Vrije Universiteit Amsterdam

Bachelor Thesis

Embedded Domain Specific Language: A Streamlined Approach for Framework Abstraction

---

Author: Daniel Berzak (2671724)

1st supervisor:	Animesh Trivedi
daily supervisor:	Matthijs Jansen
2nd reader:	-

A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science

August 18, 2023

Abstract

With the growing complexity of software systems, the development limitations using only general-purpose languages (GPLs) become evident. GPL development tends to be slow, error-prone, and difficult to grasp. Therefore, creating a domain-specific language (DSL) can be highly beneficial. However, DSLs are expensive, requiring domain and language development expertise, and are thus rarely used in solving software engineering problems.

This paper explores a different approach: using an Embedded Domain-Specific Language (EDSL). By embedding a DSL in a GPL, the proposed language can leverage the capabilities of existing tools developed for the GPL while still being tailored to suit a particular domain.

A qualitative study was performed with the use of past literary work in order to establish critical considerations to be made when analyzing a particular domain. These findings are then adapted to consider characteristics unique to EDSLs and turned into design guidelines. A use-case study was then performed in which the considerations and guidelines proposed were implemented in an EDSL in TypeScript in a real-world system called Continuum. Additionally, this paper explores the usage of VS-Code Snippets, a particular IDE tool that provides unique benefits, and how it can improve the usage and adoption of EDSLs. Finally, the final product was compared to a standardized approach currently prevalent in the domain of interest (e.g., configuration files).

The findings show that implementing and using a custom EDSL in a particular domain can improve its development's reliability, usability, productivity, and learnability, making it a viable tool worth consideration.

Contents

Chapter 1

Introduction

1.1 Context

To become an expert in a particular field, one would likely need to learn a dialect that conveys ideas relating to their domain of interest. This type of communication is a form of a domain-specific language (DSL), where the terms only have meaning within the scope of that specific field (or "domain"). In the context of computer systems, DSLs are languages made to improve the development process in a particular domain of interest. They abstract general concepts into simpler, more concise terms directly related to that domain. There are many widely used examples of DSLs, such as HTML, CSS, and SQL (1). Each of these languages has a unique syntax (or language) meant to improve the development process in their respective domains. By using such languages, domain experts need to spend less time learning how to interact with a particular system and more time on the problem at hand. By design, DSLs are much less capable than standard programming languages, also known as general-purpose languages (GPLs)(2), examples of which are C++, JavaScript, and Python. However, DSLs can significantly speed up software development and increase program accuracy(3). Creating a custom DSL comes with trade-offs. According to Freeman et al.,(4) DSLs are challenging to execute correctly. They are expensive to maintain and require additional syntax to be created and learned. The semantics, logic, and any additional development tools would need to be developed and documented from the ground up, requiring resources and time, making them less ideal. Therefore, most software engineering problems are solved primarily using GPLs. While all computable problems can be solved using only GPLs, the development process would be slow, error-prone, and difficult to comprehend when reading code written by others.

1.1.1 Embedding a Domain Specific Language

DSLs can only be used to describe ideas in a specific domain. GPLs can describe ideas in any domain but are more challenging to interact with. Considering these two things, it is reasonable to consider whether a GPL can be abstracted to describe a particular domain instead of creating an entirely new DSL. According to Freeman et al., (4) domain-specific features can be embedded into existing GPLs to take advantage of their features and development tools. In such solutions, domain experts would interact with simplified GPL code to perform tasks within their respective domains. We refer to such implementations as embedded domain-specific languages (EDSLs), as the domain-specific language is embedded within a general-purpose language(5). Notable examples of such languages are front-end JavaScript frameworks such as React, which abstracts JavaScript to ease user interface development or Django, which simplifies database development using Python. The main downside of using EDSLs over normal DSLs is that the language would be bound by specific rules and syntax dictated by the hosting language. However, leveraging the capabilities and tools made for GPLs using modern integrated development environments (IDEs) provides a unique set of advantages, which will be explored in detail in this paper.

1.1.2 Utilizing Visual Studio Code Snippets

Visual Studio Code (or VS-Code) is a conventional IDE. Its popularity is due to its wide range of tools and features, which assist developers in various ways. One popular tool it offers, "Visual Studio Code Snippets," allows developers to create time-saving shortcuts for inserting commonly used code segments. By leveraging VS-Code Snippets, program creators can increase productivity and enforce coding standards. In the context of developing an EDSL, these added benefits hold the potential for enhancing the development process and promoting EDSL adoption.

1.1.3 Software Configuration Management

Domain experts often require the ability to adapt how particular software instances are configured to suit their needs. This process becomes increasingly difficult to manage when dealing with complex systems as they tend to be highly configurable and contain many interconnected modules that need to communicate. This process has led to the Software Configuration Management (SCM) paradigm, which involves controlling change in computer systems.(6) While SCM has received notable research attention, the development

of an EDSL specifically for SCM is an area that has yet to receive much research attention. Exploring the development of an EDSL for SCM can provide domain experts with a more efficient and tailored approach to managing software configurations. By embedding domain-specific features within a GPL and leveraging the capabilities of modern IDEs, developers can significantly enhance the SCM of complex systems.

1.2 Problem Statement

Creating and maintaining custom DSLs can be resource-intensive and time-consuming while using general-purpose languages (GPLs) for domain-specific tasks can be slow and error-prone.

This research aims to address the development challenges introduced in complex domains. It will explore the concept of embedded domain-specific languages (EDSLs). This approach aims to leverage the capabilities offered by GPLs while still abstracting them to suit the domain of interest.

In particular, this paper will explore the benefits and drawbacks of using EDSLs to improve Software Configuration Management(SCM). The study will investigate the benefits and challenges introduced by EDSLs, and evaluate whether using them to define software configuration directly in code can provide significant advantages compared to more traditional configuration management approaches. Additionally, the study will propose design guidelines for developing an EDSL in complex systems and explore how implementing VS-Code Snippets can further assist domain-specific development.

1.3 Research Questions

The research questions addressed by this thesis and their motivation are:

RQ1 What are important considerations and design guidelines for creating an embedded Domain-Specific Language (EDSL) for complex computer systems? Implementing an EDSL can provide many benefits, but evaluating the specific domain before its development is crucial. This research question aims to identify vital factors to consider and establish design guidelines for developing an EDSL to improve interaction with complex systems. By analyzing domain-specific considerations and examining DSL design principles described in the literature, this research question aims to provide insights that will enhance the development process and effectiveness

of EDSLs. The goal is to prevent unnecessary development when EDSLs are not the optimal approach and promote informed decision-making regarding their design.

RQ2 What are benefits and drawbacks associated with designing and implementing an EDSL in the context of Software Configuration Management(SCM) compared to traditional approaches? By implementing an EDSL designed specifically for SCM, this research question aims to evaluate the practicality of an EDSL implementation in practice. The use case of the EDSL will provide insights into the amount of effort needed for its creation and analyze the benefits and drawbacks. Furthermore, the EDSL proposed is an open-source tool that can be adapted to custom configuration needs beyond the scope of this paper. This extends the potential benefits of the research to the broader community interested in SCM, providing a practical asset that can enhance the development process by increasing productivity and reducing errors and inconsistencies currently prevalent in SCM.

RQ3 In what ways can the utilization of Visual Studio Code snippets enhance the interaction between domain experts and an EDSL? This research question examines how Visual Studio Code snippets can facilitate a more efficient and intuitive utilization of an EDSL by domain experts. This research question aims to provide insights into the effectiveness of Visual Studio Code snippets in improving the interaction with an EDSL. In practice, the utilization of this tool will be explored by creating code templates that demonstrate the functionality of the EDSL implemented to answer **RQ3**.

1.4 Research Methodology

The primary objective of this paper is to lay the foundations for EDSL design and development for complex systems. Initially, we will perform a comprehensive **qualitative research methodology** with the use of past literary work(**RQ1**). The aim is to lay out the considerations that need to be made regarding any selected domain of interest one would intend on making an EDSL for. Doing so would provide an understanding of whether an EDSL is an appropriate solution for their use case without spending additional time learning what they are or how to use them. The literature review will be considered alongside the feature set provided by GPLs to create concrete guidelines for an EDSL design. **RQ2** and **RQ3** will be explored by studying a use case that involves applying the relevant concepts within a real-world open-source benchmarking framework called Continuum(7)

which will be explained in the following chapter. This methodology allows for a practical exploration of the research questions, leveraging the functionalities and features offered by Continuum to gain insights and draw conclusions.

1.5 Thesis Contributions

This paper aims to provide an analysis of the costs and benefits when using EDSLs. Additionally, it aims to define best practices and design principles to help future researchers and software engineers make better, more informed decisions when considering using EDSL. The main contributions of this paper are:

MC1 General EDSL Analysis, Conceptual - Provide a general overview of the appropriation of a DSL or an EDSL development depending on the domain of interest. The aim is to assist future engineers in better understanding what EDSLs can and cannot do and what they should keep in mind when considering them in the context of their particular domain of interest. Furthermore, we propose guidelines for future EDSL designs. The aim is to establish design principles and practices to provide a structured way of designing an EDSL and promote the field of research.

MC2 EDSL implementation, Artifact - As part of the research of this paper, we apply the concepts described to create a custom EDSL to assist with the configuration of an existing system. The code is open-source and is open to the public.

MC3 Implementation of code snippets, Artifact - As EDSLs are hosted in a GPL, their semantics have to match the ones of the hosting language. This could pose a difficulty for individuals with little to no understanding of the hosting language. By leveraging Visual Studio Code snippets implemented, this issue can be addressed. By typing predefined keywords, instances of fully functional configuration templates can be generated, learned, and used. The code snippets will be implemented in the same project as **MC3**.

1.6 Plagiarism Declaration

I confirm that this thesis work is my own work, is not copied from any other source (person, Internet, or machine), and has not been submitted elsewhere for assessment.

1.7 Thesis Structure

The thesis follows the following structure to address the research objectives: The introduction (Chapter 1) section describes the problem, research questions, and methodology. The background (Chapter 2) section provides insight into relevant topics addressed in this paper. The domain analysis (Chapter 3) explores existing technologies and requirements for using domain-specific languages. It proposes design guidelines to lay out the considerations for selecting an appropriate GPL and designing effective EDSLs using it. The implementation section (Chapter 4) showcases building an EDSL using Typescript on a real-life use case and evaluating the practicality of EDSLs. "Utilizing VS-Code Snippets" (Chapter 5) describes the addition of VS-Code Snippets to the EDSL proposed in Chapter 4 and the gained benefits for EDSL usage and adoption. Finally, the conclusion summarizes the findings, answers the research questions, and suggests future work on EDSLs.

Chapter 2

Background

This work explores the design and use of Domain-specific languages (DSLs) embedded within a General-purpose language (GPL). By design, DSLs require a particular domain for which they are tailored to improve the development process. This section provides an overview of the particular domain on which the research is conducted and related concepts. The domain this paper explores concerns itself with the configuration of a particular software system.

2.1 Configuration

Configuration refers to the way a program is set. Usually, this refers to characteristics that are adjustable between different instances of the same software (6). This aspect of software systems is referred to in the literature as software configuration management (SCM). The simplest way to handle SCM is to pass the different settings as Command-line arguments. However, this approach becomes increasingly hard to manage as software scales and would require typing all variables the program requires into the terminal upon every execution.

Another way of defining settings is using configuration files (files ending with .cfg). Configuration files are a commonly used way to describe how a program should operate. They are simple to use, easy to extend, and cheap to implement, making them an ideal tool for many software systems. While configuration files can benefit many domains, they are restrictive, can become long, and are highly error-prone, especially when dealing with highly configurable systems.

2.2 Continuum

In this paper, the use-case research that was performed aimed to answer **RQ3** and **RQ4**. It was applied to a particular software system called "Continuum." *Continuum* is a development framework that automates infrastructure deployment and benchmarking(7). This framework aims to assist domain experts (i.e., researchers and engineers) in testing and deploying different configurations of complex computer networks. It performs its tasks by emulating infrastructure according to settings passed to it by domain experts. Prior to this research, Continuum used configuration files to handle SCM. Continuum provided a template file (template.cfg) to assist domain experts by describing all possible configurations and their corresponding values.

The following code contains parts of the template file:

```
[infrastructure]
provider = qemu          # Options: qemu, gcp, baremetal (mandatory)
infra_only = False       # Options: True, False. Default: False

# Number of VMs to spawn per tier (should be at least 1 in total)
# If X_nodes > 0, then ...
cloud_nodes = 1          # Options: >= 0
edge_nodes = 1           # Options: >= 0
endpoint_nodes = 1       # Options: >= 0

# Number of cores per VM
cloud_cores = 4          # Options: >= 2
edge_cores = 2           # Options: >= 1
endpoint_cores = 1       # Options: >= 1
...
[benchmark]
resource_manager = kubernetes   # Options: kubernetes,  ...
resource_manager_only = False   # Options: True (only if ...) ...
docker_pull = False      # Options: ...
application = image_classification  # Options: ...
application_worker_cpu = 3.5       # Options: ...
application_worker_memory = 10.5   # Options: ...
...
```

All text following the # character are comments and are skipped when the configuration files are read. They are used in this context to describe the different variables' purpose

and expected values.

In order to create a new configuration, one would need to copy the variables needed from the provided template file into a new configuration file and execute Continuum, passing the path to the newly created configuration file as a command-line argument.

There are several disadvantages of using this approach:

1. **Cluttered documentation** - The template file was made in order to aid domain experts in learning what they can and cannot do when using Continuum with only one file. That said, the template file is long (almost 200 lines) and requires reading extensive comments making the configuration management process difficult to learn and time-consuming.

2. **No error-handling** - By design, configuration files are made to be flexible. While this means they can be used in any domain, they do not enforce any restrictions. Incorrect spelling, use of incorrect data types, or passing invalid values will all pass on to Continuum undetected. As Continuum offers many configuration options, the SCM becomes highly error-prone.

3. **Multiple configurations require multiple files** - Since configuration files are used to define unique key-value pairs, they are not convenient for defining multiple instances of objects. This attribute makes them restrictive when considering the execution of multiple configurations in a single execution.

The SCM of Continuum could greatly benefit from the DSL proposed in this research as it improves all of these drawbacks. In the context of this paper, we are considering the implementation of an embedded DSL, meaning a general purpose language(GPL) must be used to host the DSL.

2.3 General-Purpose-Languages & Development Tools

While they have been described before and are the convention of software development, general-purpose languages (GPLs) are only a subset of all programming languages. What differentiates them from other programming languages is that they are considered Turing-complete (i.e., they can be used to solve any computable problem). The continuous use of GPLs incentivizes developers to continuously create tools aimed at improving the development process.

The interpreter is a development tool that exists for most (if not all) widely adopted GPLs. It allows the Integrated development environment (IDE) to analyze written code and evaluate its syntax and type correctness. With that said, the way types are evaluated differs between GPLs.

2.4 Typing

"Typing" refers to how a GPL handles its data types. Some languages require developers to define the data types explicitly, while others expect them implicitly. There are notable advantages to both of these approaches. When a GPL is statically-typed, its variables' types must be explicitly defined. The explicit definition of variables is helpful when attempting to debug code. These types of languages are safer but less flexible. Contrary to statically-typed GPLs, dynamically-typed languages give more freedom to their developers. They do not require explicit type specification making their use quicker and easier. In most cases, dynamically typed GPLs infer data types from the values provided. In addition, using the same variable to store multiple data types is usually acceptable in dynamically-typed GPLs. Since statically-typed GPLs explicitly require developers to specify variable types, the GPLs interpreter can conclude when incorrect data is passed by performing a static analysis of the written code. This form of analysis is primarily found in statically-typed GPLs and is referred to as static type-checking. An EDSL made to improve the SCM of Continuum would greatly benefit from static-type checking as it would check the correctness of types without executing any code. While an EDSL may seem ideal for all use cases, drawbacks to this approach need to be considered.

2.5 I/O Streams

In many cases, GPL programs use I/O (Input/Output) streams to read and write data. I/O streams are primarily used to read or write to files and pass information between different running programs. When writing to a stream, the information needs to be encoded (or serialized) into bytes, and when reading, the information needs to be decoded (or deserialized) into usable data. Serialization is the process of writing an object to a stream, while deserialization is the process of rebuilding the stream back into an object(8). Unlike configuration files which can be parsed, GPL code can not be interpreted by other GPLs directly. Instead, the information it intends to provide can be encoded and written to output streams, from which a different program can read and deserialize it. In the context

of the EDSL for the SCM of Continuum, Continuum will observe the output stream for valid data and the error stream for any errors thrown by the EDSL. JavaScriptObjectModel (JSON) JSON is a standardized structure for serializing data and is the format in which data is passed from the EDSL to Continuum.

Chapter 3

Domain Analysis

Creating a domain-specific language(DSL) aims to improve the development process in particular domains. This is achieved by creating a syntax tailored to suit the domain of interest. In SQL, every operation concerns database interaction; HTML only contains objects that describe the order in which web components are laid out, and CSS only handles the design of those different components. Thanks to these DSLs, developers do not need to concern themselves with the underlying complexities and focus on the particular domain these languages are used for developing. While many domains can benefit greatly from a tailored programming language to interact with them, the added development costs and continuous maintenance they require might only be worth it for some use cases.

This chapter aims to aid software engineers in understanding the appropriation of an EDSL for their particular use case. By understanding existing technologies, EDSL quality requirements depending on the domain and considerations regarding the selection and use of the host GPL, Engineers can gain insight into whether an EDSL is worth the additional development costs they introduce. The following research aims to answer **RQ1** as it describes broad considerations before EDSL development. By considering existing technologies, efficient development can be achieved without creating a custom EDSL. Performing source-code analysis can aid in understanding particular complexities of the underlying implementation that the EDSL is made to improve. The following qualitative characteristics found in the literature can be used to establish fundamental EDSL requirements, which can later assist in selecting the host GPL and the development process. Lastly, a description of complexities introduced by the host GPL provides insight into unavoidable complexities that were not trivial before EDSL development. It is important to note that this paper concerns the implementation of an EDSL in the scope of a single, complex system, replacing existing means of interaction within a particular domain of that system.

All considerations proposed consider this assumption.

3.1 Consideration of Existing Technologies

The most fundamental consideration for creating an EDSL is whether it is worth the added development costs considering what already exists. By evaluating existing technologies and whether they can be used to effectively develop the domain of interest, one can conclude whether creating a custom EDSL is excessive. Furthermore, custom EDSLs must continuously adapt to new features and requirements added to their respective domain, making them costly in frequently changing domains. However, if the current development in the domain of interest fails to meet its expected requirements, creating an EDSL might still be worth consideration. Consider creating an EDSL for front-end web development. It may be excessive as JavaScript front-end frameworks like React offer a reach feature set and have a large community contributing to its further development. That would be challenging to compete with. On the contrary, as SCM is primarily concerned with complex systems, the SCM process becomes lengthy and error-prone, which makes it an appropriate domain for a custom EDSL.

3.2 Source Code Analysis

EDSLs are specialized languages that require their creators to understand how the domain of interest operates. If the EDSL intends to improve a particular domain of development (like React attempting to improve standard front-end web development), an extensive understanding of the host-GPL would be beneficial. However, this paper explores creating an EDSL for particular systems. In that case, there are different considerations to make. Firstly, the only code that needs to be analyzed is the parts of the system that the EDSL replaces or interacts with. In the context of Continuum, that only includes Configuration files and the parts of the Continuum code that parse and interact with them. Secondly, adapting complex systems to consider EDSL design choices might be challenging. It may require repetitive changes to existing structures and variables, a slow and error-prone process. This does not mean an EDSL creator can not make personal design choices regarding naming and structure, but rather that those choices may need conversion to the structure that the particular system they interact with expects to receive. This notion directly applies to Continuum, where the structure of the EDSL differs from the structure that Continuum expects and is therefore adapted accordingly. Alternatively, one could

change the project's source code to abide by the EDSL's structure. While this approach will likely require more work, it will reduce the computation overhead of the EDSL at run-time.

3.3 Requirements Engineering

According to Kahraman and Bilgen(9), a broad measure of agreed-upon characteristics can be used to assess a DSL. They are based on experiences and publications on the topic. In this paper, we apply these notions to an EDSL implementation. The most notable characteristics are:

1. **Functional suitability:** The extent to which the EDSL is developed. The EDSL should offer all the necessary functionality of the domain.

2. **Usability**: The degree to which a domain expert can achieve their needed goals. The topic of usability is quite broad, but in simple terms, it refers to ease of use. EDSL syntax should be simple to understand and use. With that said, EDSLs are strictly bounded to the rules and syntax of the host GPL. A dynamically typed GPL would provide a simple-to-use syntax that can benefit usability. A standard DSL might be more appropriate for creators who want exact control over their language's syntax and rules.

3. **Reliability**: The EDSL should aid in producing reliable programs. According to Kärnä et al., (10), error prevention and model checking as considered significant quality. For EDSLs, reliability can be achieved relatively easily using type safety features in statically type GPLs.

4. **Maintainability & Extensibility**: modifying or adding functionality should require as little effort as possible. This can be achieved in an EDSL by creating reusable code constructs that encapsulate repetitive patterns found in the domain. Additionally, EDSLs are likely to be easier to maintain and extend to individuals with experience with GPL development than standard DSLs. New functionality can be achieved by only extending the existing code structures and variables.

5. **Productivity**: A Good EDSL provides domain experts with resources and tooling that promote development speed. Many development tools already exist since an

EDSL is hosted in a GPL. Using these tools could aid productivity. EDSL productivity can be extended further using custom IDE tools (such as custom code snippets).

These qualitative characteristics will describe the fundamental EDSL requirements, continuously considered during development. Their priority against one another depends on the specific needs of the domain of interest. Some domains value productivity and ease of use, while others require extensive error detection. The EDSL's creator would need to investigate the primary concerns that arise from using existing technologies and how an EDSL could assist in solving them. Additionally, these characteristics will be used to evaluate the EDSL's effectiveness in improving the domain of interest compared to the previously existing approach.

3.3.1 GPL Considerations

When creating an EDSL, the choice of the host-GPL used should be carefully considered. Each GPL comes with its own set of features. While the goal of creating an EDSL is to create a custom language, it will still need to abide by the syntax of the host language. In cases where domain experts are likely to make mistakes, using a statically-typed GPL would be helpful. It is more likely to help detect errors before and after run-time. On the other hand, in cases where productivity is of higher priority or domain experts are less technical, a dynamically typed language, which offers simple syntax, might be deemed most appropriate.

Ideally, the first GPL to consider for the EDSL implementation is the one used to develop the code that the EDSL aims to abstract. By doing so, the EDSL and the program can communicate directly by acting as different code modules of a single project. If a different GPL is chosen, handling the communication between different GPLs will be an additional requirement.

3.3.2 Cross-GPL communication

EDSLs consist entirely of GPL code. If the EDSL and the program it is built to interact with are developed using the same programming language, communication between the two can be resolved by simply importing the EDSL's constructs. With that said, There is no strict requirement for the EDSL and the program it is built for to use the same GPL. Using the same GPL would ease communication between the two; EDSL creators may see

a different GPL more fitting for the needs of their DSL. In that case, creating means of communication between those two code bases is required.

Serialization & Deserialization

Serialization is the process of writing an object to a stream, while deserialization is the process of rebuilding the stream back into an object(8). Since the communication between the EDSL and the program would be done through streams, serialization is needed. There are many standard formats for serialization. Commonly used examples are XML and JSON. Which serialization method is best is out of the scope of this paper.

Writing to Temporary Files

In many cases using temporary files to pass information between different programs is quite common. It is simple to implement. The problem with this approach is that it is unorganized, might require multiple files (for standard output and error logs), and would need to account for old files that are no longer relevant.

EDSL as a Sub-process

A more elegant approach is to read the output and error streams of the EDSL in the main program. This can be accomplished by executing an EDSL file as a sub-process and ensuring it outputs to the error stream upon receiving an error; otherwise, it outputs to the output stream. While it is more complicated to implement than writing the data to files, it usually only requires a few lines of code.

3.4 EDSL Guidelines

Once understanding the domain requirements and what needs complexities EDSLs introduce, the design process of an EDSL process can commence. When creating a DSL, the overall design, syntax, and structure can be made depending on the creator's preferences. In HTML, every element is written with angle brackets (<>) around it. SQL only uses natural language terms to describe its operations, and CSS takes a more traditional coding structure using curly braces ({}) around its objects. That said, according to past research, certain guidelines can be followed regarding the design of DSLs that can greatly enhance their usage.

The following guidelines provide qualitative requirements specific to the implementation of an EDSL. To implement an EDSL, the particular domain that is worked on needs to be

well understood. DSLs are use-case specific and will only provide an optimal solution if they are developed to suit their particular use case. That is why the following guidelines should be considered following an extensive analysis of the domain of interest and EDSL complexities described in the previous sections.

3.4.1 GPL Selection

Unlike standard DSLs, EDSLs inherit a substantial part of their feature set from the selected GPL they use as their host. This means that in the context of EDSLs specifically, GPL selection becomes an integral part of the design process. While the host-GPL can be abstracted significantly to tailor particular domains, they will remain bound to their GPL's rules and syntax. This makes the selection of the GPL a crucial part that directly relates to the design process, as the host-GPL selected will directly impact the syntax, error handling, and behavior of the EDSL. The following guidelines aim to assist in GPL selection:

1. **SG1 - Syntax and Error Detection**: Select a host GPL that complements the domain requirements. Consider statically-typed GPLs for improved error handling or dynamically typed GPLs for simplicity and ease of use.
2. **SG2 - Integration and Communication**: Ideally, choose the GPL used in the domain of interest as the host language of the EDSL. This would facilitate seamless integration and direct communication between the EDSL and the domain's source code. Consider the additional requirements and challenges associated with communication between different GPLs described in the previous chapter when using a different GPL.

Once a GPL is selected, implementation of the EDSL can begin. Domain-specific requirements need to be kept in close consideration. However, additional design guidelines can be used to enhance the EDSL design further. The following design guidelines aim to provide vital considerations regarding the design of an EDSL.

1. **DG1 - Reliability**: Focus on producing reliable programs using the EDSL. Implement error prevention and model checking to minimize potential errors and ensure the correctness of the generated code.
2. **DG2 - Functional Suitability**: Ensure that the EDSL offers all the necessary functionality required for the specific use case. Identify the core requirements and design the EDSL to fulfill them effectively.

3. **DG3 - Productivity**: Enhance domain experts' productivity by providing appropriate resources and tooling. Develop the EDSL to support fast development speed, allowing users to express their intent concisely and effectively.

4. **DG4 - Simplicity**: This notion relates to usability. Prioritize ease of use for domain experts. Design the EDSL's syntax to be simple, intuitive, and easily understandable(11). Consider the target audience's knowledge and technical expertise to create a user-friendly experience.

5. **DG5 - Descriptiveness**: Creating clear and descriptive semantics will likely prevent misinterpretation of concepts and assist in learnability and comprehensibility of the EDSL(11).

6. **DG6 - Consistency**: Using the same look-and-feel across the EDSL implementation can improve the understandability and productivity of its usage. By doing so, domain experts can gain an intuition to use features of the EDSL based on their knowledge of other ones(11).

7. **DG7 - Maintainability & Extensibility**: Aim to minimize the effort required to update or extend the EDSL by creating reusable code constructs, enabling developers to adapt to changing requirements efficiently.

3.5 Summary

In conclusion, the chapter provides valuable insights for software engineers considering the adoption of an EDSL for their specific domain. It emphasizes the importance of evaluating existing technologies, understanding source code behavior, establishing fundamental EDSL requirements, considering the host GPL, and implementing effective cross-GPL communication. These considerations can be used to evaluate whether an EDSL would benefit a particular use case. The purpose is to determine whether the benefits outweigh the associated costs for a particular domain. The EDSL guidelines proposed aim to capture the domain requirements and additional qualitative characteristics into concrete guidelines to be effectively applied. By doing so, we streamline the EDSL development process into concise considerations that are easy to follow and be applied effectively.

Chapter 4

EDSL Implementation in Typescript

This chapter's purpose is to apply the proposed considerations and guidelines from the previous chapter to a real-world use case. More specifically, the implementation proposed will replace the existing conventional method of using configuration files in the Continuum framework. To design and implement an EDSL, we will follow the steps described in the previous sections.

4.1 Consideration of existing technologies

Configuration files are the current means by which Continuum receives data to fulfill its purpose. Continuum expects its executor to pass the path to the used configuration file as the first command line argument.

Continuum is a highly configurable, heterogeneous system(7). While configuration files can be a handy tool to perform SCM at a relatively low cost, they become highly error-prone when used in highly configurable systems such as Continuum, as configuration files perform no validation on the data they receive. As a result, even a single character missing from a variable can cause unexpected behavior at run-time. Another way to consider when passing information to Continuum is command-line arguments. However, they introduce the same problems as configuration files and are even harder to manage.

Existing technologies do not offer an appropriate solution for this domain. Continuum SCM could be faster, less error-prone, and easier to learn.

4.2 Source-Code Analysis

This section aims to establish an understanding of Continuum's source code. While it can be beneficial to understand how Continuum's source code is implemented, we only need to

consider the parts of the code that the EDSL intends to replace. 2 sections of Continuum's code need to be considered:

1. **Configuration files** used in Continuum, their structure, and variables. This is the domain that the EDSL aims to replace.

2. The parts of Continuum's **GPL code** (written in Python) are responsible for parsing the configuration files and any other part of the code where the parsed configurations are accessed.

These parts of Continuum's code need to be broken down and understood so they can be replaced accordingly.

4.2.1 Configuration Files

The following table describes some of the options offered by Continuum that can be passed to it using configuration files. For a more extensive overview of how Continuum handles configuration, refer to the background (Chapter 2) for a more in-depth explanation.

Table 4.1: Variable Definitions in Continuum configuration

Variable Name	Options	Mandatory	Default Value
provider	qemu, gcp, baremetal	Yes	-
infra_only	True, False	No	False
cloud_nodes	Integer, x >= 0	Yes	0
cloud_cores	Integer, x >= 2	Only if cloud_nodes > 0	-
cloud_memory	Integer, x >= 1	Only if cloud_nodes > 0	-
cloud_quota	Float, 0.1 <= x <= 1	Only if cloud_nodes > 0	-
endpoint_nodes	Integer, x >= 0	Yes	0
endpoint_cores	Integer, x >= 1	Only if endpoint_nodes > 0	-
endpoint_memory	Integer, x >= 1	Only if endpoint_nodes > 0	-
endpoint_quota	Float, 0.1 <= x <= 1	Only if endpoint_nodes > 0	-

Table 4.1 shows different variables of the Continuum configuration. The variable "provider" expects 1 of 3 possibilities. Other variables, such as "cloud_quota," expect a numeric value within a specific range. In practice, this means that in this context, handling precise values of both strings and numbers can aid in creating reliable configurations.

4.2.2 GPL Code

While Continuum expects to receive information in a particular format to operate correctly, it is written in Python. As Python is dynamically typed, Continuum does not perform static type-checking prior to execution. Errors can only be detected at run-time or not at all. Due to the resource-intensive nature of Continuum, its incorrect execution would be highly wasteful and yield no benefit. If Continuum could leverage, static analysis features that are provided by statically-typed GPLs, finding mistakes both prior and during execution becomes much more likely. This would improve both the reliability and productivity of the configuration process significantly.

After careful observation of Continuum's configuration handling, it becomes evident that relying solely on configuration files lacks sufficient error-handling capabilities. Additionally, Continuum using a dynamically typed GPL makes it less optimal for validation and type-checking than statically typed programming languages.

4.3 Requirements Engineering

Once done with a domain analysis, The requirements described in Chapter 3 can be used to describe the domain's requirements. These requirements and their motivations are:

1. **Reliability - Domain experts need to be able to produce reliable configurations.** In the context of large-scale and complex systems, configuration errors are among the dominant causes of failure(12). Therefore, ensuring the exact values of variables are correct is the most fundamental requirement of creating an EDSL in the domain of SCM. This is especially true due to the lack of error detection and model checking in configuration files.

2. **Functional suitability - Domain experts should be able to define most (if not all) configuration options Continuum has for benchmarking.** In the context of network infrastructure deployment at a large scale, many features need to be taken into account. While handling every variable in Continuum in the EDSL would be time-consuming, an implementation that only partially incorporates the functionality of Continuum might make the EDSL incompatible with defining certain critical configurations.

3. **Maintainability & Extensibility - Making modifications to the EDSL should not require much effort.** The continuous advancement of technology on the web allows researchers to find innovative solutions that render older technologies obsolete. As Continuum operates in the domain of complex networks, it will need to adapt to new trends and approaches continuously. The DSL should be able to continuously adapt to the changes in Continuum without requiring much development effort.

4. **Productivity - Domain experts should be able to define configurations efficiently.** Configurations of complex systems can be long and repetitive. This means there is room to greatly enhance productivity by understanding those repetitive patterns and streamlining their creation.

5. **Usability - Domain experts should be able to define configurations with ease.** While it is always helpful to have an intuitive means for performing work, other quality requirements are more crucial in the scope of network benchmarking. Additionally, individuals performing network benchmarks are likely to come from a technical background and are, therefore, more likely to be able to use a less "user-friendly" syntax.

4.4 Applying EDSL Guidelines

At this stage, the EDSL guidelines can be applied to this particular EDSL implementation. Doing so will establish a more concrete description of what needs to be implemented. This process will be done considering the domain requirements to make them more specific for an EDSL implementation in particular.

4.4.1 Applying the GPL Selection Guidelines

In Chapter 3, two design guidelines were proposed to assist with selecting the host GPL. The following describes their application in the domain of Continuum SCM:

1. **SG1 - Syntax and Error Detection** - In the context of Continuum, extensive error handling can substantially assist its operations. Additionally, domain experts in the domain of network benchmarking are likely to be highly technical, which makes the simplicity of the syntax less critical. Therefore, the proposed host-GPL should be statically typed and provide extensive error handling and type safety features.

2. **SG2 - Integration and Communication** - Choosing Python, the GPL used to implement Continuum, for the EDSL implementation would provide seamless integration between Continuum and the EDSL. With that said, Python is dynamically-typed and does not provide the error handling and type safety that is required for this domain. For that reason, it would be ideal to consider other GPLs instead. As the EDSL and Continuum will be built using different GPLs, their communication will need to be handled.

Examples of statically typed GPLs that can be considered for an EDSL implementation are Java, C#, or TypeScript. For the implementation of the EDSL, Typescript was specifically chosen as the host-GPL. TypeScript stands out from other statically typed GPLs due to its unique data-typing and type-checking capabilities. Using those features, the EDSL can ensure the correctness of configurations and greatly enhance the EDSL's reliability.

4.4.2 Applying the Design Guidelines

The following code was taken from the EDSL implemented in Continuum (generics.ts file):

```
export type NodeMap = {
    cloud: number,
    edge: number,
    endpoint: number,
}
export type ReadWriteSpeed = {
    readSpeed?: NodeMap;
    writeSpeed?: NodeMap;
}
...
```

```
export type InfrastructureConfig = {
    provider: "qemu" | "GCP" | "baremetal",
    infraOnly?: boolean,
    nodes: NodeMap;
    cores: NodeMap;
    memory: NodeMap;
    quota: NodeMap;
```

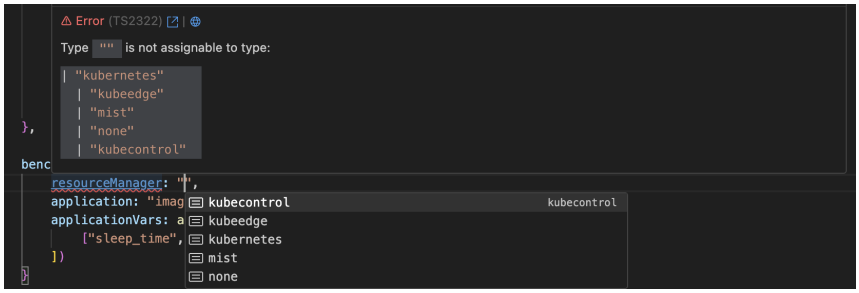

Figure 4.1: Specific Value Types handled in TypeScript When Defining a Configuration

```
    ...
    wirelessNetworkPreset?: "4g" | "5g"
    ...
    executionMode?: "openfaas"
}
```

This code will be used to demonstrate how different design guidelines were applied.

1. **DG1 - Reliability**: In this code, TypeScript Type Aliases and specific data types are used to increase the reliability of the SCM process. Denoted by the **"type"** keyword, Typescript Type Aliases are used to define object structures concisely. Typescript also allows the specification of the data types of the different members of the object, adding a layer of type enforcement. Additionally, TypeScript has a unique capability of defining a finite set of values instead of the type of the variable. These values would be checked using static type-checking and detected without running any code. In the "InfrastructureConfig" Type Alias, there is a variable "provider ." Denoted in **Table 1** provider can only be one of three possible strings. Therefore it is defined using the three possible strings with the "|" character in between. Doing so will present an error if any other value is given. The "NodeMap" Type Alias only accepts three variables, nodes, cores, and endpoints, which are all numbers. This adds a layer of type checking, increasing reliability further.

   The variables **"provider"**, **"wirelessNetworkPreset"**, and **"execution mode"** are all examples of variables using specific values as their types. If any other value is specified when defining them, it will be detected, and an appropriate error message will be provided. Additionally, by typing in quote marks, the IDE will automatically suggest this variable's possibilities. Both of these features can be seen in **Figure 4.1**, which shows how **"resource manager"** can only be one of five possibilities.

2. **DG2 - Functional Suitability**: This is achieved by implementing all of the configuration options demonstrated in the configuration template provided by Continuum in the EDSL. This allows the EDSL to replace the current implementation using configuration files completely.

3. **DG3 - Productivity**: As shown earlier, Continuum expects domain experts to refer to a template configuration file to understand its possibilities. When certain variables are not assigned but are needed, there are pre-defined default values that Continuum automatically assigns to them. The problem with this approach is that domain experts need to continuously refer to this template file to understand what values are assigned at what stage. In an EDSL, this can be solved with a class constructor. In the constructor, default values can be assigned directly in the EDSL, meaning domain experts can print out the configuration they have made and observe its default values. This added feature assists both reliability and productivity in the configuration management process. In practice, this is achieved by using nullable types. In the code above, "readSpeed" and "writeSpeed" have a question mark next to them; this indicates to TypeScript that these variables are not required and will be null if not specified. All nullable variables are then checked in the constructor of the primary class, "Configuration," which combines all EDSL functionality into a single object. The following code is a part of the Configuration class constructor:

```
constructor(config: ConfigurationMap) {
    ...
    this.infrastructure.readWriteSpeed =
        config.infrastructure.readWriteSpeed != null ?
    {
        readSpeed: config.infrastructure.readWriteSpeed.readSpeed != null
            ? config.infrastructure.readWriteSpeed.readSpeed
            : { cloud: 0, edge: 0, endpoint: 0 },
        writeSpeed: config.infrastructure.readWriteSpeed.writeSpeed != null
            ? config.infrastructure.readWriteSpeed.writeSpeed
            : { cloud: 0, edge: 0, endpoint: 0 },
    } : {
        readSpeed: { cloud: 0, edge: 0, endpoint: 0 },
        writeSpeed: { cloud: 0, edge: 0, endpoint: 0 }
    }
    ...
```

```
}
```

This code demonstrates how the EDSL checks whether different values are provided (i.e., not equal to null) and assigns pre-defined default values if not. "readWriteSpeed" uses the Type Alias "ReadWriteSpeed," which was shown in the previous code listing provided.

4. **DG4 - Simplicity**: A class called "Configuration" was defined for simplicity. It handles the creation validation and output of created configurations. This means that all EDSL functionality can be handled using a single object. The following code demonstrates the definition of a configuration class:

```
    const configuration = new Configuration({
    infrastructure: {
        provider: 'qemu',
        nodes: { cloud: 0, edge: 1, endpoint: 1 },
        cores: { cloud: 1, edge: 1, endpoint: 11 },
        memory: { cloud: 2, edge: 1, endpoint: 1 },
        quota: { cloud: 0.4, edge: 0.3, endpoint: 0.5 },
    },
    benchmark: {
        resourceManager: "kubernetes",
        application: "empty",
        applicationVars: applicationVars([
            ["sleep_time", 60],

        ])
    }
})
configuration.validate()
configuration.output()
```

The code showcases how a configuration is defined within a "Configuration" object called "Configuration." That same object is validated with the "validate" function and provided to Continuum using the "output" function. Both functions are part of the Configuration Class, making the interaction more straightforward, requiring only calling functions that are members of this same class. While this provides relative

simplicity, The syntax is relatively complex compared to traditional DSLs. The strict syntax of TypeScript restricts the extent to which the interaction can be simplified.

5. **DG5 - Descriptiveness**: The configuration instance in the last code listing shows a basic example of the configuration class. The class design aims to be descriptive of the purpose of the different features it has to offer. This is achieved by providing meaningful names that describe the meaning of the variable. Hovering over the different members of the class demonstrates what structure they expect, assisting in the descriptiveness of the configuration.

   While using TypeScript for an EDSL implementation provides significant benefits, it is still a GPL. Because of this, some complexities are not possible to avoid entirely. An example of this is the case of defining application variables. For its execution, Continuum requires to be able to receive any user-defined key-value pairs which act as local variables for the application it intends to run benchmarks on. This makes a map of key-value pairs the most appropriate data structure. With that said, TypeScript can not serialize maps, resulting in all variables being discarded. Instead, the map needs to be converted to an object. While it is possible to refer to the specific function that does this operation, it is hard to memorize and not very elegant. Instead of tailoring this to the particular domain, the function "applicationVars" takes a list of key-value pairs and casts it to an object; this gives a more user-friendly solution (See code listing below). It can be seen in the previous code listing how "applicationVars" gets defined when creating a new configuration.

   ```
   const applicationVars = (variables: Iterable<[PropertyKey, any]>) =>
       Object.fromEntries(variables)
   ```

6. **DG6 - Consistency**: Most complex features within the configuration class use Type Aliases with a similar structure. It can be seen in the previous code listing that nodes, cores, memory, and quota are all defined using an identical structure, making the configuration process highly consistent.

7. **DG7 - Maintainability & Extensibility**: Generic data structures can be created accordingly by observing reoccurring patterns in the configuration process. By doing so, adding features with a similar structure becomes simpler during implementation and later extension of the EDSL. **Table 1** showcases a pattern that appears

throughout the configuration process. That pattern is that many configuration options have cloud, endpoint, and edge options. This reoccurring pattern would be much easier to manage using a Type Alias. In the code listing at the top of this section, it can be seen that these three variables are abstracted into a Type Alias called **"NodeMap"**. This approach can be extended to create more complex objects as well. The code listing at the top of this section also shows the Type Alias **"ReadWriteSpeed"**, which consists of two NodeMaps. Below readWriteSpeed, a Type Alias called **"InfrastructureConfig"** encapsulates all Type Aliases in the EDSL. By doing so, InfrastructureConfig describes the entire infrastructure of Continuum using a single construct.

Another way to improve DG7 is custom error handling. In the context of Continuum's Configuration, handling errors becomes a repetitive process that would benefit the creation of reusable code. Therefore, a Type Alias called **"Validator"** was created. Its purpose is to abstract the notion of throwing errors into concise terms. Additionally, **"successValidator"** and **"errorValidator"** are two functions that were created to abstract error-tracking further. Both functions return appropriate Validators that can later be assessed by the **"checkValidator"** function. The function outputs to the error stream and terminates the program if it detects an error (See code listing below).

```
export type Validator = {
    success: boolean,
    errorMessage: string
}

export function successValidator(): Validator {
    return { success: true, errorMessage: "" }
}

export function errorValidator(error: string): Validator {
    return { success: false, errorMessage: error + " " }
}

export function checkValidator(validator: Validator): void {
    try {
        if (!validator.success) {
```

```
            throw validator.errorMessage
        }
    } catch (errorMessage) {
        console.error(("Error: " + errorMessage))
        process.exit(1)
    }
}
```

4.4.3 Cross-GPL communication

Chapter 3 provides an analysis of what considerations need to be made prior to creating an EDSL. These considerations need to be taken into account when selecting a GPL for the EDSL that is different from the one used in the domain of interest. In this particular implementation, TypeScript was used for the EDSL, while Continuum uses Python. This means that the communication between the 2 needs to be handled.

Serialization & Deserialization

For simplicity, JSON was chosen to serialize the configuration provided in the EDSL. As Typescript is an extension of JavaScript, JSON serialization takes relatively little effort to implement. Any serialization format can be used as long as both GPLs can deserialize it. Additionally, a function called **toJson** formats the data to adhere to the way Continuum expects it directly.

EDSL as a Sub-process

Once the EDSL serializes the data, it can be transferred to Continuum. Before the EDSL implementation, Continuum expected to receive the path leading to a configuration file. That file was later parsed into an object and used accordingly. The problem with using an EDSL is that while it can serialize the data to a format readable by Continuum, it can only do so by being executed. In other words, Continuum needs to execute the EDSL code. To handle this, Continuum will execute the EDSL as a separate process (a sub-process) and read its output. This approach stands out because both valid and invalid configurations can be handled in this step. That is because Continuum has access to both the output stream and the error stream. The error stream will contain any errors that were thrown by the EDSL. This means that it will not contain anything when the configuration is valid. In practice, if there is an error in the EDSL, an error needs to be thrown in Continuum as

well. Otherwise, Continuum should deserialize the JSON provided in the output stream and use it as the provided configuration.

4.5 Summary

This chapter focused on applying the considerations and guidelines from the previous chapter to a real-world use case of developing an EDSL for SCM in the context of Continuum. The implementation aimed to replace the existing conventional method of using configuration files in Continuum. The chapter started by discussing the limitations and drawbacks of using configuration files in highly configurable systems like Continuum. It highlighted the need for an EDSL that provides a more reliable and efficient approach to managing software configurations. Next, the source-code analysis of Continuum was presented, focusing on the configuration files and the relevant parts of the GPL code responsible for parsing and accessing the configurations. Understanding these components was crucial for designing the EDSL to replace them effectively. Requirements engineering was then applied to define the specific requirements for the EDSL in the Continuum SCM domain. These requirements included reliability, functional suitability, maintainability, extensibility, productivity, and usability. The chapter proceeded to apply the EDSL guidelines to the implementation.

Cross-GPL communication was also addressed, considering the serialization and deserialization of data between the EDSL and Continuum. JSON formatting was chosen for simplicity, and the EDSL was executed as a sub-process within Continuum, allowing Continuum to read both the output and errors provided by the EDSL directly.

In summary, this chapter applied the proposed considerations and guidelines to develop an EDSL for SCM in the context of Continuum. The implementation aimed to enhance the reliability, efficiency, and flexibility of managing software configurations, providing domain experts with a more tailored and productive approach.

Chapter 5

Utilizing VS-Code Snippets

Implementing an EDSL can yield substantial benefits and assist in creating better means of interacting with complex computer systems. However, EDSLs come with their learning curve and challenges. The most common way to solve this is to create documentation describing all of the EDSL's functionalities. While that is a solution, learning through documentation is slow and requires continuous switching between the configuration process and the documentation they use. This paper proposes using a different methodology using VS-Code Snippets. Using this IDE tool, configuration templates can be generated using custom keywords. Due to the nature of the configuration process, this can be used to enhance the learning process for domain experts. They would learn how to use the EDSL from concrete examples of configurations. Additionally, it will force them to adhere to the expected structure intended by the EDSL's creator. This adds significant value to the productivity, usability, and reliability of the EDSL and is therefore considered an integral part of what this research aims to explore.

5.1 Creating Snippets

Code Snippets can be defined globally(machine-specific) and locally(project-specific). They should be defined locally to make the most out of using snippets. That way, up-to-date Snippet versions can be distributed directly with the software, making it easier to extend.

Figure 5.1 shows an example of a code snippet used in Continuum. The text at the top level of the structure (in this case **"Create Basic Configuration"**) is used to indicate to snippet creators what that particular snippet is used for. The **"scope"** variable requires specification of which GPLs should have access to the snippets. **"prefix"** is the keyword that needs to be typed to generate the code, and the body should contain the code to

```
"Create Basic Configuration": {
    "scope": "typescript",
    "prefix": "cfg_basic",
    "body": [
        "//TODO: press tab to quickly fill in placeholder values. NOTE: make sure only one configuration instance prints at a time",
        "const ${1:newConfiguration} = new Configuration({",
            "\tinfrastructure: {",
            "\t\tprovider: '${2|qemu,gcp,baremetal|}',",
            "\t\tnodes: { cloud: ${3:0}, edge: ${4:0}, endpoint: ${5:0} },  // Options: x >= 0",
            "\t\tcores: { cloud: ${6:0}, edge: ${7:0}, endpoint: ${8:0} }, // Options: cloud >= 2, edge & endpoint >= 1 (each)",
            "\t\tmemory: { cloud: ${9:0}, edge: ${10:0}, endpoint: ${11:0} }, // x >= 1",
            "\t\tquota: { cloud: ${12:0}, edge: ${13:0}, endpoint: ${14:0} }, // Options: 0.1 <=x <= 1.0",
            "\t},",
            "\t//if infra only is set to true benchmarkConfig can be removed",
            "\tbenchmark: {",
                "\t\tresourceManager: \"${15|kubernetes,kubeedge,mist,none,kubecontrol|}\",",
                "\t\tapplication: \"${16|empty,image_classification|}\", // has to correspond to an existing application module",
                "\t\tapplicationVars: applicationVars([",
                    "\t\t\t[\"sleep_time\", 60], //variable in the sleep application",
                    "\t\t\t// key value pair syntax: [\"frequency\", 5]",
                "\t\t])",
            "\t}",
        "})",
        "",
        "${1:newConfiguration}.validate()",
        "${1:newConfiguration}.output()",
    ],
    "description": "Creates basic instance of configuration class"
},
```

Figure 5.1: Snippet Used by the EDSL to Create a Basic Configuration for Continuum

be generated. Lastly, the description adds a short explanation for snippet users that will appear every time the prefix is typed.

The body takes a list of strings acting as the generated code's rows with a new tab character (\t) used to indent code in certain places.

5.1.1 Placeholders

By placing text in the following format, it becomes a placeholder.

```
${number:placeholderName}
```

Once the snippet code is generated, the different placeholders can be traversed by pressing the tab key and filled in accordingly. They are traversed in ascending order by using the "number" value, which needs to be an integer. This addition can significantly assist in quickly creating configurations and the productivity of snippet usage.

5.1.2 Finite List selection

Another way to improve snippets' productivity is to provide a finite list of options in the following format.

```
${number|option1, option2, option3|}
```

When getting to placeholders of this particular format, a list of possibilities will be displayed. Creating this placeholder type is helpful for cases requiring a finite set of options.

5.2 Configuration Using Snippets

When typing the prefix term, it generates a basic configuration and adds function calls that validate and output it to be used by Continuum. Without VS-Code Snippets, this process would require a thorough understanding of the EDSL and would take a few minutes. As a result of using Snippets, domain experts require little to no understanding of the EDSL to create a functioning configuration, and the entire process can take only a few seconds. This is a substantial difference, especially when considering cases where multiple configurations need to be defined and executed. Therefore, the EDSL significantly benefits from implementing VS-Code Snippets.

Chapter 6

Evaluation

In this section, we will evaluate the implementation of the EDSL in Typescript for Continuum SCM based on the goals and expectations set. The evaluation will focus on the particular domain requirements and how well they are addressed. Additionally, this evaluation will consider the added benefits of using code snippets for generating Configurations.

6.1 Experimental Setup

In order to set up an experiment that demonstrates the effectiveness of the EDSL, it will be directly compared to the conventional method of using configuration files used in Continuum prior to its implementation. By assessing these two approaches using the domain-specific requirements described in Chapter 5 benefits and drawbacks of the EDSL implementation can be concluded.

6.2 Assessing Domain Requirements

The following requirements describe the qualitative requirements of the domain of the SCM process in Continuum. They are ranked from top to bottom in importance. In each section, a comparison between the two methodologies described in the experimental setup will be made.

6.2.1 Reliability

Figure 7.1 and **Figure 7.2** are identical configurations. One was built using standard configuration files, and the other used the EDSL. They both contain the same errors:

1. The provider needs to be one of three pre-defined strings

```
[infrastructure]
provider = invalid1

cloud_nodes = invalid2
cloud_cores = 4
cloud_memory = 4
cloud_quota = 1.0

endpoint_nodes = 1
endpoint_cores = 1
endpoint_memory = 1
endpoint_quota = 0.5

network_emulation = True
wireless_network_preset = 4g

middleIP = 160
middleIP_base = 161

[benchmark]
resource_manager = invalid3

application = image_classification
frequency = 5
```

Figure 6.1: Invalid Configuration using configuration file

2. The number of cloud nodes needs to be an integer

3. The resource manager needs to be one of five pre-defined strings

Static Type-Checking

Since the EDSL can leverage the capability of TypeScript's advanced type system, all three errors are detected by static type-checking, notifying their creator that something is incorrect. On the other hand, the configuration file does not detect anything prior to run-time. This means that when using the EDSL, all mistakes can be instantly corrected to the appropriate values. Additionally, by hovering over the variable or retyping the quote marks, TypeScript will list all the correct possibilities. This is demonstrated in **Figure 7.3**.

```
const configuration = new Configuration({
    infrastructure: {
        provider: 'invalid1',
        nodes: { cloud: "invalid2", edge: 0, endpoint: 1 },
        cores: { cloud: 4, edge: 0, endpoint: 1 },
        memory: { cloud: 4, edge: 0, endpoint: 1 },
        quota: { cloud: 1.0, edge: 0, endpoint: 0.5 },
    },
    benchmark: {
        resourceManager: "invalid3",
        application: "image_classification",
        applicationVars: applicationVars([
            ["frequency", 5],
        ])
    }
})

configuration.validate()
configuration.output()
```

Figure 6.2: Invalid Configuration using EDSL

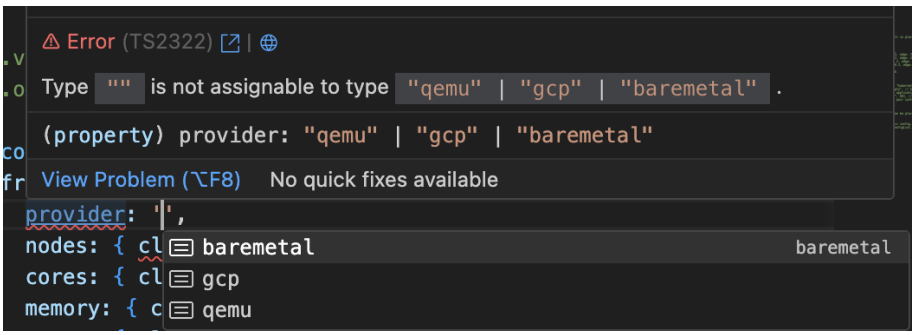

Figure 6.3: EDSL Error Detection and Suggestions

Meaningful Error Messages

When executing Continuum with the standard configuration file, it throws the following error:

```
Invalid value for option infrastructure->provider
```

This error only states that the value is wrong. To find out what values can be used, referring to some documentation is likely to be needed, requiring additional time and effort.

Assuming the errors in **Figure 7.2** are not corrected, the EDSL detects the incorrect values and provides the following error:

```
Type '"invalid1"' is not assignable to type '"qemu" | "gcp" | "baremetal"'.
Type 'string' is not assignable to type 'number'.
Type '"invalid3"' is not assignable to type '"kubernetes" | "kubeedge"
| "mist" | "none" | "kubecontrol"'.
```

It is important to note that Continuum checks the values one by one, throwing errors once it detects an incorrect value. This means that the errors would need to be corrected individually, each requiring execution of the code and analysis of the errors thrown in the terminal. This makes the process of debugging even slower.

6.2.2 Functional Suitability

Functional suitability is a metric used to assess domain experts' capability to achieve their goals. Since both implementations offer all features offered by Continuum, they are equal in this particular assessment.

6.2.3 Maintainability & Extensibility

Configuration files do not perform any checking whatsoever. They assume that the data within them is correct. This makes them extensible to any new additions or changes as long as they contain correct data. On the other hand, the EDSL has a highly structured approach, requiring all possibilities that Continuum offers to be explicitly specified. This includes the variable's name, type, and corresponding encoding to JSON formatting if used. This means using an EDSL might require significant additional development when attempting to make changes or extensions to Continuum and then adapting the EDSL to

those changes. That said, they are still simpler to modify and change than standard DSLs, as they only require adapting existing code structures to suit the new requirements.

Additionally, while VS-Code Snippets offer significant productivity benefits, they are pre-defined and need to be adjusted.

6.2.4 Productivity

Tooling

When making configuration changes, the EDSL assists by providing auto-fill features offered by TypeScript. This makes it easier to make modifications efficiently and correctly. Additionally, hovering over different variables or observing source code can be used to check the different possibilities of all data structures.

On the other hand, standard configuration files do not give any information regarding the expected structure. This means that to use them; domain experts would need to continuously switch between the configuration process and the relevant documentation to understand possible values.

VS-Code Snippets

Using VS-Code Snippets allows for complete, working examples of configurations to be generated by typing in a pre-defined term. This means a complete functional configuration can be created, modified, and used in several seconds.

As standard configuration files do not offer a similar feature, they need to catch up even further when considering the productivity of this particular implementation.

6.2.5 Usability

Usability refers to the ease of use of the language. The reason usability was ranked last is that domain experts are assumed to be highly technical in the context of Continuum. For that reason, the simplicity of the syntax and readability are not as crucial as the other requirements. Additionally, while the syntax can be simpler, it is still relatively readable and concise, making it sensible considering this metric. On the other hand, standard configuration files offer a simple and consistent syntax used to describe key-value pairs.

6.3 Summary

After analyzing how the EDSL and configuration files address the different requirements of the domain, it becomes clear that for the context of complex computer systems such as Continuum, the benefits of increased reliability and productivity out-way the difficulties introduced in usability, maintainability, and extensibility.

These findings are summarized into the following benefits and drawbacks:

6.3.1 Benefits

1. **Improved Reliability** - The EDSL significantly assists in detecting errors and creating reliable programs. It does so by performing static type-checking and providing meaningful errors.

2. **Smoother Learning Curve** - The combination of the EDSL implementation with VS-Code Snippets makes it possible to generate code templates that can be instantly used without requiring and use of documentation. Additionally, for a deeper understanding of the inner working of the EDSL, code structures can be quickly observed in the EDSL's source code.

3. **Increased Productivity** - The EDSL streamlines the configuration process in Continuum by abstracting repetitive patterns and providing a more intuitive syntax. Domain experts can define configurations more efficiently, as the EDSL captures common configuration structures using type aliases and specific data types. By reducing the effort required to create configurations, the EDSL dramatically increases the overall productivity of defining configurations for Continuum.

   Using generated code snippets, the EDSL becomes even more productive, providing the ability to create pre-defined functioning examples of configurations instantly. This can be further extended by creating additional custom snippets.

6.3.2 Drawbacks

1. **Usability** - While the EDSL does not provide a particularly simple syntax, its target audience of technical domain experts in network benchmarking can effectively use the EDSL. The EDSL provides a balance between usability and the specific requirements of the domain. That said, it is less optimal in usability compared to configuration files.

2. **Maintainability & Extensibility** - Configuration files can be extended with ease as they require no additional development. However, unlike configuration files EDSLs require additional development to accommodate any change in the domain they interact with.

Chapter 7

Conclusion

This thesis has explored the concept of Embedded Domain Specific Languages (EDSLs) and their potential impact on software development. The research shows that EDSLs offer extensive features not typically found in traditional Domain Specific Languages (DSLs).

One notable advantage of EDSLs is their cost-effectiveness in terms of development. The embedded nature of EDSLs allows them to leverage existing infrastructure and tools, reducing the need for extensive development efforts. This aspect makes EDSLs an attractive option for organizations seeking to develop specialized software solutions while maintaining cost efficiency.

7.1 Answering Research Questions

RQ1 What are important considerations and design guidelines for creating an embedded Domain-Specific Language (EDSL) for complex computer systems? - The thesis highlights several considerations that need to be made regarding the domain of interest and the application of an EDSL for it. This can assist researchers and engineers in understanding whether a DSL is appropriate for the particular domain they are interested in improving. Additionally, guidelines were proposed regarding the GPL selection process and the design that can be taken into account to streamline the development of an EDSL.

RQ2 What are benefits and drawbacks associated with designing and implementing an EDSL in the context of Software Configuration Management(SCM) compared to traditional approaches? - The evaluation of this research performs an in-depth analysis comparing the EDSL implementation that was made for the

SCM process of Continuum to the traditional approach it used prior (i.e., configuration files). A description of the benefits and drawbacks was provided.

RQ3 In what ways can the utilization of Visual Studio Code snippets enhance the interaction between domain experts and an EDSL? - Thanks to the incorporating of snippets into the Continuum EDSL, configuration templates can now be generated and used with ease. This reduces the need for manual typing, minimizes errors, and enhances overall development efficiency. Furthermore, the evaluation of this thesis highlights that properly implemented snippets can also serve as educational tools, guiding domain experts in understanding the syntax, available options, and expected structure of configurations. This promotes a smoother learning curve and can potentially increase the adoption of the EDSL by domain experts.

7.2 Limitations and Future Work

EDSLs may lack the same level of flexibility as traditional DSLs. Therefore, when considering the use of EDSLs, it is crucial to carefully evaluate the domain of interest and the trade-offs between the desired features against the required flexibility of the language.

Another limitation is the need for more extensive research available on EDSLs. Although this thesis aims to aid future researchers, it is important to acknowledge that the knowledge and resources for EDSLs are relatively limited.

The design guidelines proposed in this thesis have been developed to assist future researchers and engineers in implementing and utilizing EDSLs. These guidelines aim to provide valuable insights into best practices and considerations when working with EDSLs, contributing to advancing research on this particular topic.

While an EDSL implementation was applied to an existing use case, further research could include quantitative performance studies to further assess the practical implementation of EDSLs in real-world scenarios.

The findings presented in this thesis highlight the potential of EDSLs as a cost-effective solution for developing specialized languages for developing programs in particular domains. The design guidelines proposed aim to serve as a resource for researchers and engineers, facilitating the implementation and exploration of EDSLs. While further research is needed to understand the capabilities and limitations of EDSLs fully, this thesis aims to lay a basis for future development and adoption of EDSLs in the industry.

References

[1] Ivo Damyanov and Mila Sukalinska. **Domain Specific Languages in Practice**. *International Journal of Computer Applications*, **115**:42–45, 04 2015. 1

[2] Arie van Deursen, Paul Klint, and Joost Visser. **Domain-Specific Languages: An Annotated Bibliography**. *SIGPLAN Not.*, **35**(6):26–36, jun 2000. 1

[3] Marjan Mernik, Jan Heering, and Anthony M. Sloane. **When and How to Develop Domain-Specific Languages**. *ACM Comput. Surv.*, **37**(4):316–344, dec 2005. 1

[4] Steve Freeman and Nat Pryce. **Evolving an Embedded Domain-Specific Language in Java**. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '06, page 855–865, New York, NY, USA, 2006. Association for Computing Machinery. 1, 2

[5] Arie van Deursen, Paul Klint, and Joost Visser. **Domain-Specific Languages: An Annotated Bibliography**. *SIGPLAN Not.*, **35**(6):26–36, jun 2000. 2

[6] Jacky Estublier, David Leblang, André van der Hoek, Reidar Conradi, Geoffrey Clemm, Walter Tichy, and Darcy Wiborg-Weber. **Impact of Software Engineering Research on the Practice of Software Configuration Management**. *ACM Trans. Softw. Eng. Methodol.*, **14**(4):383–430, oct 2005. 2, 7

[7] Matthijs Jansen, Linus Wagner, Animesh Trivedi, and Alexandru Iosup. **Continuum: Automate Infrastructure Deployment and Benchmarking in the Compute Continuum**. In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering*, ICPE '23 Companion, page 181–188, New York, NY, USA, 2023. Association for Computing Machinery. 4, 8, 21

[8] Marjan Hericko, Matjaz B. Juric, Ivan Rozman, Simon Beloglavec, and Ales Zivkovic. **Object Serialization Analysis and Comparison in Java and .NET**. *SIGPLAN Not.*, **38**(8):44–54, aug 2003. 10, 17

[9] Gökhan Kahraman and Semih Bilgen. **A framework for qualitative assessment of domain-specific languages**. *Software & Systems Modeling*, **14**:1505–1526, 2015. 15

[10] Juha Kärnä, Juha-Pekka Tolvanen, and Steven Kelly. **Evaluating the use of domain-specific modeling in practice**. *The 9th OOPSLA Workshop on Domain-Specific Modeling*, 10 2009. 15

[11] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. **Design Guidelines for Domain Specific Languages**, 2014. 19

[12] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. **An Empirical Study on Configuration Errors in Commercial and Open Source Systems**. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 159–172, New York, NY, USA, 2011. Association for Computing Machinery. 23

Appendix A

Reproducibility

A.1 Abstract

This section describes the reproduction of the EDSL development using TypeScript within the Continuum framework. The following outlines the necessary steps, dependencies, and considerations for replicating the EDSL implementation.

A.2 Artifact check-list (meta-information)

- **Compilation:** Using the ts-node package version 10.9.1. installed using Node.js,
- **Run-time environment:** Ubuntu(OS), Node.js
- **Compilation and Execution:** Executed by ts-node
- **Output:** Console, JSON Object
- **Disk space required:** 4MB
- **How much time is needed to prepare work environment:** 30 minutes
- **Publicly available?:** Yes
- **Code licenses:** MIT License

A.3 Description

The EDSL implementation made for this paper involved implementing a domain-specific language within the Continuum framework, enabling efficient software configuration and management (SCM) using TypeScript. This section provides details on how to access, install, and use the EDSL, along with its software dependencies.

A.3.1 How to access

The code is publicly available as part of the Continuum framework at `https://github.com/atlarge-research/continuum`.

A.3.2 Software dependencies

Since TypeScript builds on top of JavaScript, it can make use of the **Node.js package manager (NPM)**, which assists in both execution and the addition of external libraries.

Once Node.js is installed there are 2 additional dependencies that need to be installed (using NPM):

1. **ts-node:** Compiles and executes Typescript code.
2. **process:** Enables process termination.

A.4 Installation

First, NPM needs to be installed. This can be achieved running the following command in an Ubuntu terminal:

```
sudo apt install nodejs
```

Once NPM is installed we can download the other dependencies by running:

```
npm install
```

A.5 Experiment workflow

To perform experiments on this framework, a "Configuration" instance needs to be created. An example of such a configuration is:

```
import Configuration from "./configuration"
import { applicationVars } from "./generics"

const configuration = new Configuration({
    infrastructure: {
        provider: 'qemu',
        nodes: { cloud: 0, edge: 1, endpoint: 1 },
        cores: { cloud: 1, edge: 1, endpoint: 11 },
        memory: { cloud: 2, edge: 1, endpoint: 1 },
```

```
        quota: { cloud: 0.4, edge: 0.3, endpoint: 0.5 },
    },
    benchmark: {
        resourceManager: "kubernetes",
        application: "empty",
        applicationVars: applicationVars([
            ["sleep_time", 60],
        ])
    }
})
configuration.validate()
configuration.output()
```

Once a configuration is complete, Continuum can be executed with the configuration file by running the following terminal instruction, where "<config>.ts" should be replaced with the actual TypeScript configuration file.

```
python 3 continuum.py DSL/<config>.ts
```

It is important to note that the DSL requires the Node.js dependencies defined in the DSL directory and therefore must be declared within that folder to operate correctly.

A.6 Evaluation and expected results

Evaluation of the EDSL implementation was done with qualitative measures described in the literature. It was done by comparing the EDSL to the previous means of defining configuration in Continuum (i.e., standard configuration files). The qualitative measures were reliability, functional suitability, maintainability, extensibility, productivity, and usability.

A.7 Experiment customization

In order to customise the configuration files, configuration templates can be generated in Visual Studio Code by typing pre-defined keywords (i.e., cfg basic).

A.8 Notes

The primary purpose of creating the EDSL was to provide structure, validation, and ease for defining configurations in Continuum. While this approach has proven to be effec-

tive, generating the configurations using VS-Code Snippets has played a significant role in improving all three of these facets and is expected to be used when defining future configurations.

A.9 Methodology

We evaluate the reproducibility of this work by observing its repeatability. In particular, we observe the repeatability of the condition of measurement. This is achieved by replicating the previously created configuration files defined in Continuum and observing whether they produce the same output.