software Delivery Performance. https://cd.foundation/state-of-cd-2023/, 2023. Last Accessed: 2024/04/22.

[61] Diomidis Spinellis. Notable design patterns for domain-specific languages. *Journal of systems and software*, 56(1):91–99, 2001.

[62] Thomas A Standish. Extensibility in programming language design. In *Proceedings of the May 19-22, 1975, national computer conference and exposition*, pages 287–290, 1975.

[63] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[64] Jürgen Thanhofer-Pilisch, Alexander Lang, Michael Vierhauser, and Rick Rabiser. A systematic mapping study on dsl evolution. In *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 149–156. IEEE, 2017.

[65] Juha-Pekka Tolvanen and Steven Kelly. Evaluating tool support for co-evolution of modeling languages, tools and models. In *2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 914–923. IEEE, 2023.

[66] Federico Tomassetti and Vadim Zaytsev. Reflections on the lack of adoption of domain specific languages. In *STAF Workshops*, pages 85–94, 2020.

[67] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. Silver: An extensible attribute grammar system. *Science of Computer Programming*, 75(1-2):39–54, 2010.

[68] Markus Völter. Best practices for dsls and model-driven development. *Journal of Object Technology*, 8(6):79–102, 2009.

[69] Andrzej Wąsowski and Thorsten Berger. *Domain-Specific Languages: Effective modeling, automation, and reuse*. Springer, 2023.

[70] David Wile. Lessons learned from real DSL experiments. *Science of Computer Programming*, 51(3):265–290, 2004.

[71] Vadim Zaytsev. Language design with intent. In *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 45–52. IEEE, 2017.

Appendix A

Internship Task Description

A.1 Background Information

The Machine Control & Infrastructure (MCI) department is responsible for the definition and handling of various Machine input settings. These settings are used by customers of ASML to define the structure and layout of a wafer along with all settings needed to correctly process wafers in the Lithography Machine.

Extending the software with new input settings has become a repetitive and time-consuming task, while many clusters within ASML depend on this for adding new functionality to the machine. To reduce the effort and lead time of adding new settings, the MCI department has developed a data-driven Code Generator to automate the addition of new settings. For this purpose, a Domain Specific Language (DSL) has been created in which input settings can be defined. The definitions are stored in ASML Parameter Specification (APS) files. The Code Generator has been developed with the latest technologies, using Xtext to design and create the DSL and Xtend to implement the Code Generator. The Code Generator itself generates state-of-the-art C++ code.

A.2 Assignment

The Code Generator tool is being developed iteratively, adding new functionality with each iteration. Meanwhile, the working parts of the Code Generator tool are already being used by clients. This makes it challenging to extend the DSL with new language features while keeping client impact to a minimum.

This task consists of the following:

1. Investigate current DSL and its extensibility;
2. Investigate the impact of DSL on internal/external interfaces;
3. Provide guidelines for DSL extensions (from client and development perspective);
4. Propose DSL optimisations and demonstrate them in a proof of concept.

Author's note: Of these 4 sub-tasks, 1 and 3 are directly related to this research, while the others are tangential and may not be documented

---

Appendix B

Developer Interviews

B.1 Consent Brochure for Interview

Study Information

This is a study aiming to evaluate the current state of the APS DSL and to devise guidelines for extensions to the language. To answer my first research question, which is "What are the current challenges in extending APS?", I need to perform interviews with people with experience in developing the language. The interviews will be recorded and transcribed for the purposes of extraction of relevant data. The audio recordings will be deleted within 5 days of the interview, as soon as they have been transcribed. The transcriptions will be kept with the researcher for the purposes of writing the thesis, until it has been completed or until the 5th of July. Afterwards the transcriptions will be deleted too.

The participant retains the right to ask for the data to be deleted at any time. The gathered information will be shared with the participant after the interview, to ensure that their opinions are correctly represented. They will also get access to the thesis whenever it is completed.

Consent Form for Extensibility Of Domain-Specific Languages: A Case Study of an industrial DSL

YOU WILL BE GIVEN A COPY OF THIS INFORMED CONSENT FORM

Please tick the appropriate boxes	Yes	No
Taking part in the study		
I have read and understood the study information or it has been read to me. I have been able to ask questions about the study and my questions have been answered to my satisfaction.	○	○
I consent voluntarily to be a participant in this study and understand that I can refuse to answer questions and I can withdraw from the study at any time, without having to give a reason.	○	○
I understand that taking part in the study involves an audio-recorded, transcribed interview.	○	○
Use of the information in the study		
I understand that information I provide will be used for a thesis about DSL extensibility.	○	○
I understand that personal information collected about me that can identify me, such as my name or my position, will not be shared beyond the study organizer.	○	○
I agree to be audio recorded.	○	○

Signatures

______________________ ______________________ ________

Name of participant

Signature Date

I have accurately read out the information sheet to the potential participant and, to the best of my ability, ensured that the participant understands to what they are freely consenting.

Naum Tomov ______________________ ________

Name of researcher

Signature Date

UNIVERSITY OF TWENTE.

B.2 Questions for Interview

1. Can you briefly describe your involvement in the development of APS?
2. Do you, as a user (or as a developer) of APS spend more time reading or writing APS code?
3. Are APS extensions mainly focused on the front end or the back end?
4. Have you had instances where an extension does not meet its intended goal (had to be removed or redone)?
5. Do you recall any examples of challenges you've had with the development of APS?
6. In your experience, have the challenges encountered during APS extensions been predominantly technical (algorithms, framework constraints, etc) or structural (syntax, abstraction level, design choices)?
7. When you initiate an extension to APS, what is your primary focus? Is it enhancing the syntax, improving tool support or expanding the vocabulary?
8. During the extension process, has the impact on usability and learnability been a consideration?
   - If yes, how has it impacted the development process?
   - If not, why do you think it has been overlooked?
9. How do you decide which extensions to prioritise for development within APS? E.g. out of proposed extensions, what priorities (such as technical feasibility, or alignment with future goals) go into choosing one extension over the others?
10. When extending APS, have you found the usage of regression testing via the Jenkins CI/CD a useful asset?
11. How have the Xtext and Xtend impacted your experience with the development of the APS language.
12. Do you get feedback on implemented extensions? How is this feedback evaluated and integrated into future extensions?

B.3 Interview Answers

You will find below in table B.1 and table B.2 the answers from all the conducted interviews. The two tables are parts of a whole, separated **on the horizontal axis** to be on 2 pages for legibility. Each column corresponds to a question asked as reflected in the list of questions above, whereas each row is representative of the answers of an interviewee. These results are discussed in Chapter 6.

1	2	3	4	5	6
Joined development	Both	Back end	Inconsistency introduced by different extensions of APS	Structural and domain-related	Back end function shadowing
Joined development	Writing	Back end	No such recollections due to rigorous impact analysis	Both	Ensuring consistency
From conception	Reading	Back end	Does not recognise such instances, rigorous process to avoid them	Structural	Extracting commonalities from different interfaces
From conception	Writing	Back end	Recalls several instances, linked to realisations keywords are not needed, or they are put in the wrong place	Both, depending on developer experience	Legacy issues, oversight during planning
From conception	Reading	Back end	Recalls changing implicit behaviours to be explicit and also needing to retroactively patch legacy support to some extensions	Structural	Interoperability between the DSL and other projects within the same context
High-level	Reading	Back end	No recollections, but anticipation of future such instances, as the tool becomes adopted	Technical issues were common in early stages, now predominantly structural	None come to mind
High-level	Reading	Back end	Legacy extensions not matching their structures; implicit legacy specs needed to be redone; knowledge of legacy code was lacking	Technical challenges are more common, but easier to address	Legacy issues

Table B.1: Aggregated, summarised interview answers (part 1)

7	8	9	10	11	12
Feature-driven	Yes	Based on direct necessity	Useful	Positive	No official feedback collection mechanism; sees value in such an investment
Feature-driven	Indirect	Responsibility of the architect	Useful	Positive	No feedback due to the current lack of end-users
Feature-driven	Yes	User demand (when delivered)	Useful	Positive	Feedback from external teams; no formal way to process it
Feature-driven	Indirect	Brainstorming	Useful	Positive	Only from developers, informal, recognises need for formalisation
Feature-driven; value-for-money	Yes	Alignment with future goals	Useful	Positive	No official feedback
Feature-driven	Indirect	Adding more functionality	Useful	Positive	Only from developers, informal, recognises need for formalisation
Feature-driven	Yes	Adding more functionality	Useful	Positive	A round every month from all the development teams; no end-user feedback

Table B.2: Aggregated, summarised interview answers (part 2)