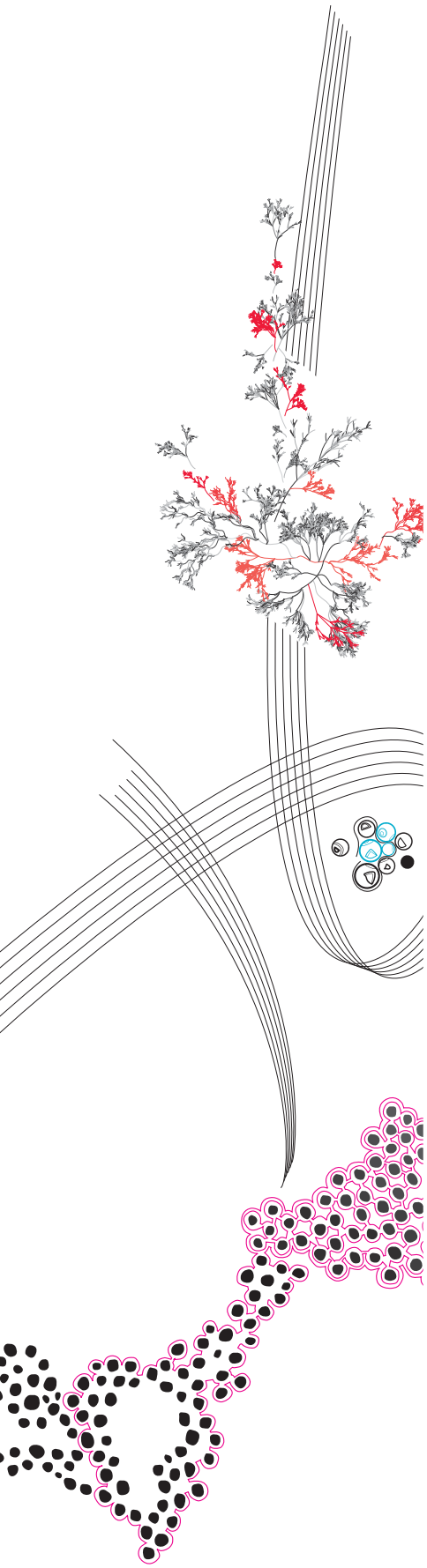

MSc Business Information Technology
Thesis

Designing a Methodology for the Development of Domain Specific Languages with both Graphical and Textual Elements

Simon van Roozendaal

Luis Ferreira Pires
Renata Guizzardi-Silva Souza
João Rebelo Moreira

September, 2024

Master Business Information Technology
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Contents

Abstract

In the rapidly evolving field of software development, the need for highly skilled developers is growing, yet the industry faces a significant shortage of such talent. This gap presents a pressing challenge for organizations striving to develop complex software systems quickly and efficiently. A promising solution lies in enabling non-technical domain experts to contribute directly to the development process through Domain-Specific Languages (DSLs). However, while textual DSLs offer tailored solutions for specific domains, they can be inaccessible to non-technical users due to their complexity. This thesis addresses this issue by proposing a comprehensive methodology for transforming a Textual Domain-Specific Language (TDSL) into a Hybrid Domain-Specific Language (HDSL) that integrates graphical and textual elements, making the system more intuitive and accessible.

The proposed methodology guides the entire development process, from the decision phase, which assesses the feasibility and scope, to the design and implementation of an HDSL that maintains the functional integrity of the original TDSL while enhancing usability through graphical components. The methodology is validated through a case study using the Ampersand platform, a tool for generating information systems based on formal specifications. A prototype of an HDSL for Ampersand was developed and tested through expert interviews and usability assessments. The results demonstrate that while the graphical interface improved user accessibility and efficiency, challenges remain in operability and satisfaction compared to traditional textual interfaces.

This research provides a structured approach to developing HDSLs, contributing to a broader inclusion of non-technical domain experts in software development processes, and offers insights into refining HDSLs for future application through iterative testing and validation.

Keywords: Hybrid Domain-Specific Language (HDSL), Textual Domain-Specific Language (TDSL), Graphical Syntax, Software Development Methodology, Domain-Specific Language (DSL), Ampersand Platform, Information System, Formal Specifications, Model-Driven Engineering

Chapter 1

Introduction

The objective of this chapter is to provide an overview of the research topic, its significance, and the goals of this thesis. It discusses the motivation behind the study, the primary research questions, and the methodology used to address these questions. The chapter sets the stage for the detailed exploration and analysis presented in the subsequent chapters.

1.1 Motivation

In today's software development landscape, rapidly creating well-functioning systems is a priority, while technically skilled developers who can support this fast development are scarce. Organizations are trying to quickly develop their software and information systems, which leads to an increase in demand for the professionals able to generate this software [12]. A survey from 2020 [7] found that 85 percent of recruiters and hiring managers observed a shortage of software developers on the job, and data trends even indicate growth in the total shortage of developers worldwide. The shortage of qualified personnel to develop systems leads to pressure on the existing developers, further increased by the difficulty of managing and using the rapidly evolving systems.

A method of reducing this shortage is engaging professionals from non-technical sectors in the development process. This is strategically complementary to the traditional development practices, but seems to have a positive effect when applied to appropriate use cases in organizations [12]. It helps facilitate direct business knowledge integration, as people can turn ideas directly into solutions [32]. However, employing professionals from non-technical sectors is not readily available; the development process must become more suitable to accommodate this inclusion.

One significant barrier to this suitability is the complexity of software development, especially when using various text-based syntaxes. These languages often require a detailed understanding of syntax rules and structures, which can be complex and error-prone even for experienced developers. Writing code in a textual format requires adherence to specific syntax conventions, making it challenging for beginners or non-experts to grasp and use effectively [9]. Simplified or more manageable explanations of what specific textual syntaxes entail are essential for understanding and applying them. This can be achieved by combining them with other methodologies to achieve a higher level of abstraction. An example of this is the use of Domain Specific Languages (DSLs), which provide abstraction by being targeted towards specific domains, known as abstraction through language [9].

Abstracting through language alone is not sufficient. While DSLs are more accessible to domain specialists by leveraging their expertise, the textual barrier remains a significant obstacle. Other methods to increase accessibility involve converting a DSL with purely textual syntax

(textual DSL or TDSL) into a DSL with purely graphical syntax (graphical DSL or GDSL). Although developers have historically preferred textual languages to visual ones, graphical alternatives offer advantages in terms of visual communication [1]. By combining both graphical and textual elements, developers can leverage the strengths of each approach to create more expressive and effective code representations. Rather than aiming to replace textual syntax, the goal is to supplement it with interactive and visual programming constructs tailored to specific problem domains [1, 10].

1.2 Problem Statement

The complexity of TDSL often acts as a barrier in software development, limiting their accessibility, particularly for domain experts who lack extensive programming knowledge. While TDSLs offer valuable domain-specific abstractions, they still require users to understand and navigate complex syntax rules. By utilising visual elements to represent programming constructs, GDSL offer a potential solution by lowering the entry barrier and enhancing intuitive understanding through visual communication. However, transitioning from a purely TDSL to a Hybrid graphical-textual domain specific language (HDSL) presents significant challenges. These include maintaining the expressiveness and functional integrity of the original DSL while making it more accessible and user-friendly. This thesis delves into the transformation process, focusing on the design, implementation, and validation of enhancing a TDSL with graphical features to make it functional and accessible to a broader audience.

1.3 Research Goals

The primary objective of this research is to explore how a HDSL can be developed from a TDSL to improve its accessibility and usability. To achieve this goal, the research focusses on the following specific objectives:

- *Develop a Comprehensive Methodology:* formulate a systematic approach for developing a HDSL from a TDSLs. This methodology addresses the preservation of functional integrity while enhancing usability and accessibility.
- *Implement a Prototype:* apply the developed methodology to create a HDSL from an existing TDSL. This prototype has been assessed by using the validation method.

1.4 Research Methodology

This thesis adopts the Design Science Research Methodology (DSRM) to guide the transformation of a TDSL into a GDSL. The research followed the six stages of DSRM as depicted in figure 1.1, based on the research of [27].

1. *Problem Identification and Motivation:* this stage has been reported in the previous sections, highlighting the challenges faced by domain experts due to the complex syntax of TDSLs and the potential of HDSLs to make the development process more accessible.
2. *Define objectives of a Solution:* the objective has been set; to develop a comprehensive methodology that can systematically guide the development of HDSLs from TDSLs. The aim of the methodology is to lower the entry barrier to programming for non-technical users by enhancing intuitive understanding through visual representations, and ensure that the functional integrity and expressiveness of the original TDSL are preserved in the resulting HDSL.

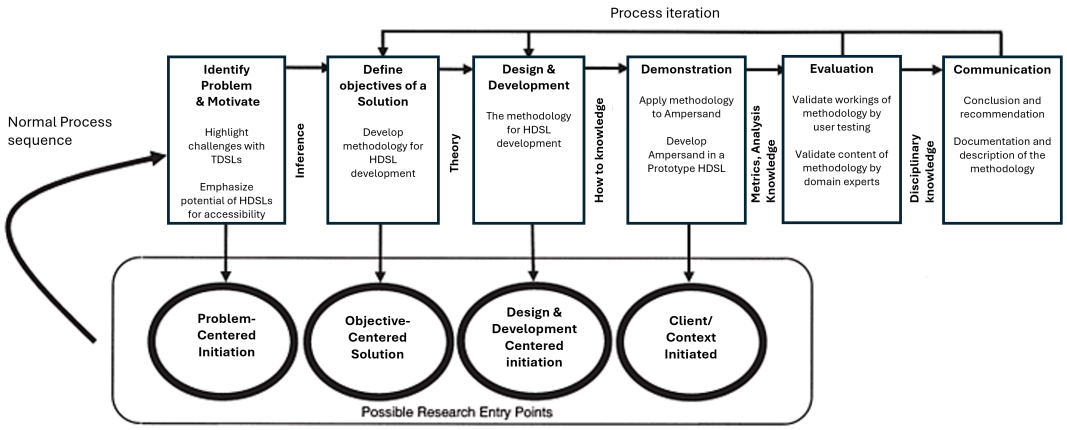

FIGURE 1.1: Overview of the implemented DSRM based on the work of [27]

3. *Design and Development:* a methodology for developing a HDSL from a TDSL has been developed. This methodology acts as a blueprint outlining the necessary steps for the development .

4. *Demonstration:* the developed methodology has been applied to a concrete environment (prototype) to showcase its usability and effectiveness in guiding the development process. This prototype served to demonstrate the feasibility and effectiveness of the development process outlined in the methodology.

5. *Evaluation:* the methodology has been validated using a validation approach, including usability testing with potential users of the demonstrated prototype and assessment of the methodology itself by domain experts by interviews.

6. *Communication:* the methodology and demonstration of this research is documented comprehensively. Additionally, the conclusion and recommendation of the research, including summarizing key findings, contributions, and implications for future research are documented.

1.5 Outline

This thesis is structured following the DSRM as follows: Chapter 2 provides a foundational understanding of the research environment, highlighting the differences between textual and graphical formats and reviewing existing transformation methods. Chapter 3.1 details the development of a methodology to transform a TDSL into a HDSL. Chapter 4 demonstrates the application of this methodology through a case study with the Ampersand platform. Chapter 5 evaluates the methodology by assessing the HDSL prototype using expert interviews and usability testing. Finally, Chapter 6 concludes with a synthesis of the research findings, offering recommendations for future research and discussing the broader implications of the study.

Chapter 2

Background

The objective of this chapter is to provide a foundational understanding of DSLs, emphasizing the differences between textual and graphical formats and reviewing a prominent existing methodology for their development. A review of work with similar research question as our research is also included, showcasing various approaches and techniques used in previous research to development of (H)DSLs. The result is a comprehensive background that sets the stage for the subsequent development of the methodology for the HDSL. It provides the necessary theoretical framework and context, enabling a better understanding of the challenges and considerations involved in developing HDSL from TDSLs

2.1 Domain Specific Languages

DSLs are specialized programming languages designed to address the particular requirements of a specific application domain. Unlike general-purpose programming languages (GPLs) such as Java or Python, DSLs are specifically designed to be used in particular domains or fields, thereby facilitating more effective communication and operational efficiency within those domains. In contrast to the broad and detailed syntax of GPLs that are designed to handle a wide variety of programming tasks, DSLs simplify interactions by focusing only on those elements that are relevant to the specific domain [18].

Even though the term 'Language' suggests that DSLs consist of written text, they actually exist in various formats like text-, graphical-, or sound-based. TDSLs have textual syntax but specifically include syntax and semantics optimized for a particular domain, while GDSLs use visual representations to specify software solutions or specific problems within a domain. More types of DSLs exist; for example, Audio DSLs are specifically designed for working with audio and sound. In this research, however, we will focus on GDSLs and TDSLs.

The use of DSLs is a method of abstraction through language definition [9] since by providing a higher level of abstraction and domain specificity it simplifies the programming task. This is particularly beneficial in environments where domain experts may not necessarily be trained programmers, but are participants in the software development process. The design of DSLs allows these professionals to engage with the development process more intuitively, using languages that encapsulate the complexity of programming into a more accessible form.

2.1.1 Structure of the DSL

According to [26], a DSL has one Abstract Syntax, (possibly) multiple Concrete Syntaxes and Semantics.

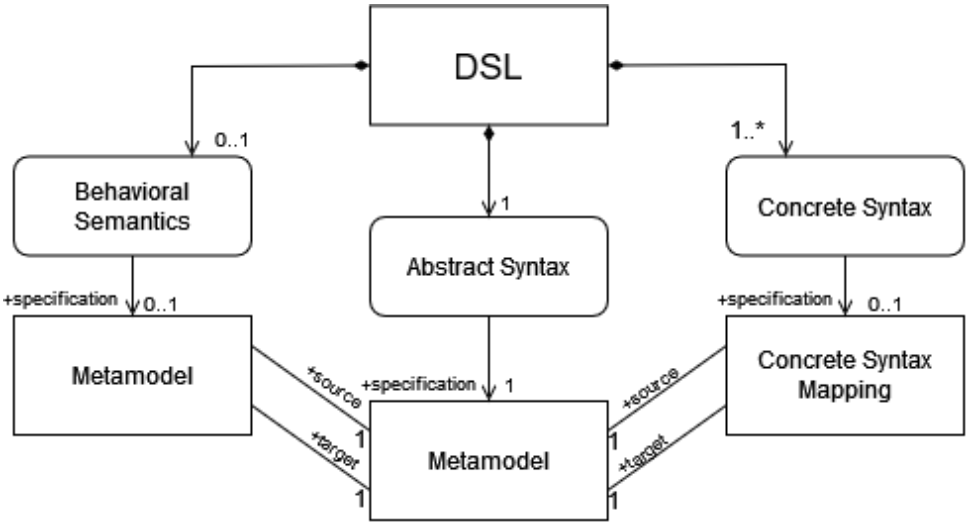

FIGURE 2.1: An overview of the elements of a DSL

The Abstract Syntax represents the structural part of a language, independent of any specific representation or notation. It defines the core concepts of a language and the relationships between them, but without detailing how these elements are written or displayed. *For instance, in a DSL for a banking application, the abstract syntax would define entities like Account, Transaction, and Customer without specifying how these elements are shown in the user interface or stored in a database.*

It is crucial to define the core concepts and constructs of the DSL at an abstract level, helping to ensure that the DSL captures the essential elements of the domain without being tied to a specific representation. Abstract syntax serves as the foundation for both the development and understanding of the DSL.

A Metamodel is essentially a description of the abstract syntax of a DSL. It specifies what elements are possible in the language and how they can be connected or related. It is a model of the models that can be built using the DSL. The concrete syntax must conform to the abstract syntax as defined by the metamodel. This means whatever is expressible in the concrete syntax should be mapped to the structures and constraints set by the metamodel. For instance, if the metamodel defines an entity "Account" with attributes "balance" and "accountNumber," the concrete syntax must provide a way for users to specify these attributes, whether through textual commands or graphical inputs. Establishing a metamodel is essential, as it provides a structured framework that defines the relationships and constraints of the language constructs. This metamodel acts as the blueprint for the DSL, ensuring consistency and coherence in its design.

The Semantics involves defining the meaning of the constructs of the language—how they behave during execution or simulation, as well as the rules that must be followed at all times. While DSLs always have semantics to ensure the correct interpretation and behaviour of their constructs, it is not always explicitly defined. Specifically, this pertains to two main types of semantics: static and behaviour semantics.

Static Semantics define the rules that must be followed at all times, regardless of execution. These rules ensure the integrity and validity of the DSL constructs before execution. For example, static semantics in a banking DSL might enforce that each account must have a unique account number and that certain fields cannot be left empty.

Behavioural Semantics describe how the system's state changes in response to the execution

of DSL constructs. These can be formal or informal:

- *Formal Semantics:* involves using rigorous methods such as attribute grammars, rewrite systems, and abstract state machines to define the operational, denotational, and axiomatic aspects of the DSL. *For example, in a banking DSL, formal semantics might specify that an account balance must always be non-negative.*
- *Informal Semantics:* utilizes natural language descriptions supplemented by illustrative examples. *For example, in a banking DSL, informal semantics might describe that if a withdrawal exceeds the account balance, the transaction is denied.*

A Concrete Syntax refers to the actual notation or representation of the language. It details how the elements defined in the abstract syntax are expressed, either textually or graphically. A DSL can have multiple concrete syntaxes, which should all adhere to the same abstract syntax. For example, whether a Transaction is represented as a table of values in a UI or as XML elements in a configuration file.

2.1.2 Concrete Syntaxes

Textual Languages

Textual languages in the context of software development are languages with text-based syntaxes, which are used to represent models of software systems. These models represent the abstract structure and behaviour of a system. This method of development is seen as the traditional way of representing programming constructs using text-based symbols and structure [1]. Often referred to as 'programming' or 'writing code', widely used programming languages like Python or Java are practical implementations of textual syntax-based development. Text is used to instruct the system to perform tasks, following specific syntactic rules unique to each language. The core components of textual syntax are its syntax and semantics; the syntax describes the structure of valid sentences, and the semantics assign meaning to these sentences [10].

Graphical Languages

Graphical languages leverage the manipulation of graphical elements like icons, blocks, diagrams, and forms to represent program logic and structure in order to construct software applications [15]. According to Burnett [4], this allows developers to use the spatial relationships between elements by programming in a multidimensional space, since the y and x-axis of the screen and even layers of elements are now available. The usage of space and graphical elements are used to hide the complex implementation details from users and allows them to focus on the logic and structure of their programs without getting bogged down in technical intricacies [15].

Hybrid graphical-textual languages

While traditionally DSL are either text-based or graphical, they are increasingly being developed as hybrid graphical-textual languages to harness the strengths of both forms. These hybrid languages combine the detailed expressiveness of textual syntax with the intuitive clarity of graphical syntax, creating a versatile tool for domain experts and developers alike. It typically incorporates an Abstract Syntax that defines the core concepts and relationships independent of their visual or textual representation. This abstract syntax underlies both the graphical and textual concrete syntaxes, ensuring that they are semantically aligned and interchangeable:

- *Textual Concrete Syntax*: involves the traditional text-based representation of models, where programming constructs are specified using textual symbols within a structured syntax. This approach is valued for its precision and ability to handle complex expressions and detailed specifications.

- *Graphical Concrete Syntax*: utilizes visual elements such as icons, blocks, and diagrams to represent the components of the DSL. This method emphasizes ease of understanding and interaction, particularly useful for visualizing relationships and hierarchies within the domain.

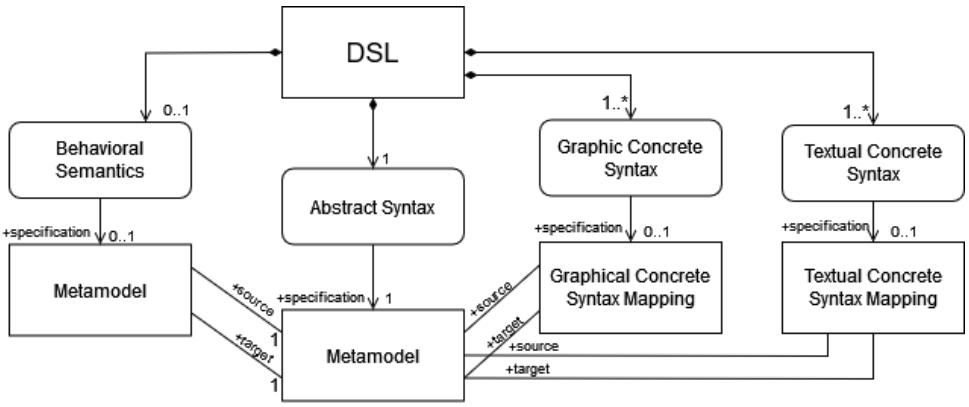

FIGURE 2.2: An overview of the elements of a HDSL

The hybrid approach involves a seamless integration between these two syntaxes, often facilitated by development environments that support live synchronization between the graphical and textual views. Changes made in the graphical interface are instantly reflected in the textual code and vice versa, enhancing the coherence and usability of the Hybrid languages.

2.2 DSL Development Process

Developing a DSL involves a process that integrates both theoretical underpinnings and practical considerations to address specific needs within a domain. Based on the comprehensive work by [18], the development of a DSL can be broken down into several distinct phases, each requiring careful attention to detail and a deep understanding of both the domain and the technology.

Decision phase

The decision to develop a DSL should be driven by a clear need to address specific problems within a domain that cannot be effectively solved by GPLs. This phase involves identifying potential gains in productivity and expressiveness that a DSL can offer over existing solutions. [18] emphasize the importance of evaluating the scope and feasibility of a DSL in the initial stages, including the consideration of the potential user base and the specific tasks that the DSL will simplify.

Analysis phase

Critical to the success of a DSL is a thorough domain analysis, which involves gathering and formalizing knowledge about the domain. Most DSLs investigated by [18] use an informal domain analysis method for this, but this also can be done formally by following a methodology. Informal domain analysis typically involves gathering and formalizing knowledge about the domain without strictly following any predefined methodology. This approach relies heavily on the expertise and intuition of domain experts.

- *Knowledge Gathering*: collecting explicit or implicit domain knowledge from various sources such as technical documents, domain experts, existing code, and customer surveys.

- *Terminology and Semantics*: developing domain-specific terminology and semantics, which can vary in their level of abstraction, to create a shared understanding among stakeholders.

Formal domain analysis involves using structured methodologies to systematically capture and formalize domain knowledge. Various formal methodologies such as FODA (Feature-Oriented Domain Analysis) [11], DARE (Domain Analysis and Reuse Environment) [8], and DSSA (Domain-Specific Software Architectures) [34] are employed. These methodologies provide frameworks and tools for systematically capturing domain knowledge and guiding the reuse of implemented components. This approach aims to produce a detailed domain model that includes:

- *Domain Definition*: defining the scope of the domain and its boundaries.
- *Terminology*: developing a comprehensive vocabulary or ontology that captures the essential concepts and their relationships within the domain.
- *Domain Concepts*: detailed descriptions of domain concepts, supported by feature models that describe the commonalities and variabilities of these concepts and their interdependencies.

Design phase

The design of a DSL involves the careful planning of its syntax and semantics. The syntax should be intuitive and aligned with the domain's terminology, while the semantics must accurately reflect the domain operations. [18] discuss different approaches to DSL design, including the use of existing language constructs (or language exploitation). In most cases, one would implement a language by defining a model covering all the necessary concepts together with their attributes. This language model builds the foundation of the language to model the desired aspects [36].

Identify Domain Concepts:

- Analyse the gathered knowledge on the domain to identify key concepts and operations that the DSL needs to support.

Define Syntax and Semantics:

- Design the syntax to be user-friendly and aligned with the domain's terminology, ensuring that it is intuitive for the end-users. This involves selecting appropriate keywords, symbols, and notations.
- Design the semantics to accurately reflect the domain operations, ensuring that the DSL's behaviour is consistent with domain-specific requirements.

Prototype and Iterate:

- Develop prototypes of the DSL to be used to gather feedback from domain experts. Based on this feedback, the design is iterated to refine syntax, semantics, and usability.

Implementation phase

The implementation of a DSL is a crucial phase where the theoretical designs are translated into practical software tools. As described by [18], this process involves coding, integration, and preliminary testing of the DSL components according to the predefined specifications. The implementation strategies are chosen based on the DSL's characteristics, which dictate the most suitable approach among interpreters, compilers, and embedding within existing environments.

- *Interpretation*: consists of creating an interpreter that executes the DSL commands directly. It allows for quick changes and testing, but might suffer from lower performance compared to compiled languages.
- *Compilation*: consists of compiling or translating the DSL into a lower-level language, by which performance and integration with other systems can be enhanced. This requires a deep understanding of both the target language and the runtime environment.
- *Embedding*: consists of incorporating the DSL into an existing programming environment that can leverage existing tools and frameworks, thus reducing development time and learning curves.

The implementation often includes preprocessing, where DSL code is converted into a format that existing compilers or interpreters can process. Macros can be used to extend the capabilities of the host language, allowing simpler implementation of complex DSL features. Additionally, the integration with current development tools is crucial to ensure that the DSL can be developed, maintained, and used efficiently within the existing software infrastructure. This integration involves enhancing or creating tools such as syntax-highlighting editors, debuggers, and automated consistency checkers, which strengthens both the usability and maintainability of the DSL. Performance tuning is also a critical component of this phase. It may involve domain-specific optimizations like partial evaluation, which precomputes invariant parts of the DSL code, or optimizations that restructure the DSL code to improve performance.

Deployment phase

Although [18] consider the deployment phase outside the scope of their article, the phase has practical importance. Deployment involves rolling out the DSL to its intended users and ensuring its integration within existing systems and workflows. This phase includes comprehensive documentation, training sessions, and support mechanisms to facilitate adoption. Moreover, continuous monitoring and maintenance are required to address any emerging issues and to update the DSL as the domain and needs of the users changes.

2.3 Related work

To prepare for this project, relevant literature was investigated into that addresses questions similar topics to the research questions. The literature was identified through multiple searches using Scopus and Google Scholar, extended with snowballing by an iterative process of selecting a growing set of papers based on the references of the (already included) set of papers. This process was iterated until the literature collection was deemed sufficiently, and yielded a set of literature sources covering various techniques, summarized in Table 2.1. The insights gathered from the reviewed articles have been used in developing and refining the new methodology for transforming a TDSL into an HDSL.

Table 2.1: Literature sources with similar topics

Article	Title
Predoaia et al. 2023 [29]	Towards Systematic Engineering of Hybrid Graphical Textual Domain Specific Languages
Predoaia et al. 2023 [30]	Streamlining the Development of Hybrid Graphical-Textual Model Editors for DSL
Denkers & Vollebregt 2018 [**?**]	Migrating Custom DSL Implementations to a Language Workbench (Tool Demo)
Cooper & Kolovos 2019 [5]	Engineering Hybrid Graphical-Textual Languages with Sirius and Xtext
Pérez Andrés et al. 2008 [28]	Domain Specific Languages with Graphical and Textual Views

Krieger 2024 [14]	HyLiMo: A Textual DSL and Hybrid Editor for Efficient Modular Diagramming
Scheidgen 2008 [33]	Textual Modelling Embedded into Graphical Modelling
Maro 2015 [17]	A DSL Supporting Textual and Graphical Views
Toussaint & Baar 2018 [36]	Enriching Textual Xtext-DSLs with a Graphical GEF-Based Editor

[30] present a framework to systematize the engineering of HDSLs. The authors identify key challenges in maintaining the functional integrity of DSLs while introducing graphical components. Their methodology employs declarative specifications to designate parts of the DSL as graphical or textual, facilitating clear delineation and integration of different syntax types. They propose using the Sirius Viewpoint Specification Model (VSM) for the graphical syntax and formal grammars for the textual syntax. This combination ensures that each part of the DSL is optimized for its specific role, thus maintaining consistency across different representations and enhancing error reporting mechanisms. This approach addresses the complexity of managing HDSLs by reducing the reliance on handwritten code, automating the generation of 'glue code' required for integrating textual and graphical components, and establishing methods for uniform error reporting and model consistency enforcement as the DSL evolves.

In their subsequent work, [29] extend the research to focus specifically on the model editors used for managing HDSLs. This paper introduces 'Graphite,' a tool designed to streamline the development of hybrid graphical-textual model editors for DSLs while minimizing manual coding. This approach integrates the Sirius graphical modelling framework and the Xtext textual modelling framework, focusing on DSLs that predominantly use graphical elements but are enhanced with textual syntax for expressing complex behaviours. They tackle the challenges of maintaining functional integrity and consistency across the graphical and textual components through model-driven engineering techniques, aiming to automate the integration and minimize errors. The Graphite tool is pivotal in facilitating the seamless combination of graphical and textual elements, ensuring that the DSLs remain robust and user-friendly.

[6] propose a methodology for migrating custom DSL implementations to a language workbench, specifically using the Spoofax platform. Their methodology emphasizes maintaining the integrity of existing DSLs while integrating graphical enhancements. The process involves transforming existing DSL implementations, which were originally based on XML and processed through Python scripts, into a more robust and feature-rich environment provided by Spoofax. This transformation allows for the preservation of existing functionality and backward compatibility while introducing improved syntax and integrated development environment (IDE) support. The approach is demonstrated through the migration of two specific types of DSLs: Interface Definition Language (IDL) and Océ Interaction Language (OIL), focusing on modular language definition to facilitate the transition and integration between graphical and textual components.

[5] discuss the integration of the Sirius and Xtext frameworks to engineer hybrid graphical-textual languages within the Eclipse Modelling environment. They focus on the requirements for embedding TDSLs into graphical modelling workbenches to harness the strengths of both textual and graphical representations. This includes using Sirius for defining graphical notations and Xtext for embedding textual syntax, aiming to maintain the integrity and enhance the functionality of DSLs. Key activities involve addressing challenges such as refactoring, code generation, and synchronization between graphical and textual components, highlighting the need for comprehensive tool support to manage the complexity introduced by integrating these two frameworks. The conclusion is that a modelling workbench supporting hybrid graphical-textual concrete syntaxes is needed to close the gap between models and code.

[28] present a methodology for defining DSLs that incorporate both graphical and textual views. Their approach leverages the metamodel ling tool AToM3 and employs triple graph grammars (TGG) for transforming and synchronizing the metamodel elements into a DSL that supports both views. The methodology begins by defining the metamodel of the language, from which subsets corresponding to different diagram types or viewpoints are derived. Each viewpoint is then equipped with graphical or textual concrete syntax as needed, using triple graph transformation systems to ensure consistency and integration of these viewpoints. This method aims to maintain the integrity of the original DSL while enriching it with graphical and textual capabilities for enhanced usability and expressiveness.

The research by [14] introduces HyLiMo, a framework that supports a modular approach for hybrid diagramming combining graphical and textual elements. The methodology focuses on integrating a DSL for textual inputs and a live-syncing graphical editor, facilitating both the manipulation of diagram layout and the programming of complex layout behaviours. The DSL allows users to define and manipulate elements graphically and textually, ensuring consistency and interactivity between both views. This approach utilizes web technologies to create a versatile environment for UML class diagrams, aiming to enhance usability and customization through programming constructs, while maintaining the visual clarity and functional integrity of the diagrams.

[17] explores methodologies for developing a DSL that supports both textual and graphical views, using a combination of EMF, Xtext, and GMF tools. The methodology focuses on transforming an existing GDSL into a textual format without increasing maintenance efforts. This process includes the creation of a prototype that operates in both modalities and examines the ease and efficiency of transitioning between these views. Key elements of the methodology involve the use of model transformations to ensure consistency between the graphical and textual representations, and the application of Xtext for constructing the TDSL environment. The approach aims to maintain the functional integrity and synchronization of both DSL forms through efficient model transformations and integration strategies.

In [36], the authors outline a methodology to enrich TDSLs created with Xtext by integrating a graphical editor using the GEF (Graphical Editing Framework). They focus on maintaining the functional integrity of the original DSL while allowing for graphical manipulation. The process involves using the existing Xtext grammar of a DSL, supplemented with validators and code generators, and augmenting this with a GEF-based graphical editor. This integration allows for a hybrid representation, where elements of the DSL can be manipulated both textually and graphically. The approach ensures that changes in the graphical interface reflect accurately in the textual model through synchronization, preserving the underlying DSL's consistency and functionality.

2.4 Application of Literature Review

Overall, the reviewed studies exhibit a similar structure to the methodology proposed by [18]. Even though they do not explicitly divide the process into the distinct phases outlined by Mernik, the reviewed studies use comparable steps and approaches. The information and activities from these studies have been categorized into the various phases of DSL development, which have subsequently informed the activities in the newly developed methodology.

An explanation of the activities of the papers, separated into the five phases, can be found in Appendix A in Table A.2. In addition to this, an overview of the software used in the studies

is provided in Appendix A in Table A.1. This section offers insights into the technological tools and frameworks employed in the research, which are critical for understanding the practical aspects of DSL development.

Finally, the individual requirements for the final HDSL, derived from the various studies, are also extracted and presented in Appendix A. These requirements serve as a foundation for the development of the HDSL, ensuring that the final product meets the standards and addresses the needs identified in the existing research.

Chapter 3

Development Methodology

This chapter presents the proposed methodology for deploying HDSLs from TDSLs, integrating both textual and graphical elements. Building upon the foundational research of [18] outlined in Section 2.2, this chapter modifies and extends the established methodology to address the unique challenges and requirements of HDSL development.

The methodology presented in this chapter was designed and developed following an in-depth examination of [18]'s original framework, as outlined in Section 2.2, and by reviewing relevant case studies that aimed to integrate textual and graphical DSLs. This chapter focuses on detailing the design and development of the methodology, building on the foundations laid in earlier chapters. It forms a critical part of the overall research, as it bridges the theoretical groundwork discussed in Chapter 2 and the practical demonstration provided in Chapter 4, ensuring a systematic transition from problem identification to the solution's implementation.

The result is a blueprint that can serves as a guideline to develop an HDSL for a purely TDSL. Each phase of the methodology is described in detail and illustrated by activities and sub-activities involved in each phase. For clarity and guidance, an overview of the methodology is provided in Figure 3.1, and summaries of the intended outcomes for each phase are included.

3.1 Overview

The methodology focuses on developing TDSLs into HDSLs that integrate both graphical and textual elements. This systematic approach enhances usability and accessibility, making DSLs more intuitive for users while preserving the expressiveness and functional integrity of the original textual format

The methodology unfolds through several phases, each tailored to achieve specific outcomes. Initially, in the Decision Phase the feasibility of the transformation is assessed and the scope of developing the existing TDSL into an HDSL is defined. The activities in this phase include evaluating the DSL's current capabilities, identifying user needs, defining enhancement objectives, and selecting the development environment. The result is a clearly defined set of requirements and objectives for the HDSL, aligning the project with user expectations and technological capabilities.

Following this, in the Analysis Phase a detailed examination of the existing TDSL is performed to identify components suitable for graphical representation. This phase focuses on examining domain-specific elements that could benefit from visualization, thereby improving clarity and user interaction. The outcome is a detailed domain analysis that indicates which elements of the DSL will be developed into graphical representations, aligned with the overall goals of the HDSL.

In the Design Phase, an integrated syntax and semantics framework is designed to combine

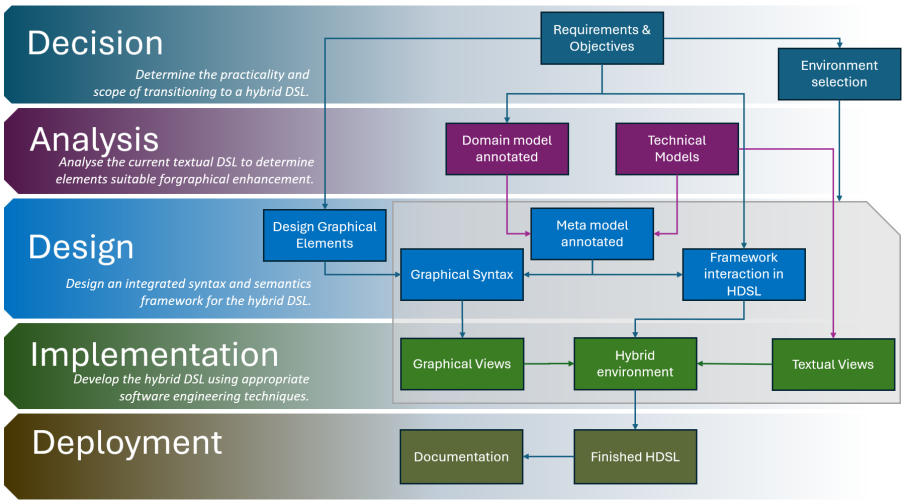

FIGURE 3.1: Overview of the language development methodology

textual and graphical elements. The activities in this phase revolve around designing a coherent structure for the HDSL that supports both graphical and textual syntaxes in a unified environment. The resulting design specifies the interactions between graphical and textual components, ensuring that modifications in one are accurately reflected in the other. Additionally, the notation of the new graphical concrete syntax is designed.

The Implementation Phase is where the designed HDSL is developed into a functional prototype. This involves implementing the HDSL using the selected environment from the decision phase to ensure seamless integration of textual and graphical elements. The result is a fully functional HDSL that allows users to switch between or simultaneously use both graphical and textual components, with real-time synchronization of changes.

Each phase is structured into three sections: goals, activities, and results. The first section outlines the general objectives of the phase. The second section consists of two main activities, which are further divided into various steps. These activities are explained and substantiated in detail. Finally, the results section lists the specific components that should be produced upon completion of the phase. Important to note that some activities may already be completed during the initial development of the TDSL. Consequently, certain steps within the activities may be omitted during the development process if the desired outcomes of those activities have already been achieved.

The overview of the methodology is depicted in figure 3.1. The rows correspond to the various phases in the development process, with the expected outcomes of each phase represented within the blocks. The arrows connecting the blocks illustrate the flow and interdependencies of the components. For instance, the requirements and objectives derived from the Decision Phase are subsequently utilized in the Analysis Phase to identify the key concepts that the HDSL must support, as indicated in the 'Annotated domain model'. Furthermore, these requirements and objectives also inform the design of the graphical elements for the graphical syntax.

3.2 Decision Phase

Goal: *Determine the practicality and scope of transitioning to an HDSL.*

The development of an HDSL from a TDSL must be motivated by the need to address specific challenges that cannot be effectively resolved using only a textual approach. In this phase an assessment is made on whether the hybrid model is feasible within the current technological

and operational constraints and examines the potential improvements it could bring to the users. Objectives and requirements for the implementation should be defined, and a development environment must be chosen.

Activities

TABLE 3.1: The "Enhancement Objectives" activity along with the steps it consists of.

Enhancement Objectives: define what improvements the HDSL aims to add compared to the TDSL.

- Gather feedback from stakeholders and users to identify weaknesses of the current TDSL.
- Evaluate the current DSL tool(s) to identify technological and functional gaps.
- Outline and prioritize enhancement objectives to focus development efforts on key user experience improvements.
- Conduct a literature review to gather requirements.

Defining the enhancement objectives for an HDSL is critical to direct the development efforts towards specific improvements that will increase the TDSL. [29], [5], [14], and [36] all emphasize how these enhancements can streamline the development process and thereby making the HDSL more effective and complete. [6] emphasizes the importance of only undertaking the development of the HDSL with strong business and technical justifications, since the process can lead to unnecessary complications and resistance from users accustomed to the old systems.

The objectives should be clearly defined to ensure that the HDSL meets the evolving needs of its users and leverages the strengths of both textual and graphical components effectively. This is essential for creating a more intuitive and productive environment for developers and domain experts, enhancing their ability to understand, navigate, and manipulate the HDSL efficiently. A proper methodology to define the requirements should be used; examples are the MoSCoW method [2] or Rupp's method [31].

TABLE 3.2: The "Feasibility and Scope" activity along with the steps it consists of.

Feasibility and Scope: determine whether the desired HDSL can realistically be developed within the existing constraints and whether it addresses the needs of its intended users.

- Evaluate existing environments, tools and technology stacks for their capability to support HDSL integration.
- Assess the integration capabilities of frameworks to ensure compatibility in a unified development environment.
- Conduct detailed requirement analysis of technical requirements and user needs to define the scope of the HDSL.

The HDSL must be both feasible and strategically beneficial, aligned with user requirements. Assessing the existing technological landscape helps determine if the current tools and frame-

works can support the nuanced demands of an HDSL, such as the integration of textual and graphical elements, without losing functionality or degrading the user experience [14]. The integration capability of frameworks is essential to ensure that the HDSL leverages the strengths of both textual and graphical components effectively, providing a cohesive user experience and maintaining the functionality of the DSL [36].

Conducting a detailed analysis of technical and user needs is crucial for defining the scope of the HDSL. This analysis helps in crafting a DSL that is technically sound, meets the practical needs of users, and addresses specific operational challenges identified through stakeholder engagement [14].

Results

The decision phase should result in a clearly defined set of requirements and objectives for the HDSL. These requirements should be adhered to, and kept in mind, when processing the rest of the development of the HDSL. Based on the requirements, (a) software tool(s) should be selected to design the HDSL, and an environment to develop the HDSL in should be selected. The design tool and development environment can be the same software, but this is not necessary. This depends on the objectives and requirements of the HDSL. A set of example requirements and objectives, and software tools identified based on the literature, can be found in appendix C.

3.3 Analysis Phase

Goal: *Analyse the current TDSL to determine elements suitable for graphical enhancement*

Critical to the success of an HDSL is a thorough domain analysis, which involves gathering and formalizing knowledge about the domain [18]. For the development of the HDSL, a more specific look into the domain aspects which would benefit from visual representation is needed. The objective is to identify core concepts and operations that would benefit from visual representation, thus simplifying user interactions and improving overall comprehension.

Activities

TABLE 3.3: The "Domain Examination" activity along with the steps it consists of.

Domain examination: examine the domain of the DSL to identify points of interest on domain level.
◦ Retrieve analysis of the domain of the current DSL, if not available: - decide on a domain analysis methodology, or a informal analysis. - extensively analyse the domain, specifically on relevance to the intended graphical enhancements. ◦ Separate key concepts, operations, and relationships within the domain that would benefit from graphical, textual or a dual representation on a domain level.

As with the development of a TDSL, a thorough domain analysis is essential, this time with the primary goal of identifying the elements of the domain that benefit the most from graphical (or dual) representation. If an existing domain analysis is available, it can be utilized for this purpose. Otherwise, this analysis can be conducted either formally or informally, as mentioned

in Section 2.2. A formal domain analysis uses structured methods to systematically capture and model domain knowledge, resulting in precise definitions and relationships. In contrast, an informal domain analysis relies on the expertise of domain experts and sources such as technical documents or interviews, offering more flexibility but less structure. The choice between formal and informal methods depends on the specific needs and constraints of the project.

Examining the domain lays the foundation for the development of the HDSL. [29] and [36] emphasize that the choice between textual and graphical syntax should depend on the nature of the domain elements—textual for detailed, behaviour-oriented aspects, and graphical for visualizing structural relationships and processes. This strategic differentiation can drastically improve the usability and effectiveness of the DSL by aligning the representation mode with the inherent characteristics of the domain content.

TABLE 3.4: The "Technical Examination" activity along with the steps it consists of.

Technical Examination: examine the metamodel and the current concrete syntax of the TDSL to identify points of interest on model level
◦ Collect all existing documentation and models related to the TDSL to form a comprehensive base for the HDSL development. ◦ Retrieve or develop the metamodel. ◦ Identify the elements corresponding to the domain models that would benefit from graphical or dual representation. ◦ Clearly define semantics and grammar of the current syntax related to the elements.

Gathering or developing the TDSL components is essential for ensuring that the HDSL is built on a robust foundation. A well-defined metamodel, along with clear abstract syntax and semantics, provides the structural integrity needed for effective integration of graphical elements. This solid groundwork prevents potential conflicts and inconsistencies that could arise during the transition from textual to hybrid representations, thereby maintaining the HDSL's integrity and functionality [18]. Additionally, identifying elements within the domain models that would benefit from graphical or dual representation allows for targeted enhancements that improve usability and comprehension without overcomplicating the system.

The software used to represent the TDSL components should be considered carefully, since often the software used to develop this can be combined with software used in the design phase. The designing of the HDSL in the next phases will require, among other things, adjustments in these models, so when deciding on the proper environment in the decision phase, both phases should be taken in consideration.

Results

The Analysis phase should result in a clear understanding of the problem domain, which can serve as the cornerstone for understanding the constructs and relationships inherent in the HDSL. From the domain analysis, the points of interest for transformation into a graphical syntax should be identified. Additionally, the metamodel and the current concrete syntax(es) and semantics of the DSL should be drawn up or gathered, combined with notations on the elements which would benefit from graphical or dual syntax the most.

3.4 Design Phase

Goal: *Design an integrated syntax and semantics framework for the HDSL*

The Design Phase focusses on the designing the internal workings and new syntax of the HDSL, which involves defining the appearance of the textual and graphical elements, how coexist and interact within the HDSL. Continuing on the understanding of the domain and its textual representation from the analysis phase, the selected components best fitting the graphical syntax must be redesigned to fit the requirements for the HDSL. These elements must visually represent the underlying domain concepts while still ensuring coherence with the textual syntax.

Activities

TABLE 3.5: The "Framework Design" activity along with the steps it consists of.

Framework Design: define a clear and systematic framework for the hybrid language that specifies the integration points and dependencies between graphical and textual components

- Outline the roles and interactions of each syntax type within the DSL, ensuring effective complementarity and seamless integration.
- Specify the elements in the metamodel which correspond to graphical representations.
- Outline the different viewpoints based on the metamodel and the interactions between them
- Specify the semantics and grammar rules that these graphical parts must adhere to.

The elements in the metamodel should be specified with the type of representation (graphical, textual, or both) to provide a clear development guideline that maintains semantic integrity across modalities. This process ensures that changes in the model are consistently reflected in both textual and graphical representations, which is vital for preserving the HDSL's coherence as it evolves. Additionally, this specification simplifies the development process by providing clear mappings that guide the integration and synchronization of different syntax types [30].

A detailed outlining of roles and interactions within the HDSL ensures that graphical and textual elements complement each other effectively. By clearly defining how these components interact and depend on each other, the framework supports a coherent DSL architecture where every element serves a purpose, enhancing the overall functionality and user experience. For example, a diagram of the outline can be useful to visually represent these interactions, making it easier to understand and communicate the relationships between different components. Based on these interactions, the different viewpoints for the DSL and the subset of the metamodel they cover should be outlined [28]. A top-down overview on the interaction of the viewpoints and syntaxes can clarify the workings. A clear language pipeline architecture to outline the language transformation is a method of clarifying these interactions [6].

While the actual design of the framework depends on the varying needs of each case, it is important to acknowledge existing guidelines that support the development of effective and consistent syntaxes. For instance, Moody [19] provides principles for designing cognitively effective graphical syntaxes, and foundational ontologies help ensure semantic alignment with the domain. Although this thesis does not explore these frameworks in detail, their application is

recommended to enhance the robustness and clarity of both the graphical and textual representations within the HDSL.

Finally, establishing the semantics and grammar rules ensures that the graphical syntax is not only visually intuitive but also semantically correct and aligns with the domain's requirements and operational logic. An overview (either visually or textually) of the rules the relevant elements of the HDSL should adhere to ensures that all elements adhere to the same rigorous standards as the original textual elements, maintaining the overall integrity and functionality of the HDSL.

TABLE 3.6: The "Graphical Concrete Syntax Design" activity along with the steps it consists of.

Graphical Concrete Syntax Design: design the graphical notation of the selected components

- Map out and categorise the different (to be) graphical elements.
- Design graphical elements that reflect these categories, ensuring that each symbol or notation is meaningful and semantically accurate.
- Implement this into design specifications for the concrete graphical syntax.

Map out the graphical elements together with their attributes and features, and categorize them based on the findings from the domain analysis. This ensures that the graphical representations are deeply rooted in the domain's ontology, providing a clear and intuitive reflection of the domain's structure and semantics. This foundation is essential for developing a DSL that is both useful and meaningful to its users, aiding in better comprehension and easier navigation of the domain concepts. For example, different shapes might be used to represent various types of entities, while colours could indicate their states or relationships. This visual differentiation helps users quickly grasp complex relationships and hierarchies within the domain, facilitating easier navigation and understanding. To make the transition to a new syntax easier for developers, it should also look similar to well-known languages [6].

These concrete components should be designed in a meaningful way. For this purpose, the set of visual cognitive principles outlined by [19] can be implemented to establish a cohesive and effective visual notation system. This approach ensures that the graphical elements are designed to be clear, distinct, and intuitive, enhancing the user's ability to interact with and comprehend the DSL efficiently. By applying these principles universally, the design process systematically addresses the cognitive needs of the users, ensuring the graphical syntax is accessible and manageable. Based on these principles, the visual specifications of each graphical element must be designed defined. This includes dimensions, colours, shapes, and any other attributes that were decided upon to represent various domain concepts distinctly and intuitively. The principles can be found in Table 3.7

TABLE 3.7: Principles and Explanations for Designing Graphical Elements

Principles	**Explanation**
Semiotic Clarity	Use unique symbols for each element, avoiding redundancy and overload.
Perceptual Discriminability	Design symbols that are easily distinguishable through shape, colour, and size.
Semantic Transparency	Use intuitive icons that visually represent their function or relation.
Complexity Management	Employ modularization and hierarchy to manage complex information.

Cognitive Integration	Provide navigational aids and consistent views across different diagrams.
Visual Expressiveness	Utilize a range of visual variables to encode information distinctly.
Dual Coding	Supplement graphical elements with text to enhance understanding.
Graphic Economy	Limit the number of symbols to keep the design cognitively manageable.
Cognitive Fit	Design adaptable interfaces that cater to both novice and expert users.

Results

The design phase should result in a detailed framework showcasing the interactions between textual and graphical syntaxes, where modifications in one syntax directly and instantly affect the other. This includes the layout of the mechanisms for synchronization to ensure consistency and real-time feedback within the HDSL environment. Additionally, an overview of the annotated metamodel to specify the representation modes for each element—textual, graphical, or both. These annotations are accompanied by directives on how changes in one representation should propagate to the other, ensuring that the HDSL remains coherent when switching between or integrating both views. The representation of the HDSL elements has been developed, defining the different elements in a visually meaningful way. Finally, combining these results, the visual specifications of the complete concrete graphical syntax is developed.

3.5 Implementation Phase

Goal: *Develop the HDSL using appropriate software engineering techniques*

The objective in the implementation is to achieve the integration between the textual and graphical elements, by realizing the HDSL and integrating it into the selected programming environment. The design and implementation phases can be very close to each other, as some tools used during the design of the HDSL components, automatically generate the components.

Activities

TABLE 3.8: The "Hybrid environment Implementation" activity along with the steps it consists of.

Hybrid environment Implementation: develop the design of the HDSL from the previous phases into a single development environment.

- Implement the designed framework to both support graphical and textual editing.
- Establish rules and mechanisms inside the environment that enforce consistency across graphical and textual representations to prevent discrepancies.
- Customize the environment to fit the stakeholder requirements

The development of a foundational environment that supports the HDSL is essential for the usage of the HDSL. This environment, based on the designed framework from the design phase, ensures that all components of the HDSL are integrated into a cohesive development environment, which should facilitate user interaction and enhance the overall usability of the HDSL. According to [18], having a robust foundational environment helps in maintaining the consistency and integrity of the HDSL, enabling users to switch between graphical and textual

representations effortlessly.

Ensuring consistency between graphical and textual representations is crucial for the functioning of the HDSL. Consistency can be achieved through mechanisms like synchronization with a central repository [28] or parsing the syntaxes against the language's grammar for validation [17]. The semantics identified in the design phase must be implemented to ensure that changes in one representation are accurately reflected in the other, maintaining the overall coherence of the HDSL.

Finally, unique requirements for the environment, defined in the decision phase, should be implemented in the system. Customizing the environment to fit stakeholder requirements ensures that the HDSL meets the specific needs of its users. The interpretation and implementation of these requirements depend on the specific needs and objectives identified during the decision phase. This customization enhances the relevance and usability of the HDSL for its intended users.

TABLE 3.9: The "Development of views" activity along with the steps it consists of.

Development of views based on the designs made in the previous phase, develop the views enabling the concrete syntax
◦ Create a graphical component library based on the design specifications ◦ Implement interactive features that enable users to manipulate graphical elements directly, enhancing the interactivity and engagement with the HDSL. ◦ Integrate visual feedback mechanisms to provide immediate validation of user actions, improving the learning curve and usability of the HDSL.

The graphical component library is a collection of reusable graphical elements, such as icons, shapes, and widgets, that are used to build the graphical interface of the DSL. It serves as a central repository of standardized graphical components, ensuring all graphical elements within the HDSL are consistent with the design specifications. By using predefined components, the development process becomes more efficient, and the resulting HDSL is more cohesive and easier to maintain.

Developing views that allow for the editing of the concrete syntax is essential to the overall goal of creating a versatile and user-friendly HDSL. Views, particularly graphical editors, provide interfaces that facilitate the understanding and manipulation of complex constructs. According to [28], integrating both graphical and textual views enhances the DSL by providing multiple ways to represent and interact with the language's elements. This dual representation supports users in different roles, from domain experts to developers, making the DSL more effective in capturing and communicating domain-specific knowledge.

Results

The result of the implementation phase is a fully integrated hybrid development, synthesizing the graphical and textual components envisioned in the design phase into a cohesive, interactive system. This should allow users to seamlessly toggle between graphical and textual views, ensuring consistency and real-time synchronization of changes across both formats. This hybrid environment includes the additional requirements specified in the decision phase.

3.6 Deployment Phase

the deployment phase entails releasing the HDSL to its users and ensuring its integration into existing systems and workflows. The primary goal of this phase is to ensure that the HDSL operates reliably, meets user expectations, and can be distributed in line with the specified requirements.

However, this research will not delve into the detailed activities of the deployment phase, since deployment is beyond the scope of this thesis. This research focuses on the design, development, and initial implementation of the HDSL. Detailed deployment strategies, including extensive testing, validation, and user training, require significant resources and a tailored approach that are not feasible within the limits of this research. These activities are best handled by dedicated deployment teams within organizations.

Potential activities to be considered in this phase include extensive testing and validation to ensure functionality across diverse scenarios. This involves verifying that the integrated environment operates reliably by performance testing under load conditions, ensuring the editor remains stable and responsive during extended use [36]. Additionally, comprehensive documentation must be created, providing detailed instructions on using the HDSL editor, descriptions of the graphical and textual components, and troubleshooting tips [36]. User training sessions can be organized to help users understand and efficiently use the new HDSL, familiarizing them with both the graphical and textual aspects of the HDSL.

The desired results for the deployment phase include the creation of comprehensive documentation and a complete HDSL. Comprehensive documentation is crucial, providing detailed instructions on using the HDSL editor, descriptions of the graphical and textual components, and troubleshooting tips [36]. This documentation must be clear, accessible, and regularly updated to reflect any changes or enhancements in the DSL. Future work should focus on developing a detailed deployment framework, addressing specific organizational requirements, and establishing comprehensive training and support systems to ensure the successful integration and distribution of the HDSL into real-world environments.

3.7 Documentation

In addition to the overview and phases detailed in this chapter, a comprehensive artifact in the form of a PowerPoint presentation was created to facilitate a better understanding and explanation. This presentation serves as an educational and explanatory tool, providing a visual and structured overview of the entire process.

The PowerPoint presentation [37] includes an overview of the methodology used in this research, outlining the key phases and objectives. It provides the structured descriptions of the various activities undertaken during each phase, offering clarity on the processes involved. This documentation artifact enhances the accessibility and comprehensibility of the research, making it easier for stakeholders, future researchers, and practitioners to understand and replicate the methodology. By providing a clear and concise visual representation of the research process, the PowerPoint presentation supports the communication of the project's objectives and methods. This additional documentation ensures that the methodology is practically applicable, facilitating its adoption and implementation in real-world scenarios.

Chapter 4

Case study

The chapter demonstrates the practical application of the HDSL methodology through a case study involving the Ampersand platform. We start by introducing the Ampersand platform, describing its purpose, functionality, and the DSL it uses. The case study then outlines the process of developing an HDSL for Ampersand by using the methodology from Chapter 3. This chapter illustrates how the phases were executed, detailing the decisions made and the tools used. The result is the implemented Ampersand HDSL that integrates both textual and graphical components. This case study provides an example of how the proposed methodology can be applied in practice.

4.1 Ampersand

Ampersand is an open-source tool designed to automate the process of software development for information systems, making use of the requirements of the intended system. The tool requires input on the desired information system in the form of its own formal language called Ampersand language. This syntax is based on the ideas of relational algebra and combines formal specifications and natural language, to allows users to define complex business rules and relationships between data. From the input, a complete and functioning information system is generated in the form of a monolithic, web-based application equipped with a structured database (Figure 4.2). The architecture of this created information system can be easily altered by simply changing the input code and compiling a new system.

The system can be used to easily design an information system for practical use, but it also provides an efficient way to quickly develop a prototype. Currently, both students and developers utilize this prototyping capability to explore potential structures for information systems and test their functionalities without investing significant time in developing them. In educational settings, it is used as a teaching tool to help students understand the principles of information system design and formal specifications. In professional environments, developers leverage Ampersand to prototype and validate information systems efficiently, ensuring that business rules and relationships are correctly implemented before full-scale development

4.1.1 Architecture Overview

The Ampersand platform has a single input mechanism to instruct the development platform on the desired application, namely textual syntax. By using a DSL based on formal specification written in Ampersand Language scripts (with file extension .ADL), the desired application can be specified. The language is a declarative, meaning developers define “what” results they want without specifying “how” to achieve those results. The scripts ensure that the database maintains certain conditions or rules, automatically handling data consistency through generated code that

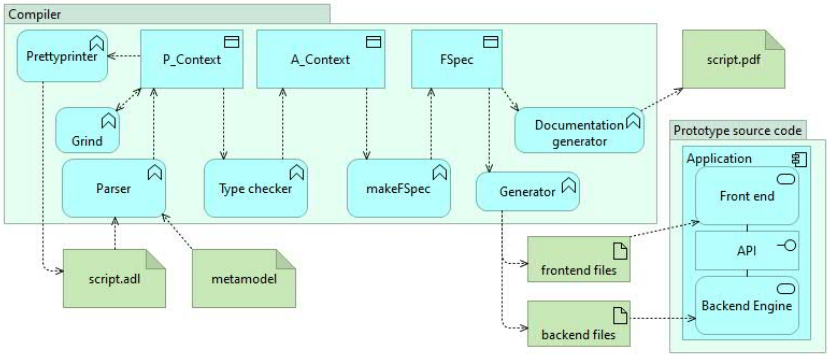

FIGURE 4.1: Overview of the Ampersand platform.

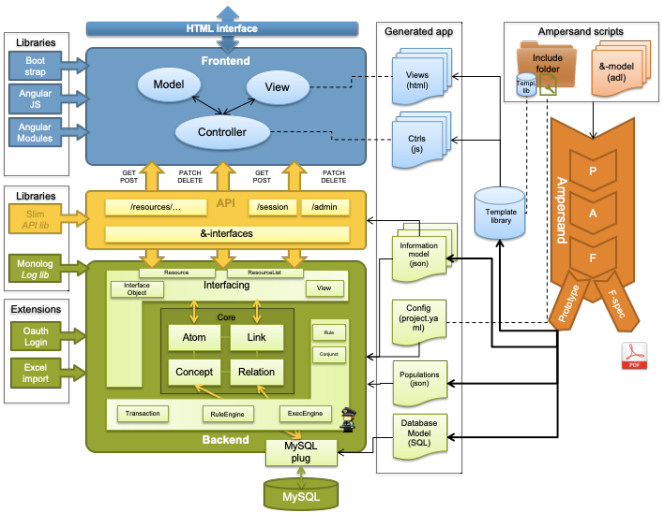

FIGURE 4.2: Overview of an application generated by the Ampersand platform.

preserves these rules during data manipulation tasks in the database. At compile-time, Ampersand processes the script to set up the database structure and initial rule validations. During runtime, interactions with the database (such as updates and queries) are managed to ensure all rules are continually satisfied, rolling back changes that violate rules [13].

Ampersand is rooted in relational algebra, a branch of Mathematics that deals with the theory of relations and the operations that can be performed on them. The language itself allows for the specification of information systems in a formal, precise, and unambiguous manner. This formal specification ensures that the behaviour of the system is well-defined and verifiable. The core of the language is composed of a set of rules, relations, and concepts, represented as a triple ⟨H,R,C⟩

- *Concepts (C):* The fundamental types of data items within the system, which are used to classify data elements into meaningful categories. For example, a concept might be "Person" or "Order". Concepts represents entities within the database, essential for defining the structure and types of stored data. They are expressed via *CONCEPT ConceptName "Definition"*.

- *Relations (R):* Define how concepts are related to each other. A relation is a set of ordered pairs, each representing a connection between instances of two concepts. Relations are used for modelling the interactions and dependencies within the system. They are expressed via *RELATION RelationName[Concept1*Concept2] [PROP] MEANING "Description"*.

- *Rules (H):* The constraints or conditions that must be satisfied by the data in the system. They are expressed as logical statements involving concepts and relations, which ensure the integrity and correctness of the data by enforcing specific conditions that must always hold true. They are expressed via RULE RuleName : expression MEANING "Explanation".

These triples are used to construct the underlying algebraic structure that powers the Ampersand platform. The algebraic implementation of these triples is the driving force behind the generator, enabling it to manipulate sets and relations effectively. Functions applied to these relations can, for example, remove pairs from relations or modify the structure of data to ensure consistency and adherence to the specified rules. This approach allows the generator to maintain the integrity of the information system automatically. By leveraging relational algebra, Ampersand can perform operations such as selection, projection, and join. These operations are abstracted in the Ampersand language, allowing users to focus on the high-level design of their information system without needing to understand the underlying mathematical details.

4.1.2 The RAP Environment

The Rapid Application Prototyping (RAP) is a component of the Ampersand platform used to make the input mechanism of Ampersand accessible. This web-based application facilitates a method for designing and compiling .ADL scripts by using the .ADL Script Editor to allow developers to modify the .ADL scripts directly, facilitating rapid prototyping and iterative refinement. Changes made to the script are reflected in the generated application upon recompilation, enabling developers to quickly incorporate feedback and adjust specifications. The ATLAS Visualization Tool provides a graphical representation of the application model, illustrating the relationships and structures defined in the .ADL script. This visualization helps developers and stakeholders understand the system's architecture and verify if it meets the specified requirements.

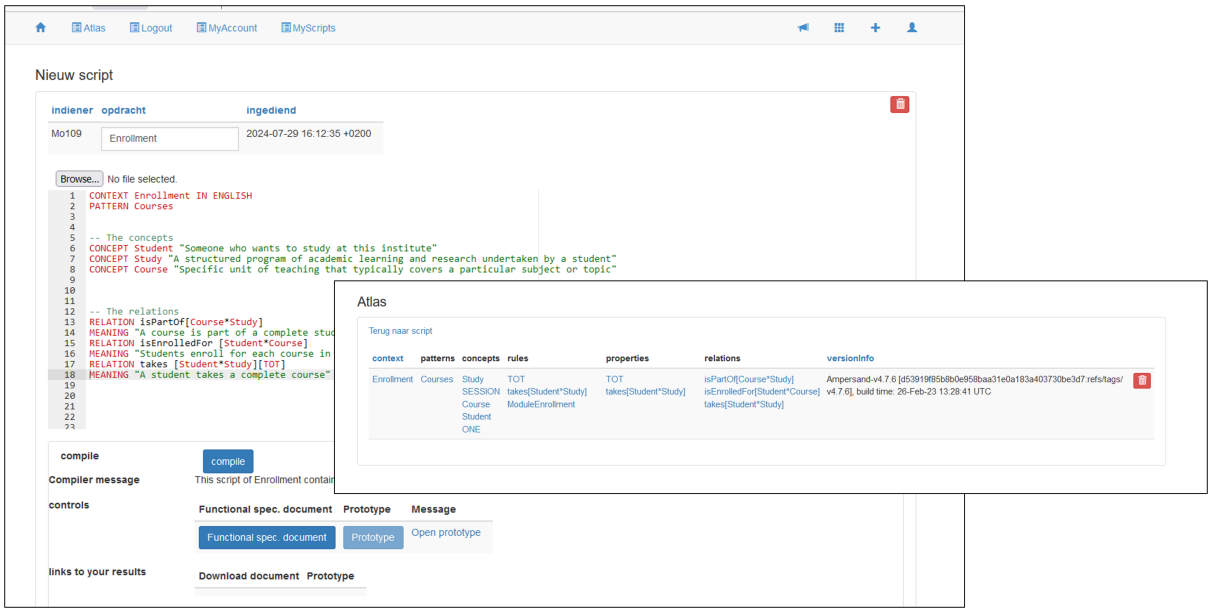

FIGURE 4.3: Interfaces in RAP: the script editor (back) and the ATLAS (front)

RAP is a highly advanced and customized system generated by the Ampersand platform itself. The workflow within RAP involves several key steps, each contributing to the efficient development and deployment of a prototype. Developers begin by editing the necessary scripts using the RAP editor tool. The tool provides a plain text interface for creating and modifying .ADL scripts. RAP uses the Ampersand platform's compiler to transform the .ADL scripts

into a prototype, along with two key outputs: the functional specification document and ATLAS. The functional specification document provides a breakdown of the system's architecture, highlighting its core components, relationships, and business rules summarised in a document. ATLAS generates graphical representations of the system's architecture visible in the RAP application itself. During compilation, users have the flexibility to either generate the complete working prototype or, if needed, focus solely on producing the functional specification document and ATLAS to review and refine the system's architecture.

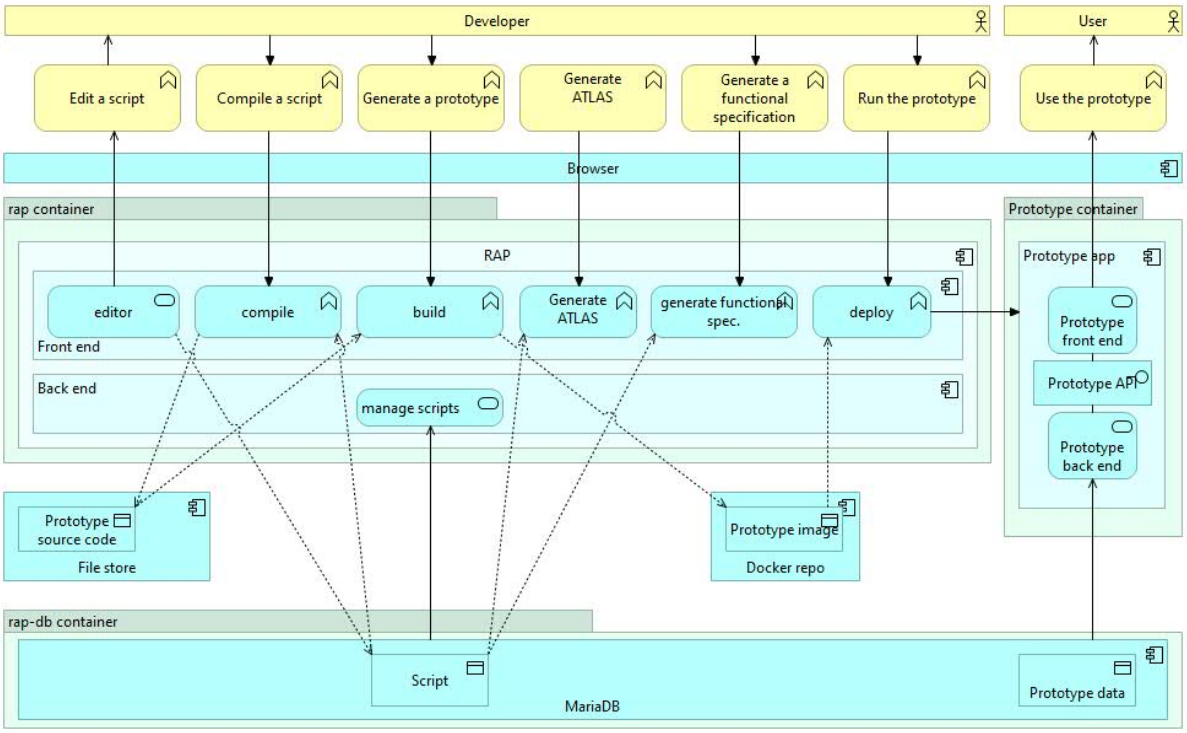

FIGURE 4.4: Overview of the workings of the RAP environment

When the scripts are ready, the prototype can be generated by RAP. The deployment sets up the necessary containers to host the application components, ensuring they are properly configured and ready for use. The prototype container includes the frontend, backend, and associated database component. End-users can now interact with the deployed prototype through the provided web interfaces. These interfaces allow for data entry, updates, queries, and reporting, all governed by the rules and constraints defined in the .ADL scripts.

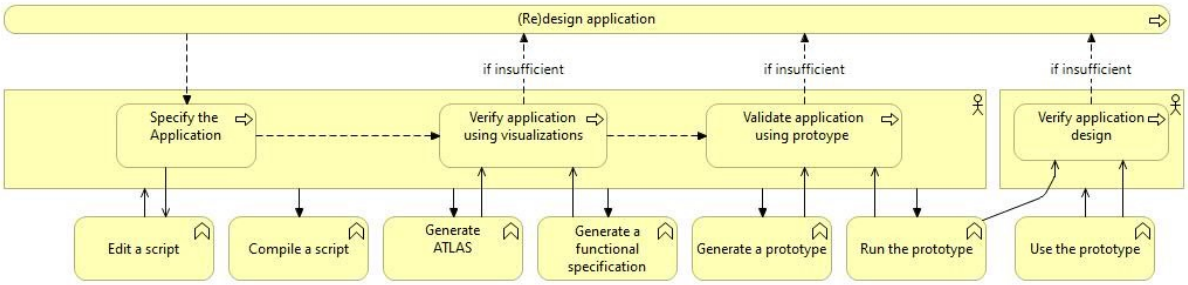

FIGURE 4.5: RAP usage workflow

4.2 Developing an HDSL for Ampersand

Ampersand struggles with the issue that users find it too difficult to use. Since text based systems can be seen as a complex development mechanism [9]; the now textual syntax based Ampersand language must be developed to HDSL.

The Ampersand platform currently has a component, the ATLAS, which visualizes the elements of a formal specification for users. The main goal of this transformation is to design and develop into a graphical language editor for Ampersand, allowing users to create functional specifications without having to use them directly.

4.2.1 Decision Phase

Goal: *Determine the practicality and scope of transitioning to an HDSL.*

Enhancement Objectives

To establish the enhancement objectives for Ampersand, we performed a brief literature review focused on understanding how an interactive editor can be designed to make formal specifications more accessible. This review was essential as the formal specifications need to be converted from textual syntax to a graphical format. An overview of this literature review can be found in the Appendix B. The requirements identified from the literature review were expanded and merged with the example requirements and objectives mentioned in Appendix A. This was extended with requirements arisen from an investigation into the workings of Ampersand. This list was defined using the MoSCoW method as described by [2]. An excerpt of the most relevant requirements included below, categorised in functional and non-functional requirements. The full list can be found in Appendix C.

TABLE 4.1: Excerpt of the Functional (F) and Non-functional (NF) requirements

Sig	Requirements	Justification
F1	The system must become less difficult for user to use	This is the main reason why the HDSL is developed
F2	The system must enable a unified compilation process.	This ensures synchronization and compilation of changes across formats.
F3	The system must support hybrid-syntax editing	Both the graphical and textual syntaxes should be separately adjustable [36] [29].
NF1	The existing compiler must be used to handle inputs from both graphical and textual sources.	This ensures synchronization and compilation of changes across both formats
NF2	The system must use Graphviz to visualize the graphical syntax	This application is already in use by the environment
NF3	The system must use the existing RAP environment	This leverages existing infrastructure and reduces the need for additional training

Feasibility and Scope

Previous developers of the TDSL were consulted to examine the integration frameworks and environment. For the Ampersand, this environment is RAP. Since one of the requirements we captured explicitly mandates the use of RAP, alternative technologies were not considered. Furthermore, it was decided that the implementation of the HDSL should only encompass a prototype to ensure a manageable scope for the initial implementation and to focus on the aspects of the TDSL that would benefit the most from a graphical representation.

The integration capabilities of the frameworks were investigated into by developing an overview

of both the workings of the RAP environment (Figure 4.4) and the Ampersand platform itself (Figure 4.1). This overview was developed based in cooperation with previous developers. The results of this investigation can be seen in Section 4.1. Additionally, the desired workflow of the usage of RAP was developed with help of an Ampersand user (Figure 4.6). Combining this desired workflow and the current workings of RAP, the shortcomings of the current environment were examined.

Analysis of assessment

The decision phase resulted in a detailed assessment of both the current and desired functionalities of the RAP environment for the successful implementation of an HDSL. Figure 4.6 presents an overview of the desired workflow for RAP. When compared to the current operational overview shown in Figure 4.5 certain enhancement requirements were apparent, which are added to the requirements list. Most importantly, it is necessary to develop interfaces that allow simultaneous interactions with both the graphical component (ATLAS) and the textual editor within the RAP database. This ensures the user can simultaneously but independently make changes graphically and see those changes reflected in the textual script (and vice versa), thereby maintaining the two representations consistent and synchronised.

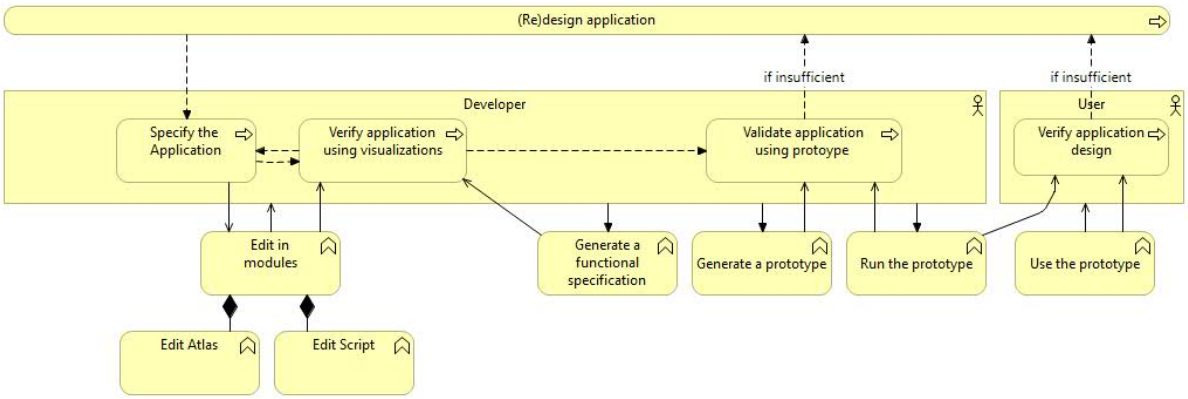

FIGURE 4.6: Desired development workflow using the RAP environment

RAP uses textual input to populate the database, which subsequently generated prototypes and visual representations. However, for the system to support two-way synchronization, it is necessary to implement functionality that allows for the generation of the textual scripts directly from the database. This ensures that both textual and graphical representations can be consistently derived from the same underlying data model. This capability would ensure that any modifications made in the database, whether through graphical or textual interfaces, can be automatically updated in the script. This integration is vital for maintaining the integrity and accuracy of the system, as it allows for a seamless flow of data and rules across different components of the DSL. An important additional finding is the integration of Graphviz in the Ampersand platform. Graphviz is a visualization tool which enables the automatic generation of schematic diagrams that represent Ampersand scripts.

The final product of this phase is a set of functional and non-functional requirements, based on the user and stakeholder feedback, as well as comprehensive literature review findings. Each requirement is assessed based on the value towards the main business goals using the MoSCoW analysis technique [2]. Additionally, an explanation of its significance and the sources that justify its inclusion provided. This structured presentation aids in ensuring a holistic view of what is needed for the successful development and implementation of the HDSL. The resulting lists

can be found in Appendix C.

4.2.2 Analysis Phase

Goal: *Analyse the current TDSL to determine elements suitable for graphical enhancement*

Domain examination

A comprehensive examination of the existing domain of Ampersand was conducted to identify the key concepts and operations that would benefit from graphical representation. This process coincided partly with the research from the decision phase, and included engaging with users of Ampersand to gather insights into their experiences, challenges, and expectations for the HDSL. This engagement provided valuable practical insights to how users interact with the system, and identified areas where graphical elements could enhance usability. The Ampersand platform (and mostly RAP) was used to gain first-hand experience of its functionalities and user interactions. This practical usage ensured a deeper understanding of the workings of the program. Finally, the existing literature and documentation [25] on the Ampersand platform was studied extensively. A summary of this research can be seen in section 4.1. The result of this investigation was a comprehensive understanding of the language, and the selection of core elements in need of graphical representation.

Technical examination

The documentation review also served to deepen the understanding of the technical aspects. Additionally, using the Ampersand platform, the metamodel was automatically generated with Graphviz, accurately reflecting the current state of the TDSL and capturing all relevant components and relationships. This metamodel was then analysed to pinpoint the key elements related to the database, as identified during the domain examination.

The concrete syntax of Ampersand, as detailed in the official syntax reference, was reviewed. This documentation provided a clear explanation of how the core elements are expressed textually within the DSL. The insights from the concrete syntax documentation were then combined with the generated metamodel to provide a precise breakdown of the core elements identified during the domain examination, both in their textual form and potential graphical representation. Additionally, the grammar of the textual syntax was investigated to ensure the resulting HDSL adheres to the same rules.

Analysis of assessment

Through a detailed analysis conducted by the researcher, it became evident that certain elements—specifically those defining the database structure and enforcing its rules—would benefit from graphical representation. This conclusion was reached based on user feedback and technical evaluations, which suggested that a graphical interface for those components would improve usability by simplifying complex interactions and providing a clearer, more intuitive visualization of the system's core functionalities. The selected components from the metamodel include concepts, relations, and rules. These elements correspond to specific statements in Ampersand's concrete syntax. An example of textual input for Ampersand can be seen in Figure 4.7.

The semantics and grammar rules related to these elements have also been analysed and summarized. These include, but are not limited to:

- A Concept must have exactly one Name.
- Each Relation must have a signature consisting of a source and target concept

```
11  CONCEPT NameOfConcept "This is the definition of a CONCEPT"
12
13  RELATION nameOfRelation [Concept1*Concept2] [PROP]
14  MEANING   "this is the meaning of the RELATION nameOfRelation"
15
16  RULE NameOfRule: relation1 |- relation2;relation3~
17  MEANING "this is the meaning of a RULE, specifically -NameOfRule"
18  MESSAGE "this is the message desplayed when this rule is violated"
```

FIGURE 4.7: Example of textual input for Ampersand

- Each Rule must have a unique name (RuleName).

A description of these components and their respective sections of the metamodel, concrete syntax, and semantics can be found in appendix D.

4.2.3 Design Phase

Goal: *Design an integrated syntax and semantics framework for the HDSL*

Framework Design

The first objective in the Design Phase was to establish an effective framework for the Ampersand HDSL that integrates both textual and graphical components through a unified backend, rather than direct interactions between the concrete syntaxes. This approach aimed to leverage the strengths of each modality while ensuring system integrity and coherence. Based on the insights gathered during the Analysis Phase, it was decided that the HDSL would be designed with two separately operating syntaxes, instead of the textual and graphical directly being connected. These syntaxes interact indirectly through shared modifications in an underlying database. This design choice minimizes the complexity associated with directly linking graphical and textual representations, facilitating maintenance and scalability. A simplified version of the design of the workings of the syntax and the eventual design of our implementation can be seen in Figure 4.8.

While the framework ensures semantic correctness and usability, it should be noted that no formal ontology was explicitly applied to guide the semantics of the HDSL during the design phase. The semantic integrity was maintained through the internal consistency of the DSL's elements, but a foundational ontology was not referenced in this case study. The focus was more on the practical implementation of the DSL, with an emphasis on making sure that textual and graphical elements remained synchronized and coherent.

While the framework ensures semantic correctness and usability, no formal ontology was explicitly applied during the design phase to guide the semantics of the HDSL. Semantic integrity was maintained through internal consistency within the DSL, but a foundational ontology was not referenced in this case study. The focus remained on practical implementation, ensuring synchronization and coherence between textual and graphical elements. Similarly, while principles of effective syntax design were important, frameworks such as Moody's [19] were not directly applied. Usability and clarity were emphasized, but the use of formal syntax design principles was implicit rather than structured. These guidelines could be considered in future iterations to further improve the cognitive effectiveness of the graphical syntax, especially for more complex use cases.

The textual and graphical syntaxes interact indirectly via model transformations handled by the compiler. Changes made in the textual editor are compiled into an intermediate model, a data structure stored in the database, which serves as the foundation for both syntaxes. The data in the database acts as the central source of truth, storing the system's state, and ensuring consistency between the two representations. When modifications occur in the graphical interface, they are translated back into changes in the database, which can then be reflected in

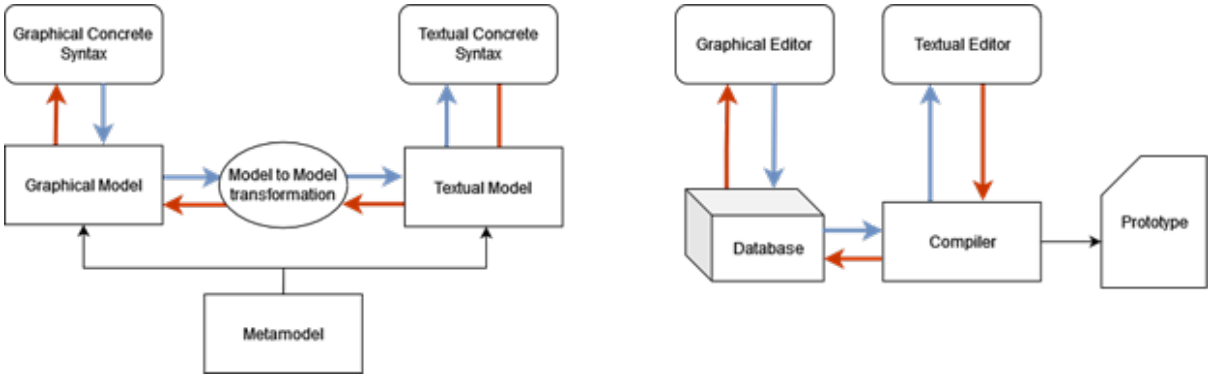

FIGURE 4.8: Schematic overview of the interaction between the separate syntaxes and views

the textual syntax upon recompilation. This ensures that both representations are synchronized and compliant with the system's semantics. The compiler is responsible for ensuring that the rules and constraints defined in the textual syntax are preserved in the graphical syntax, and vice versa. As a result, even though the content in the graphical interface is generated from the underlying model, the consistency, and integrity of the system are maintained across both interfaces. As outlined in the decision phase, one of the requirements is the ability to generate textual scripts directly from the database, enabling two-way synchronization. This enables any modifications made within the graphical interface to be reflected into the textual scripts, maintaining consistency across both syntaxes. Further details on the development of this functionality are provided in the Appendix E.

Graphical Concrete Syntax Design

The graphical elements for Concepts, Relations, and Rules, identified during the analysis phase, were designed during the metamodel analysis. Following the principle of modularity, as suggested by [14], the graphical syntax was developed with two distinct but related views. The first syntax provides a high-level overview of the entire project. This project-level view simplifies the understanding of the system by clearly illustrating the relationships and interactions between key elements. With functionality similar to that of an ER diagram, it offers users a broad perspective, making it easier to comprehend the overall structure.
The second syntax offers a more detailed view, focusing on individual elements and closely aligning with the textual syntax to support development tasks. This component-level view is designed to visually represent the relationships between the elements within the script, helping users better understand how different concepts are interconnected

Both syntax designs build upon the foundational work of previous developers and have been refined to address the current requirements of the HDSL. The dual approach aims to make the HDSL both comprehensive and user-friendly, accommodating the diverse needs of its users. The graphical notation for both views adheres to the design principles outlined in Table 3.7.

Analysis of assessment

The design of the ATLAS tool is organized into two main areas: sub-views for each individual element (for each Concept, Relation, or Rule) and an overview of the entire system. The sub-views (categorised by the type of element) include both an editor and a visualizer. These views allow users to work on the different parts of the final script independently. The overarching view summarizes the entire system, integrating all the sub-views. Changes made in the graphical interface are compiled if they meet the semantic requirements, and from there, the textual syntax

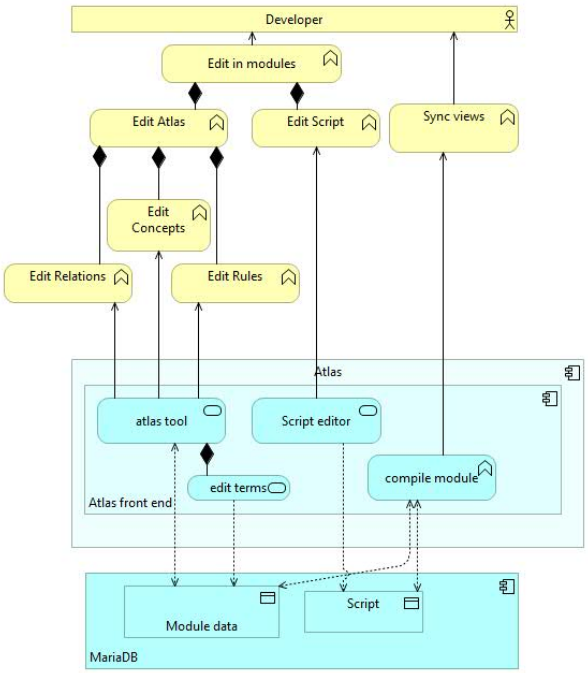

FIGURE 4.9: Detailed design of the workings of the ATLAS tool in RAP

is generated. This ensures that any adjustment made in one syntax is immediately reflected in the other, maintaining synchronization. A simplified illustration of ATLAS's operation is shown in Figure 4.9.

This integrated system allows developers to edit concepts, relations, and rules within the ATLAS tool, either graphically or textually. The synchronization between these views ensures that any update in one view is consistently reflected in the other, providing a seamless and efficient development experience. The architecture also supports the validation of changes, ensuring that only semantically correct modifications are propagated across both syntaxes, maintaining the integrity and functionality of the DSL.

Syntax

The graphical notation for the syntaxes was built on the existing graphical syntax generated by Graphviz, ensuring that all key elements and relationships were accurately captured. To achieve a complete syntax for the Ampersand Language, the existing graphical notation was extended to include missing elements. For example, additional notations for relationships within the conceptual models were added based on established design principles. This ensures that the graphical representation is comprehensive and capable of depicting all relevant relationships within the DSL.

Two different visualizations were chosen to meet the distinct needs of users at different levels of interaction with the system. The project-level syntax provides a high-level overview of the entire project. This view shows all concepts and their relations in a simplified format, focusing on how they fit into the larger system architecture. Here, the same elements—such as concepts and relationships—are visualized in a more abstract and generalized way to highlight their interactions and structural roles. This broad overview is particularly useful for system architects or users who need to understand the overall flow and dependencies within the system. Figure 4.10 demonstrates this, where the focus is on depicting key relationships and properties clearly and simply.

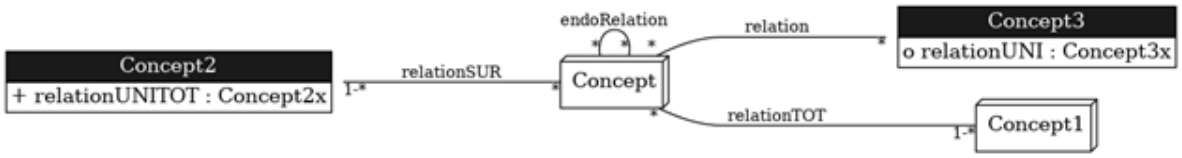

FIGURE 4.10: Example of the Project Level Syntax

In contrast, the component-level syntax offers a more detailed, granular view of the same elements. At this level, concepts and relationships are visualized with all their specific attributes and detailed interactions. This view is essential for users directly involved in development, where understanding the exact behaviour and configuration of each element is critical. In this syntax, relationships between components are shown with greater specificity, such as cardinality, constraints, or other properties that need to be addressed during detailed development tasks. Figure 4.11 illustrates how these detailed relationships and interactions are visualized, helping users to edit and refine the system accurately.

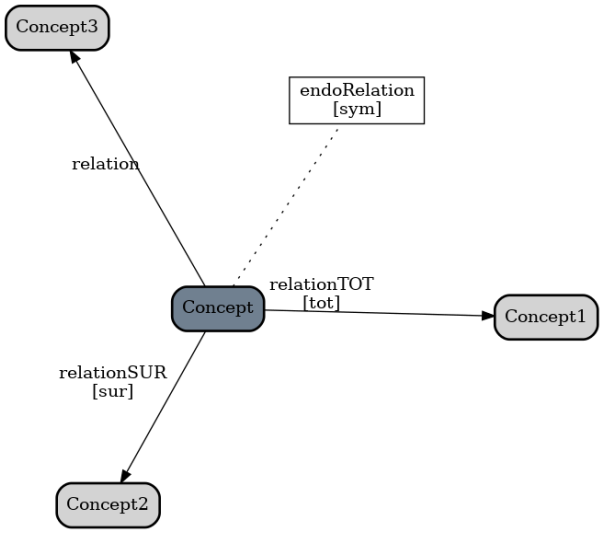

FIGURE 4.11: Example of the Detailed Syntax

The reason for visualizing the same elements differently across these two syntaxes is that they serve different purposes: at the project level, users need an abstracted, high-level view to focus on overall structure and interactions; at the component level, users need precision and detail to effectively manage the intricacies of individual elements. This dual representation allows users to engage with the system at varying levels of abstraction depending on their task, ensuring both a high-level understanding of the system's architecture and the ability to perform detailed, component-specific work.

By visualizing the same elements differently, the system remains flexible and adaptable to various roles and needs, providing both a bird's-eye view for planners and detailed insights for developers. Furthermore, the design leveraged existing distinctions between relationship properties. By clearly differentiating these properties in the graphical syntax, the design phase ensured that the visual elements are both informative and easy to interpret.

4.2.4 Implementation Phase

Goal: *Develop the HDSL using appropriate software engineering techniques*

Hybrid environment Implementation

In the first part of the implementation phase we developed the environment by creating transformation functionalities in both RAP and the Ampersand platform, as show in an overview in Appendix F. This comprehensive effort resulted in the environment depicted in Figure E.1. Following this, the identified rules and mechanisms for consistency enforcement were developed within the RAP environment.

To ensure smoother interface operations, the grammar was extended to include rules that automatically populate elements of the graphical syntax, facilitating adherence to semantic rules. For example, if every concept must have a definition, a definition is automatically created when the concept is created. The user is then notified that this definition needs to be completed. This approach maintains semantic integrity without compromising user convenience. Additionally, error messages when the semantic integrity is violated were implemented, to notify the user of their error. These rules were added to RAP using the existing rule framework of the environment. An extensive explanation can be found in the Appendix F. These rules also were used to implement the environment customizations required from the requirements.

Development of views

Based on the designs made in the previous phase, the interfaces enabling the concrete syntax were developed. The graphical syntax, developed in Graphviz, was implemented making use of functionalities already in the existing structure of the environment. The graphical components are defined in the Graphviz generator, which serves as the graphical component library. Diagrams of the implemented graphical syntax are automatically generated during the compilation of the new RAP database.

In addition to the diagrams, separate editors were developed for each of the core elements: Concepts, Relations, and Rules. These editors were implemented across three distinct views, with each view dedicated to one specific element. The editors were designed to ensure that all the features available in the textual syntax are also fully supported in the graphical interface, allowing users to manage each element—Concepts, Relations, and Rules—with the same level of functionality. The editors can be seen next to the graphical notation in the viewpoints in Figure 4.13 and Figure 4.14.

Visual feedback mechanisms were implemented using the existing message system of the Ampersand Rules. These mechanisms provide immediate validation of user actions, ensuring that any modifications adhere to the defined rules and enhancing the overall usability of the system.

Analysis of assessment

The implementation phase resulted in a fully functional RAP environment that integrates both the existing textual syntax editor and the newly developed graphical interfaces, in line with the design objectives outlined in Chapter 4 and the blueprint detailed in Appendix E.

- The first module is the existing textual syntax editor, which has remained unchanged. This editor continues to provide users with a reliable and familiar interface for textual DSL development. This view can be seen in figure 4.3

- The second module is the Global overview in the ATLAS tool, providing a comprehensive view of all elements of the desired application. It includes a linked table listing all individual components, allowing users to easily navigate and manage different parts of the application. This view serves as an entry point for accessing detailed information about each component. This view can be seen in figure 4.12.
- The third module is the detailed editing environment, which is organized into three tabs, each dedicated to each specific element type: CONCEPT, RELATION, and RULE. Users can navigate these tabs to define and edit these elements, utilizing the detailed graphical syntax designed for each type. This graphical syntax allows for an intuitive and visually clear representation of the DSL components, enhancing user interaction and understanding. This view can be seen in Figure 4.13 and Figure 4.14.

Additionally. the integrated error message system of the RAP environment has been utilized to make users aware of potential errors or missing elements during the development process. This system provides real-time feedback, displayed prominently at the top of the page, ensuring that users can address issues promptly and maintain the integrity of their DSL models.

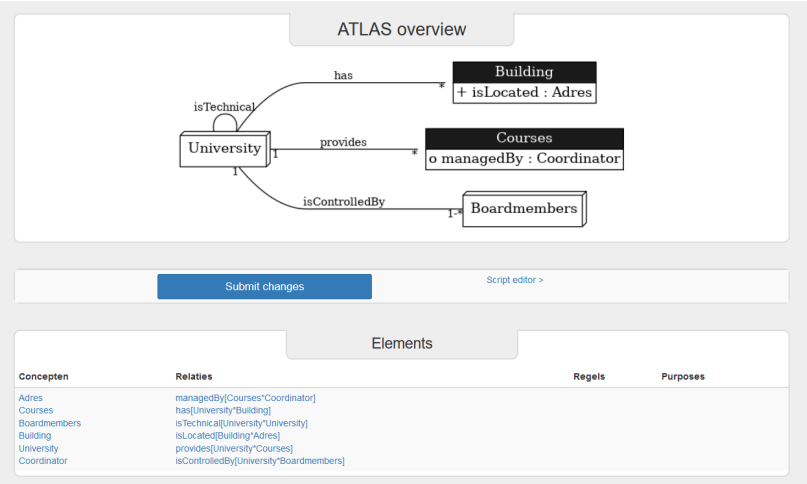

FIGURE 4.12: The RAP overview module

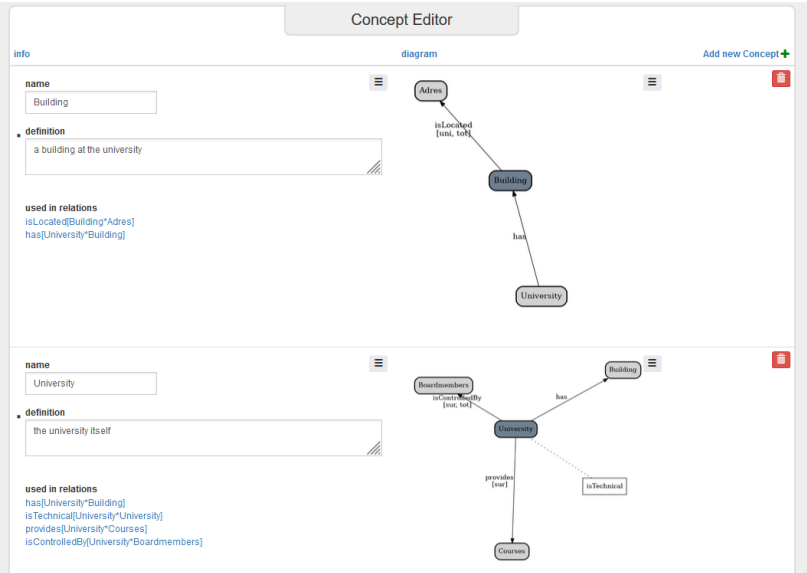

FIGURE 4.13: The RAP detailed viewpoint for CONCEPT

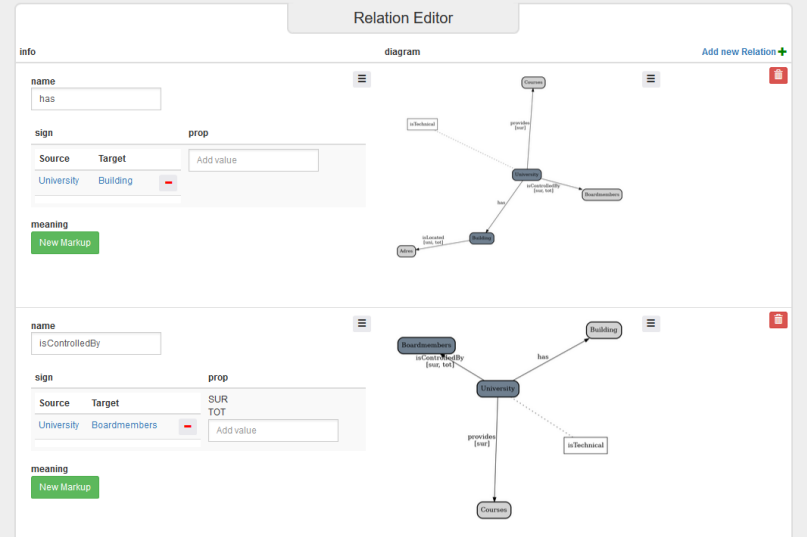

FIGURE 4.14: The RAP detailed viewpoint for RELATION

Figure 4.12, Figure 4.13 and Figure 4.14 provides an example of how a university system can be modelled, by defining key entities and the relationships between them. Concepts include Courses (university courses), a Coordinator (who manages courses), Buildings, and the University itself. The relationships describe how these entities interact. For example, a University can have multiple Buildings, and each building must be located at a specific address, ensuring that every building has an address, and each address is unique. Similarly, each course is managed by a coordinator, with the UNI constraint ensuring that a course is managed by only one coordinator. In essence, this script establishes a structured model of a university, detailing how entities like courses, coordinators, and buildings relate within the system.

4.2.5 Deployment Phase

In this research, no specific activities were deliberately undertaken to fit into the deployment phase. The primary focus was on the design, development, and initial validation of the HDSL. However, recognizing the importance of deployment, some preparatory steps were naturally incorporated during the implementation phase to facilitate future deployment efforts. A somewhat deployable version of the new RAP environment was developed to be used for the usability test described in Chapter 5. This version ensured that the HDSL could be practically tested and evaluated, providing valuable feedback for further refinement.

Additionally, the case study described in this thesis serves as a form of documentation on the development of the Ampersand HDSL. By detailing the process and outcomes of developing and testing the HDSL, this Chapter 4 provides valuable insights and guidance that can support future deployment efforts.

Chapter 5

Evaluation

This chapter evaluates the HDSL methodology and its implementation, assessing both the theoretical framework and its practical application. Given the scope and constraints of a master's thesis, comprehensive validation through multiple system implementations is not feasible. Therefore, the methodology has been reviewed by domain experts for potential improvements and validated through usability testing of the case study results. This usability test will evaluate the suitability of the HDSL produced using the methodology compared to the original TDSL, The result of this is an evaluation highlighting the strengths and weaknesses of the HDSL methodology. These insights provide valuable feedback for refining the methodology and enhancing the usability of the HDSL

5.1 Expert interviews

The expert interviews aimed to review the proposed methodology. For this, we gave a presentation to the expert, in which we also explained an overview of the methodology and gave a detailed description of the activities. Additionally, we also explained the steps undertaken in the case study. This allowed the experts to gain a better understanding of the implementation process. The interviews were designed to be open-ended: guiding questions were prepared to steer the conversation slightly, but the core idea was to allow the experts to provide unprompted feedback. These guiding questions were intended to steer towards feedback on the quality of the content. The following questions were posed:

- Would you update any activity or steps in the artifact? If so, please explain what and why.
- Is the method explained of the presentation understandable, assuming the reader has a basic understanding of the concepts? (Understandability)
- Does the method explained in the presentation sufficiently cover the development of an HDSL? (Sufficiency)
- Is the method explained the presentation useful for understanding the development of an HDSL? (Usefulness)

The feedback gathered from the expert interviews was analysed to identify areas for improvement in the methodology. The experts' insights helped refine the process and ensure its practical applicability and robustness. The expert interviews were conducted with two professionals who are experienced in the development of a DSL, namely Ampersand. Both experts have extensive knowledge and practical experience with the Ampersand platform, which facilitated a smooth understanding of the implementation and allowed for more in-depth discussions regarding the strengths and weaknesses of the current methodology.

We received the following feedback:

- *Introduction of the Artifact*

Both experts noted that the artifact itself needed a better introduction. They recommended that the ultimate goal of the methodology should be clearer upfront, including a concise explanation of what HDSL and TDSL mean. They suggested that the purpose of the different steps in the methodology should also be explicitly stated at the beginning to provide context and improve understandability. Additionally, for the PowerPoint, they recommended creating a table that matched concepts with their concrete instances in Ampersand to enhance clarity.

- *Logical Sequence of Steps*

The experts agreed that all steps in the methodology were logical and followed a clear sequential order. This structured approach was appreciated, as it provided a coherent flow from one phase to the next. However, they emphasized the need for a more detailed explanation of each step's purpose and how it contributes to the overall goal of developing an HDSL.

- *Validation in Methodology*

A significant point raised by both experts was the absence of a specific step aimed at validating the HDSL in the methodology. While the methodology ensures the requirements for the HDSL are defined in the decision phase, there was no clear mechanism to demonstrate that the resulting HDSL meets those requirements. They pointed out that a feedback loop is essential to ensure that the HDSL developed aligns with the initial goals and user needs. One expert suggested incorporating a validation phase where the developed HDSL is tested against the defined requirements, with adjustments made as necessary based on feedback.

- *Agile Approach*

Continuing on the need for validation, one expert recommended designing the methodology with an Agile approach in mind rather than a traditional waterfall model. They argued that using short-cycle user stories could facilitate continuous feedback and iterative improvement in the development process of the HDSL. By structuring the methodology to encourage this approach, it would allow for regular validation of progress against the requirements, ensuring that the development stays aligned with user needs and expectations throughout the process.

5.2 Usability Test

The usability test has been conducted by engaging software developers and domain experts to test result of the case study. To quantify user satisfaction, an enhanced version of the System Usability Scale (SUS) by Brooke [3] has been utilized. The SUS is a widely used tool that provides a reliable measure of system usability, which consists of a ten-item questionnaire with five response options ranging from "Strongly agree" to "Strongly disagree." The SUS yields a single score representing a composite measure of the overall usability of the system. This score helps compare the usability of different systems and identify areas needing improvement [3].

In their research, [35] developed a framework that addresses some limitations of the traditional SUS by incorporating additional criteria from ISO 9126 and ISO 9241-11 standards, which are

crucial for evaluating the user experience in an interactive editing environment. Their version of the SUS adds nine additional questions to the survey, aimed at measuring the efficiency, effectiveness, usability compliance, and attractiveness of the system. This is expected to provide a more complete assessment of the renewed tool [35].

To ensure a proper evaluation, both the TDSL and the HDSL tool has been assessed using the Enhanced SUS survey. The test subjects were asked to perform the same tasks on both systems, after which they were filled in the survey and explain their choice. The results from these assessments were compared to identify improvements and areas that still require enhancement. By comparing the usability scores of the old and new versions, we can quantify the effectiveness of the redesign and ensure that the interactive editor significantly enhances the learnability and usability. By testing both tools, it was possible to measure whether the methodology successfully guided the development of the HDSL toward meeting its goals of improving usability and accessibility. If the HDSL showed measurable enhancements over the TDSL, it confirmed that the methodology had been successful in achieving its objectives.

The usability test consisted of four steps: collecting demographics, familiarizing participants with Ampersand, assessing the TDSL, and assessing the HDSL.

- **Step 1: Collect Demographics**

Participants were first asked to complete a brief questionnaire to gather demographic information and obtain their consent to use their feedback. The questions aimed to collect data relevant to the study, focusing on their programming experience, type of experience, and familiarity with programming languages. Participants described their overall programming experience (ranging from no experience to expert), elaborated on the nature of their experience (such as work or study), and selected the programming languages they have used from a predefined list, with an option to specify other languages.

- **Step 2: Familiarization with Ampersand**

To ensure participants could provide meaningful input, they were given an introduction to Ampersand, covering the basics of information systems and how Ampersand facilitates their development. This included an overview of information systems and an explanation of Ampersand's textual syntax.

- **Step 3: Assessment of the TDSL**

In the third step, participants assessed the TDSL by modifying and improving a piece of code in the textual editor. They were tasked with adding functionalities to the code. During this exercise, participants could request explanations about the workings of the TDSL as needed. Upon completing the task, they were asked to fill out the Enhanced SUS questionnaire and provide any additional comments.

- **Step 4: Assessment of the HDSL**

The assessment of the HDSL followed a similar process to that of the TDSL. Participants were asked to perform the same task as in the TDSL assessment, along with an additional task specific to the HDSL. Again, they could request explanations if necessary. After completing these tasks, participants filled out the Enhanced SUS questionnaire once more and had the opportunity to offer further feedback.

Result

In Appendix G, the Enhanced SUS is displayed, containing a column that states the categories of each question. Each category was calculated to determine how well both the TDSL and HDSL performed, which were compared between the TDSL and HDSL assessment. The comparison of the SUS scores for the TDSL and HDSL provided insights into the areas where the HDSL improved usability and where further enhancements are needed. It allowed for a comprehensive evaluation of the user experience, highlighting the strengths and weaknesses of the HDSL compared to the original TDSL. The result of the calculated scores of both DSLs can be found in Table 5.1. Six users participated in the usability testing.

TABLE 5.1: Calculated SUS Score by Category of the TDSL and HDSL

Category	**TDSL**	**HDSL**	**Delta**
Efficiency	4.6	5.6	+1.0
Effectiveness	6.0	5.0	-1.0
Satisfaction	6.6	5.5	-1.1
Understandability	6.5	6.7	+0.2
Learnability	6.0	6.7	+0.7
Operability	7.2	3.1	-4.1
Attractiveness	4.9	5.0	+0.1
Usability Compliance	7.1	6.1	-1.0
Total	6.1	5.5	-0.6

In addition to the Enhanced SUS scores, participants provided verbal feedback during the usability testing. Several key themes emerged from these discussions:

- Participants recognized the potential benefits of the graphical syntax introduced in the HDSL. They appreciated the idea of visual representations, which can offer a more intuitive understanding of complex systems. However, they found the current implementation of the graphical syntax to be neither smooth nor intuitive enough for practical use, indicating that further refinement is needed to enhance its usability.
- Multiple participants expressed a preference for the textual syntax, not because they were familiar with the specific TDSL, but because they were more accustomed to working in textual syntaxes in general. While they also had experience with graphical syntaxes, their greater familiarity with coding in textual formats made them more comfortable and efficient in that environment. This highlighted a significant challenge in transitioning users to the new graphical interface.
- Participants noted that the graphical interface often required more steps to accomplish tasks that were quicker to execute in the textual interface. This increased complexity negatively impacted their overall satisfaction and perceived usability of the HDSL. As a result, while the graphical syntax holds promise, it currently falls short in delivering a seamless and efficient user experience compared to the more familiar textual syntax.

5.2.1 Assessment of the test

The usability test produced mixed results when comparing the TDSL and HDSL systems, as shown in Table 5.1. Overall, HDSL received a lower total score (5.5) compared to TDSL (6.1),

suggesting that the new system, while offering some advantages, also posed challenges in terms of operability and user satisfaction.

HDSL performed better than TDSL in Efficiency (+1.0) and Learnability (+0.7). One possible explanation for the higher efficiency score is that the graphical syntax allowed participants to manage complex relationships more easily, as visualizing these elements may have provided clearer insights. Furthermore, participants indicated that once they became familiar with the graphical syntax, they found certain tasks faster and more intuitive, which could explain the improved learnability score.

However, HDSL scored lower in Effectiveness (-1.0) and significantly lower in Operability (-4.1). The larger drop in operability could indicate that while the graphical syntax was beneficial for specific tasks, it introduced additional steps or complexity when compared to the simpler, more direct interactions available in the textual syntax. This might have led participants to struggle with performing certain operations as quickly or efficiently as they could in TDSL.

Participants also rated HDSL slightly higher in Understandability (+0.2), suggesting that, despite some challenges in operation, the graphical elements were not particularly confusing or difficult to understand. This score indicates that the graphical interface may have potential for improving comprehension, especially for more complex tasks, although it did not dramatically outperform TDSL in this regard.

Despite the improvements in efficiency and learnability, HDSL scored lower in Satisfaction (-1.1) and Usability Compliance (-1.0). This suggests that while participants found certain aspects of the HDSL system helpful, they may have experienced frustration with other parts of the interface or the overall experience. The complexity added by the graphical interface might have been a factor contributing to this lower satisfaction score.

Regarding the relative importance of different categories, while all categories contributed equally to the total score, Learnability and Understandability could be considered more significant in this context, especially for users who are new to the DSL environment. As the HDSL system is designed to introduce graphical syntax, higher scores in these categories suggest that it could potentially lower the barrier to entry for less experienced users. However, since operability remains a critical factor in daily use, the low score in this category points to areas for improvement

5.3 Discussion

The evaluation of HDSL methodology has yielded several significant conclusions. The expert interviews focused on the methodology itself, while the usability tests evaluated the implementation of the case study, providing a comprehensive perspective on both the theoretical framework and practical application of the HDSL.

The expert interviews indicated that the steps outlined in the methodology were generally sufficient. However, a critical gap identified by the experts was the lack of a validation phase and/or iterative testing within the methodology. They emphasized the importance of incorporating iterative design and validation cycles to ensure continuous feedback and improvement. Specifically, experts noted that final validation steps were missing, and shorter iterations over the design of the HDSL were necessary to refine the system progressively.

This gap in iterative testing and validation could explain the negative usability scores observed in the usability tests. The final users were not involved early enough in the design process of the HDSL, which might have led to a disconnect between the theoretical benefits and the practical usability of the system. Participants in the usability tests recognized the overarching value of the HDSL, particularly the potential of graphical syntax to enhance understanding and ease of use. However, they found the specific implementation lacking in smoothness and intuitiveness, which negatively impacted their overall user experience.

The usability test results demonstrated that while the HDSL offers potential, it underperformed compared to the TDSL in several key areas, including Effectiveness, Satisfaction, Operability, and Usability Compliance. Participants preferred the familiar textual syntax, which allowed for quicker task completion. The graphical interface, although visually appealing, required more steps for certain operations, introducing complexity that diminished user satisfaction. Notably, Operability scored significantly lower for HDSL, reflecting the increased complexity and number of steps required in the graphical syntax.

At the same time, HDSL scored higher in Efficiency and Learnability, suggesting that once users adapted to the graphical interface, it became easier to navigate and allowed for faster execution of specific tasks, particularly those involving complex relationships or visual data management. This indicates that HDSL has long-term potential, especially for novice users or those unfamiliar with textual syntax. However, the lower Operability and Satisfaction scores indicate that the graphical interface requires further refinement to make it more intuitive and efficient for everyday use.

In conclusion, the evaluation revealed that while the HDSL methodology holds promise, there is a need for significant improvements, particularly in the design and validation phases. Introducing iterative testing and continuous feedback loops throughout the methodology would allow for user feedback to be integrated at each stage of development, ensuring the system evolves based on real user experience. By addressing these gaps, the HDSL can be refined to meet usability standards more effectively, ultimately improving its accessibility and effectiveness for a wider range of users.

Chapter 6

Conclusion

This chapter aims to summarise the research findings and provide recommendations for future work. The chapter outlines the key conclusions drawn from the methodology and prototype evaluation, highlighting areas for improvement and potential enhancements. The concrete result of this chapter is a set of actionable recommendations to refine the HDSL development process and ensure its practical applicability and effectiveness in real-world scenarios.

6.1 Conclusion

This thesis presents a methodology for designing an HDSL for an existing TDLS, with the goal of improving accessibility and usability for non-technical users while maintaining the functional integrity of the original language. The structured approach suggested in the thesis provides clear guidance on each phase of development, from the initial decision-making to the final implementation, ensuring that both textual and graphical elements are aligned and optimized for usability.

One key finding from the development of the methodology is the need to balance the graphical and textual syntax. While graphical elements are valuable for improving usability and comprehension, they can also introduce complexity, particularly for users already familiar with textual syntax. The usability tests demonstrated that although the HDSL improved efficiency and learnability, it negatively impacted operability. This suggests that the methodology should provide clearer guidance for which elements graphical representations should be defined to ensure that they enhance the user experience without adding unnecessary steps or confusion.

A key contribution of the methodology is its structured approach to HDSL development, offering a blueprint of clear phases —from decision-making to design and implementation— each guided by specific activities and outcomes. However, the evaluation highlighted the need to incorporate iterative testing and validation phases within this structured framework. Continuous feedback loops allow the methodology to be more responsive and adaptive, ensuring that both graphical and textual components can be refined based on user input. Without regular validation phases, there is a risk that the final product may not align with user expectations. By integrating iterative design cycles, the methodology shifts from a linear process to one that evolves with each new iteration, making it more robust and user-centred. This was demonstrated in the Ampersand case study, where the systematic, step-by-step process successfully guided the development of an HDSL for an existing TDLS, which it also revealed areas for further refinement, particularly in the operability of the graphical components.

Research Questions

The primary research goals set forth in this thesis have been systematically evaluated to determine their successful fulfilment and the contributions made toward the development of an HDSL from a TDSL.

1. *Develop a Comprehensive Methodology*: formulate a systematic approach for developing an HDSL from a TDSLs. This methodology addresses the preservation of functional integrity while enhancing usability and accessibility.

The design and development stages focused on creating a structured methodology, of which the successful application was demonstrated by means of a use case. This showed that it was effective in guiding the development process and enabled the creation of a functional HDSL prototype. However, while the methodology worked as intended, the evaluation phase revealed that there were several improvements opportunities. Specifically, incorporating iterative testing and validation cycles is necessary to address usability issues and ensure continuous improvement.

2. *Implement a Prototype:* apply the developed methodology to create an HDSL from an existing TDSL. This prototype has been assessed by using the validation method.

A prototype was realized during the case study, where the methodology was practically applied to develop a functional prototype using the Ampersand platform. This prototype integrated both textual and graphical components, demonstrating the feasibility and effectiveness of the methodology. The usability tests conducted as part of the evaluation phase revealed that while the HDSL holds promise, it currently underperforms compared to the TDSL in several areas.

In summary, the research goals of developing a comprehensive methodology, designing a validation framework, and implementing a prototype have been successfully addressed. However, the evaluation highlighted areas for improvement, particularly in incorporating iterative testing and validation to enhance the methodology and the resulting HDSL's usability and effectiveness.

6.2 Recommendations

The evaluation indicates the need for improvements in the methodology by incorporating iterative testing throughout the development and/or a dedicated validation phase. Adding a test phase to the methodology is a potential solution for ensuring systematic testing and thus feedback is implemented in the development of the HDSL. This phase must be constructed in such a way that allows for shorter development and design cycles, adopting a more agile approach instead of the current waterfall model. By integrating continuous feedback loops and iterative refinements, the HDSL can be progressively improved to better meet usability standards, ultimately enhancing its accessibility and effectiveness for its users.

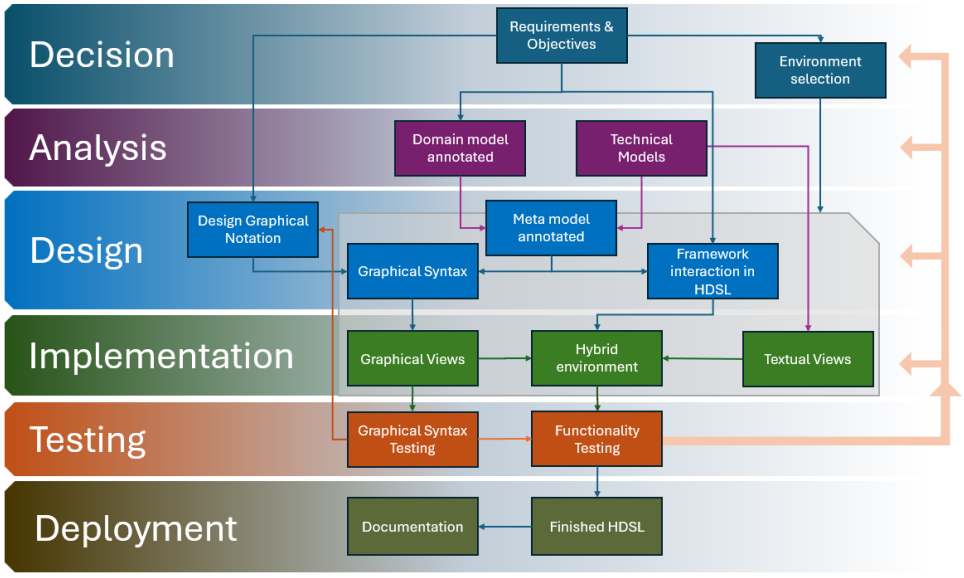

FIGURE 6.1: Extended methodology containing a phase dedicated to testing

The proposed test phase as depicted in Figure 6.1 should be divided into two main categories: *Functionality Testing* and *Graphical Syntax Testing*. Functionality Testing involves engaging stakeholders to gather feedback on the HDSL performance and usability, ensuring the integration of textual and graphical elements functions seamlessly. Graphical Syntax Testing, on the other hand, focuses on refining the graphical components based on user feedback to ensure they are intuitive and effectively communicate complex information. It is crucial that the graphical notation is tested separately to address any specific usability issues and ensure it enhances user understanding.

Incorporating these testing phases will ensure that the HDSL meets both functional and usability standards. The iterative process of testing and refinement will make the methodology more circular and agile, as insights gained during the test phase will feed back into the other phases.

- *Decision Phase:* Feedback from the test phase can inform future decisions regarding the scope and objectives of the HDSL. If certain functionalities are not meeting user needs, the decision phase can be revisited to adjust goals and priorities.
- *Analysis Phase:* Issues identified during testing may highlight gaps or errors in the initial domain analysis. Revisiting this phase can ensure a more accurate and comprehensive understanding of user requirements and domain-specific needs.
- *Design Phase:* Feedback from the graphical and functionality tests can lead to modifications in the design phase. This ensures that the design of the HDSL evolves based on practical user experiences and identified usability issues.
- *Implementation Phase:* Insights gained from testing can lead to minor refinements in the implementation phase. This might involve code adjustments, integration improvements, or enhancements in the development environment.

By validating through continuous feedback and refinement, the HDSL can be tailored to better align with user needs and expectations. This iterative process will enhance the DSL's accessibility and effectiveness, making it a more robust tool for a broader range of users.

6.3 Limitations

While this thesis provides a comprehensive methodology, several limitations were identified throughout the process that affect the generalizability and implementation of the methodology.

- Scope of the Case Study and Limited Testing

One of the main limitations of this thesis is the narrow scope of the case study. The Ampersand platform was the sole environment used to demonstrate the feasibility of the proposed methodology. While this case study offers valuable insights, it limits the generalizability of the results. The applicability of the methodology to other domains or DSL environments remains untested. This limitation is particularly important when considering the potential diversity of DSLs in real-world scenarios, where the nature of the domain and user requirements may differ significantly from those explored in this study. Additionally, while the prototype was developed and tested, the user group involved in the usability testing was small, which may limit the robustness of the findings related to the effectiveness and usability of the HDSL. Future work should involve multiple case studies and a broader range of participants to validate the methodology's adaptability and effectiveness across different contexts.

- Lack of a Comprehensive Deployment Phase

The research focuses on the design, development, and initial validation of the HDSL but does not explore the deployment phase in detail. The deployment of HDSLs in real-world environments involves various factors such as scalability, integration with existing systems, and user training—all of which were beyond the scope of this thesis. Without examining these aspects, it is difficult to determine how easily the HDSL can be adopted in practical applications, especially in complex organizational settings. The absence of detailed deployment strategies may limit the methodology's applicability when transitioning from prototype development to large-scale implementation.

- Limited Integration of Syntax Design Principles

Although principles from graphical syntax design, such as those provided by Moody [19], were referenced in the methodology, their integration could have been explored in more depth. This limited exploration may affect the cognitive effectiveness of the graphical syntax, particularly in complex or technical use cases. A deeper application of such design principles could further optimize the graphical components for clarity and usability, ensuring they meet the cognitive needs of users.

- Usability Issues with Graphical Syntax

The usability testing results revealed challenges with the graphical syntax, specifically in terms of operability. While graphical elements improved efficiency and learnability, participants found the graphical interface cumbersome for certain tasks compared to the textual syntax. This suggests that the graphical syntax design requires further refinement to ensure smoother interaction, especially for users accustomed to textual programming. These usability issues highlight the need for ongoing improvements to the methodology to ensure that graphical components enhance rather than hinder the user experience.

- Lack of Iterative Testing and Validation

A critical limitation identified by both expert interviews and usability testing is the absence of a dedicated validation phase within the methodology. Without iterative testing and validation, it is difficult to ensure that the methodology is continuously refined based on real-time user feedback. The current process is more linear and does not provide opportunities for frequent adjustments, which may lead to gaps between the intended design and actual user needs. Incorporating iterative feedback loops into the methodology would improve its responsiveness and adaptability, helping to address usability and functionality issues earlier in the development process.

6.4 Future Research

Future research should thoroughly explore and develop the test phase, ensuring it includes comprehensive Functionality Testing and Graphical Syntax Testing. This phase must be designed to incorporate continuous feedback loops, allowing for iterative refinements based on user interactions and feedback. Researchers should investigate best practices for engaging stakeholders throughout the testing process to gather meaningful insights and ensure the HDSL aligns with user needs.

One area for future exploration is the application of formal ontologies to guide the design of the HDSL's semantics. While this research relied on internal consistency, formal ontologies could offer a more rigorous way of structuring domain knowledge, improving the clarity and semantic alignment of the HDSL. Investigating how formalized ontologies enhance semantic robustness could be valuable for more complex or technical use cases.

Another critical area for future research is the development and investigation of the deployment phase. This phase is essential for ensuring a smooth transition from development to practical use. It should address issues such as scalability, integration with existing systems, and user training. Detailed guidelines for deployment, including troubleshooting and support mechanisms, should be established to facilitate widespread adoption. By thoroughly exploring the deployment phase, researchers can ensure that the HDSL is not only well-developed but also effectively implemented in real-world scenarios.

Increasing the number of test cases is also paramount for the comprehensive validation of the HDSL methodology. While one demonstration is beneficial for an initial iteration of the methodology's design process, multiple case studies across different domains are necessary to test the versatility and robustness of the HDSL methodology. Future studies could focus on applying the methodology to domains beyond Information Systems, such as healthcare, manufacturing, or education. Each domain comes with its own specific requirements and challenges, and testing the HDSL methodology in these diverse contexts will provide insights into its adaptability and highlight any domain-specific adjustments that may be required.

Moreover, expanding research into Domain-Specific Languages (DSLs) for domains other than Information Systems is an important next step. While this research primarily focused on transforming DSLs in the context of Information Systems, many other fields, such as healthcare, cybersecurity, financial services, and artificial intelligence, could benefit from tailored DSLs that integrate graphical and textual syntaxes. Investigating how the HDSL methodology can be applied to DSLs in these fields would provide a broader perspective on its usefulness and potential. For instance, DSLs in healthcare might focus on visualizing patient data flows or treatment protocols, while in cybersecurity, graphical representations of network security structures could help domain experts manage and understand complex systems more effectively. Research into these fields could identify new opportunities for HDSLs to enhance usability and efficiency across a wide range of sectors.

Increasing the number of test cases is also paramount for the comprehensive validation of the HDSL methodology. While one demonstration is sufficient for an initial iteration of the methodology's design process, multiple case studies across different domains are necessary to test its versatility and robustness. Research should focus on applying the methodology to diverse domains beyond Information Systems. Each domain presents unique challenges, and testing the HDSL methodology in these varied contexts will provide insights into its adaptability and highlight any domain-specific adjustments needed.

Finally, expanding the use of DSLs in fields like healthcare, cybersecurity, and artificial intelligence will demonstrate the broader potential of the HDSL methodology. These fields could benefit from tailored DSLs that integrate graphical and textual syntaxes to address specific challenges, such as visualizing complex data flows or managing intricate system relationships. Future research could explore how the HDSL methodology can enhance the usability and efficiency of DSLs in these diverse sectors, identifying new opportunities for improving domain-specific workflows and decision-making processes.

Appendix A

Findings from the Literature

An overview of the software used in the investigated articles.

Table A.1: Software Tools used for DSL Development

Software	**Purpose**	**articles**
Sirius	Used to define and manage graphical representations within DSLs, particularly for enhancing graphical syntax.	[30]
Xtext	Employed for managing textual components of DSLs, providing features like syntax highlighting and error detection.	[30], [28], [36]
Spoofax	Integrated for DSL development to support modular syntax definition, transformations, and cross-language references.	[5]
AToM3	Applied to design and implement frameworks integrating textual and GDSL components.	[28]
GEF (Graphical Editing Framework)	Utilized to develop rich graphical editor applications integrated into Eclipse, supporting DSL modelling.	[33], [36]
EMF (Eclipse Modelling Framework)	Used as a basis for creating modelling frameworks and DSLs, providing infrastructure for model definition and manipulation.	[33], [14], [36]
Web-based editor	implemented for developing and testing the HyLiMo framework, facilitating integration of textual and graphical editing.	[14]

Table A.2: Overview of HDSL Development Phases and Tools

	Decision Phase	Analysis Phase	Design Phase	Implementation Phase	Deployment Phase
[30]	Conducted through problem identification and analysis of existing tools to pinpoint the need for HDSLs that enhance modelling capabilities.	Not detailed explicitly	Focuses on designing a systematic methodology for DSLs using Sirius for graphical elements and Xtext for textual ones.	Implements the HDSL environment using Eclipse Modelling Framework (EMF)-based tools like Sirius and Xtext.	Integrates the HDSL into existing software development environments using Eclipse plugins.
[6]	Conducted through problem identification and analysis of existing implementations	Both languages are carefully analysed on their workings	A concise DSL syntax is designed for both languages, and an architecture that supports migrations is designed	A pipeline for language migration is implemented to connect/migrate the two languages	No specifics on the deployment were provided.
[5]	Analysis of technological environment (Xtext, Sirius Ecore) and creation of requirements. Related work is investigated into	Not a separate phase from the decision phase	By using the Xtext editor embedded in the Sirius editor, a TDSL code can be written. By referencing to Sirius diagram, graphical elements can be integrated.	Using Xtext and Sirius to simultaneously design and implement the HDSL	The research concluded that a hybrid modelling workbench is needed to allow languages to be embedded into models.
[28]	Review existing GDSLs, considering user feedback and practical challenges in the domain model.	Reviews existing GDSLs and assesses needs for textual integration, focusing on domain requirements.	Designs a framework (using AToM3) to integrate textual and GDSL components, emphasizing metamodel mappings.	Implements the DSL framework in AToM3, facilitating the integration of different DSL components.	Using AToM3, parsers were automatically generated, including the setup of semantics.
[33]	Analysis of current graphical and textual model editing tools, identifying gaps in functionality and usability.	Examines the limitations of graphical model editors and the potential integration of sophisticated text editors.	Designs a TEF-based framework to embed textual model editors within graphical editors using EMF and GMF.	Implements embedded textual editors in a graphical modelling framework, integrating textual and graphical editing features.	An editor was developed and integrated with Ecore.
[14]	Utilized expert interviews and user surveys to gather requirements and feedback, identifying the need for hybrid diagramming tools.	Collects requirements via expert interviews and surveys, focusing on domain-specific needs for hybrid diagramming.	Designs a hybrid diagramming framework (HyLiMo) that supports textual and graphical manipulations.	Implements the HyLiMo framework as a web-based editor, integrating textual and graphical editing capabilities.	By implementing the framework in a web-based editor, the deployment was automatically achieved.
[17]	Analysed the integration possibilities of textual views within existing GDSL, based on user and stakeholder feedback.	Analyses the GDSL used at Ericsson, considering integration of textual notation.	Proposes several alternatives for integrating textual views into GDSLs, focusing on transformation processes.	Details the implementation of chosen alternatives using tools like Xtext and graphical modelling frameworks.	The articles investigate possibilities of implementation and do not mention deployment.
[36]	Evaluated the Xtext framework to identify limitations, decided to incorporate GEF for graphical enhancements.	Evaluates Xtext-based DSLs and their limitations in expressing models purely textually.	Designs a graphical GEF-based editor that synchronizes with Xtext-based textual DSL models.	Implements the graphical editor using GEF, integrating it with Xtext to allow graphical manipulation of DSL models.	The HDSL editor is integrated into the Eclipse IDE.

Example objectives and requirements from the papers.

TABLE A.3: Example objectives and requirements from literature

Requirements	**Purpose**	**Article**
Syntax-Aware Editing Features	Integrate features like syntax highlighting, auto-completion, refactoring, and error detection markers to enhance the coding experience	[30], [5]
Cross-Referencing Model Elements	Allow users to navigate between textual expressions and referenced graphical model elements, enhancing navigability and understanding of complex relationships	[30]
Enhanced Usability through GUIs	Support drag-and-drop functionalities, multi-selection, and other interactive features to make the user interface more intuitive	[36]
Modular Design Approach	Enable independent development and testing of different parts of the DSL, facilitating easier maintenance and upgrades	[14]
Dynamic Interaction Integration	Allow users to switch between or use both textual and graphical editing features simultaneously, providing a more flexible and efficient workflow	[36]
Uniform Error Reporting	Implement consistent error reporting across both textual and graphical components of the model, making it easier for users to identify and rectify issues	[30], [5]
Live Synchronization	Establish live synchronization between textual and graphical components to reflect changes instantly, improving interaction patterns and expressiveness	[14]
Unified Model Management	Utilize an abstract syntax graph (ASG) that integrates elements from both the textual and graphical parts of the model, ensuring consistency and coherence in the DSL	[30], [5]
Flexibility and Extensibility	The implementation should allow for easy modification and extension of the DSL	[28]
Backwards compatibility	The syntaxes should be back and forwards compatible. This lowers the boundary for industry to adopt language workbenches for custom DSL implementations	[14]

Appendix B

Systematic Literature Review

To establish the enhancement objectives and requirements for Ampersand, a brief literature review focused on "understanding how an interactive editor can be designed to make formal specifications more accessible" was conducted. This review was essential as the formal specifications need to be converted from textual syntax to a graphical format.

Theoretical Framework

To address the research goal of understanding how an environment can be designed to make formal specifications more accessible, a systematic literature review was conducted. This review aimed to explore and analyse the existing knowledge on formal languages, interactive editors, and their integration. By examining case studies and empirical evidence, the review sought to identify best practices, design principles, and user experience considerations that can inform the development of more accessible interactive editors for formal specifications.

The framework used is based on Wolfswinkel [38], aimed at exploring and analysing the combined knowledge on formal languages and interactive editors in the area of software development. The review follows the PRISMA methodology [24], which outlines a flow diagram to handle the same steps as Wolfswinkel B.1. Scopus was chosen as the primary database due to its extensive coverage of academic works across numerous disciplines and its reputation as a dependable resource for academic research. The search query was designed to find academic literature focused on case studies in the problem domain. This resulted in the identification of a significant number of documents, initially yielding 2526 articles. The search terms used in the query included combinations of keywords related to formal methods, system design, specification

[H]

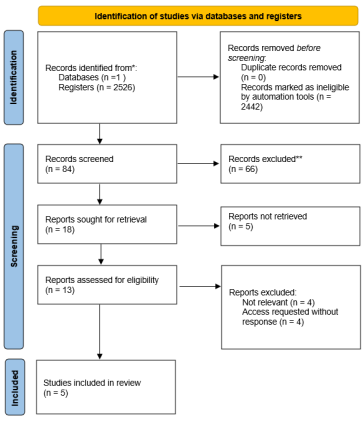

FIGURE B.1: Identification of studies

languages, and rigorous specifications (see Table 1 for the keywords used in the query).

((Formal AND Languages) OR (Formal AND Specifications) OR
(Formal AND System AND Design) OR (Specification AND Languages) OR
(Formal AND Syntax))
AND
((Interactive AND Editor) OR ((Interactive OR Integrated) AND Development AND Environment) OR
(Visual AND Editor) OR (Graphical AND Editor) OR (Interactive AND Tool) OR
(GUI) OR (Interactive AND Interface) OR (Live AND Editor))
AND
((Software AND Development) OR (Software AND Design) OR
(Application AND Development) OR (Systems AND Development) OR
(Software AND Programming))

Articles must be peer-reviewed, written in English, and directly related to the research questions. To narrow down the amount of articles, another keyword filter was implemented on "formal Languages", "Specification Languages" and "formal Specification". This resulted in 382 documents. In order to keep the research relevant, these results were filtered on the past 7 years. This resulted in 84 documents.

The screening involved reviewing the titles and abstracts of the retrieved articles to exclude those that were clearly irrelevant. This process was done by hand and reduced the number of articles to 18.

Following the initial screening, a full-text review of the remaining articles was conducted to ensure they met the inclusion criteria. This in-depth review focused on the relevance of the studies to the integration of formal specifications and interactive editors, the quality of the research methodology, and the significance of the findings. Studies that did not meet these criteria were excluded from the final analysis. This resulted in the inclusion of 5 documents.

Analysis of the Papers

The selected papers were carefully reviewed to answer the question:

- What design principles and features should an interactive editor incorporate to enhance the usability and accessibility of formal specification languages?

These questions guided the analysis, helping to identify the essential design principles and requirements that can improve the usability and accessibility of formal specification languages. Additionally, other relevant requirements mentioned in the texts were documented to ensure a comprehensive understanding of the factors necessary for designing an effective interactive editor. The selected articles can be found in Table B.1

Table B.1: Resulting articles form the literature review

Reference	Title
[20]	A Case Study of a GUI-Aided Approach to Constructing Formal Specifications
[21]	A Comparative Study of a GUI-Aided Formal Specification Construction Approach
[16]	A GUI-Aided Approach to Formal Specification Construction

Reference	Title
[23]	A Formal Modelling Tool for Exploratory Modelling in Software Development
[22]	ViennaTalk and Assertch: Building Lightweight Formal Methods Environments on Pharo 4

B.1 Findings

Based on the findings from the reviewed papers, the following requirements are identified for developing an interactive editor to enhance the accessibility and usability of formal specification languages:

Table B.2: Requirements and Explanations

Requirement	Explanation	Article
Intuitive User Interface	The editor should provide an easy-to-use interface that allows users to visualize and interact with the specification, making it accessible to those without deep technical knowledge	[16] Chapter 3; [23] Chapter 4.2; [22] Chapter 2.2
Dynamic GUI Animation	Incorporate dynamic GUI animations to illustrate potential system behaviours, enabling users to understand and validate specifications effectively	[16] Chapter 3.2
Continuous User Feedback	Implement mechanisms for continuous feedback, allowing users to receive real-time updates on their inputs and the evolving specification. Live editing and visualization can be an extension to this.	[20] Chapter 2; [21] Chapter 3.4; [16] Chapter 3.2; [23] Chapter 4.5; [22] Chapter 2.2.1
Clear Hierarchical Structure	Maintain a clear and logical hierarchical structure to organize functions, data resources, and constraints systematically	[20] Chapter 4; [21] Chapter 3.4; [16] Chapter 4
Iterative Refinement	Support rapid prototyping and iterative refinement to continuously improve the specification based on user feedback	[20] Chapter 3; [21] Chapter 3.1; [23] Chapter 4.3; [22] Chapter 2.2
Cost-Effective Integration	Ensure that the integration of the interactive editor with formal specification languages is cost-effective, reducing development time and minimizing errors	[21] Chapter 6; [23] Chapter 7; [22] Chapter 2.5
Enhanced Communication	Facilitate improved communication between developers and clients through interactive GUI models, ensuring that user requirements are accurately captured and reflected in the formal specifications	[20] Chapter 5; [16] Chapter 6; [23] Chapter 3.2; [22] Chapter 2.4
Code Generation	Incorporating automatic code generation from formal specifications can reduce manual effort and errors, enhancing the efficiency of the development process	[22] Chapter 2.6

Appendix C

Requirements and Objectives Ampersand

The final product of this phase is a set of functional and non-functional requirements, based on the user and stakeholder feedback as well as comprehensive literature review findings. Each requirement is assessed based on the value towards the main business goals using the MoSCoW analysis technique (Brennan, 2009). Additionally, an explanation of its significance and the sources that justify its inclusion is provided. This structured presentation aids in ensuring a holistic view of what is needed for the successful development and implementation of the HDSL.

- *Must (M)* – requirements that must be satisfied in the final solution for the solution to be considered a success.
- *Should (S)*– high-priority items that should be included in the solution if it is possible. This often a critical requirement which can be satisfied in other ways if strictly necessary.
- *Could (C)* – the requirement, which is considered desirable but not necessary. This will be included if time and resources permit
- *Won't or Would (W)* – the requirement that will not be implemented in a given release, but may be considered for the future. The fulfilment of such requirements does not directly affect the solution of the priority problems identified in the analysis, and the value of implementing them more likely.

The source in the tables refer to where the requirements originate from. This can be the Technical Examination (TE), the Stakeholder Needs (SN) or one of the various articles in both the systematic literature review of the example objectives from the methodology.

Table C.1: Non-functional requirements

Sig	Requirements	Justification	Source
F1	The system must become less difficult for user to use	This is the main reason why the HDSL is developed	SN
F2	The system must enable a unified compilation process	This ensures synchronization and compilation of changes across formats.	TE
F3	The system must support hybrid-syntax editing	Both the graphical and textual syntaxes should be separately adjustable	SN, TE [36] [29]
F4	The system must ensure data integrity	The changes made in either syntax must be realized without data loss or inconsistency.	SN, TE
F5	The system must provide simultaneous display of views	The user must be able to seamlessly switch or simultaneously display the views	SN, [36] [29]

F6	The system must provide (unified) error reporting	Consistent error reporting across both textual and graphical components of the systems makes it easier for users to identify and rectify issues	SN, [29] [5]
F7	The system must have intuitive user interfaces	Users with less technical knowledge also need to be able to use the interfaces	SN, [23] [22] [36]
F8	The system must enable rapid prototyping and iterative refinement	This continuously improves the program based on user feedback	[20] [21] [23] [22]
F8	The system should have a modular design approach	This enables the independent development and testing of different parts of the DSL, facilitating easier maintenance and upgrades	[14]
F9	The system should allow for extensibility	The HDSL will be modified and extended in the future.	[28]
F10	The system should allow for continuous feedback on the final product	Real-time updates on inputs and the evolving prototype	[20] [21] [16] [23] [22]
F11	The system could integrate syntax highlighting	Additional features to enhance the development experience are useful to help the user developing the applications	[29] [5]
F12	The system could integrate autocompletion		
F13	The system could integrate refactoring		
F14	The system could integrate error detection markers		
F15	The system could allow Live editing		

TABLE C.2: Non-functional requirements

Sig	Requirements	Justification	Source
NF1	The existing compiler must be used to handle inputs from both graphical and textual sources.	This ensures synchronization and compilation of changes across both formats	
NF2	The system must use Graphviz to visualize the graphical syntax	This application is already in use by the environment	SN, TE
NF3	The system must use the existing RAP environment	This leverages existing infrastructure and reduces the need for additional training	SN, TE
NF4	The current error reporting of the system must be extended to cover both syntaxes	Ensures comprehensive error detection and resolution across all input types	TE
NF5	The system should implement the existing syntaxes	Maintains consistency with current practices, facilitating user adoption	SN
NF6	The system should only encompass a part of the Ampersand Language	A working prototype is developed to test the functionalities and limit the scope	SN
NF7	The system must include multiple interfaces for different syntaxes	This provides flexibility for users to choose the interface that best suits their needs, enhancing usability	
NF8	The system could be enhanced by GUI functionalities	Drag-and-drop functionalities, multi-selection, dynamic GUI animations and other interactive features to make the user interface more intuitive	[36] [16]

Appendix D

Analysis of Ampersand

The technical examination of Ampersand's metamodel focuses on the key elements identified during the domain analysis that need to be implemented into the graphical syntax. This analysis is structured around the Concept, Relation, and Rule. Each section will follow a consistent format to ensure a comprehensive understanding of these elements:

- *Element Explanation:* - A detailed description of the element, its purpose, and its role within the Ampersand platform.
- *Concrete Syntax and Relevant Grammar:* - An explanation of the concrete syntax associated with the element, including relevant grammar rules that govern its use and ensure its proper implementation.
- *Metamodel:* - An overview of the parts of the metamodel that are relevant to the element, illustrating how it fits within the broader structure of Ampersand.
- *Semantics:* a table containing the relevant semantics displayed.

Concept

Concepts are the core entities within the domain, representing key data types or objects that hold significant meaning in the business context. Each concept is uniquely defined and contextualized to ensure clarity and relevance across various scenarios.

Concrete Syntax and Grammar

```
11  CONCEPT NameOfConcept "this is the definition of a CONCEPT"
```

FIGURE D.1: Textual Syntax Concept

In the Ampersand Language, a concept is defined within a Pattern or Context by using the "CONCEPT" statement followed by the desired name of the Concept. The definition (or meaning) of the concepts must be placed between quotation marks on the row of the CONCEPT.

- The name of a concept starts with an uppercase.
- A concept should be used for immutable concepts. E.g. use a concept Person to express that a person will always be a person and will not change in, let us say, a table. However, don't use Employee, because termination of an employee's contract causes a person to be an employee no longer. So employees are not immutable. To be an employee is a dynamic property, so model it as a relation.

- The description will be printed in the functional specification, so please check that your definition is a complete sentence.
- Concepts need not be defined. If you use a concept without a definition, Ampersand defines it for you (regardless of whether you defined it or not).

Metamodel

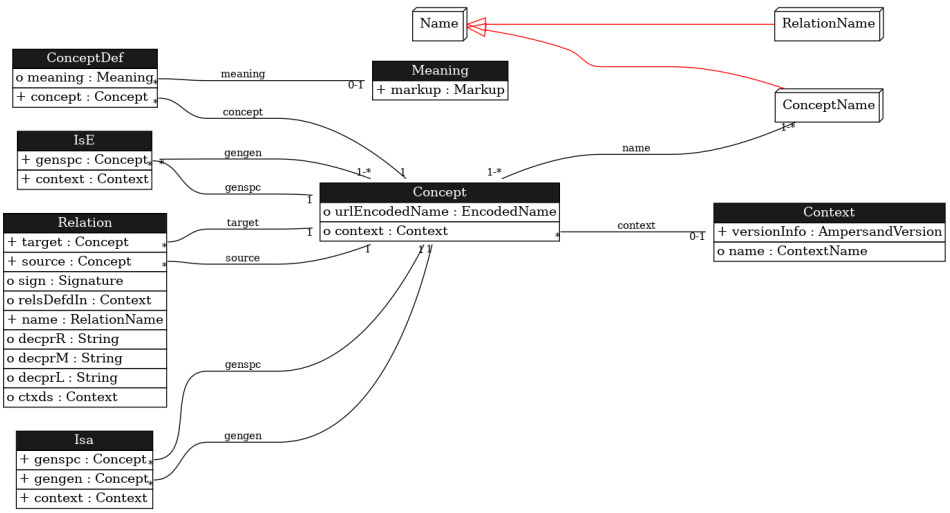

FIGURE D.2: Relevant metamodel Concept

Semantics

TABLE D.1: The semantics related to Concepts

Rule	Description	Implication
A concept must have exactly one Name.	Ensures each concept is identified by a unique, human-readable label.	Facilitates clear identification and referencing within the system, crucial for usability and clarity.
A concept must be associated with exactly one Context or Pattern.	Each concept exists within a specific contextual boundary.	Ensures concepts are interpreted and managed correctly in their respective scenarios, enhancing accuracy and relevance.
Each concept may have one `urlEncodedName`.	Optional attribute for web-friendly referencing of concepts.	Supports systems that interact with web interfaces, ensuring concepts can be safely encoded in URLs.
A concept may participate in multiple generalization relationships.	Allows concepts to be both superclasses and subclasses in hierarchical structures.	Supports complex inheritance and classification schemes, mimicking real-world relationships and hierarchies.
A concept can have multiple meanings (Meaning), each potentially with markup.	Concepts can be described or annotated in various detailed ways.	Enhances the semantic richness and versatility of concept descriptions, allowing for comprehensive documentation.

Relation

Relations specify the associations between concepts, describing how different entities interact or are connected within the domain. These can be directional, showing the flow of data or dependency, and can vary in type, such as one-to-many or many-to-many relationships.

Concrete Syntax and Grammar

```
167  RELATION nameOfRelation [Concept1*Concept2] [PROP]
```

FIGURE D.3: Textual Syntax Relation

In the Ampersand Language, a relation is defined within a Pattern or Context by using the "RELATION" statement followed by the desired name of the relation. The signature of the relation, consisting of a source and a target concept, must be specified within square brackets connected by an asterisk [.. * ..]. The multiplicity (e.g., one-to-many, many-to-many) is defined using properties in a comma separated list between square brackets '[' and ']'. E.g. [UNI,TOT]. The full list of properties can be found in Table D.2.

TABLE D.2: Property rules of the Relations

[..]	Property	Semantics
The following properties can be specified on any relation $r[A * B]$		
UNI	univalent	For any a in A there can be not more than one b in B in the population of r. This implies that every a occurs not more than once (is unique) in the source of r.
TOT	total	For any a in A there must be at least one b in B in the population of r.
INJ	injective	For any b in B there can be not more than one a in A in the population of r. So, every b occurs not more than once in the target of r.
SUR	surjective	For any b in B there must be at least one a in A in the population of r.
There are additional relations that can be specified on endo relations. An endo relation is a relation where the source and target concepts are equal. $r[A * A]$.		
SYM	symmetric	For each (a, b) in r, (b, a) is in r.
ASY	antisymmetric	If (a, b) and (b, a) are both in r, then $a = b$.
TRN	transitive	If (a, b) and (b, c) are both in r, then (a, c) is in r.
RFX	reflexive	For each a in A, the pair (a, a) is in the population of r.
IRF	irreflexive	For each a in A, the pair (a, a) is not in the population of r.
PROP	[SYM, ASY]	Shortcut for the combination of symmetric and antisymmetric.

- The name of a relation starts with a lowercase.

Metamodel

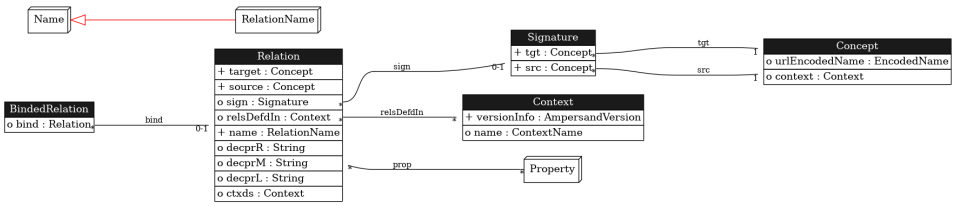

FIGURE D.4: Relevant metamodel Relation

Semantics

TABLE D.3: The semantics related to Relation

Rule	Description	Implication
Each relation must have a signature consisting of a source and target concept.	Establishes a directed connection between two concepts.	Essential for modelling interactions or dependencies between concepts, aligning with real-world or logical relationships.
Each relation can have multiple properties.	Specifies the nature of the relationship (e.g., many-to-many, one-to-many).	Helps define the cardinality and constraints of the relationship, guiding implementation and usage within the system.
A relation must be defined within a specific context (relsDefIn).	Contextualizes the relationship, indicating where it is relevant.	Ensures that the relationship's applicability and relevance are clearly defined, preventing misinterpretation.
Relations must have a descriptive name (RelationName).	Provides an identifiable and meaningful label for the relationship.	Aids in documentation and clarity, enhancing understandability for users and developers alike.
Relations can have descriptions (decrpR, decrpM, decrpL).	Offers detailed explanations or notes about the relationship.	Enhances the semantic richness of the relationship, allowing for comprehensive documentation and understanding.

Rule

Rules enforce constraints or define behaviours within the model, ensuring data integrity and enforcing business logic. They influence how data elements interact and behave under various conditions.

Concrete Syntax and Grammar

```
109  RULE NameOfRule: relation1 |- relation2;relation3~
```

FIGURE D.5: Textual Syntax Rule

In the Ampersand Language, a rule is defined within a Pattern or Context by using the "RULE" statement followed by the desired name of the rule. Term itself consist of a mathematical/algebraical expressions that relate directly to the concepts and their relationships.

- The term of a rule must be correct

Metamodel

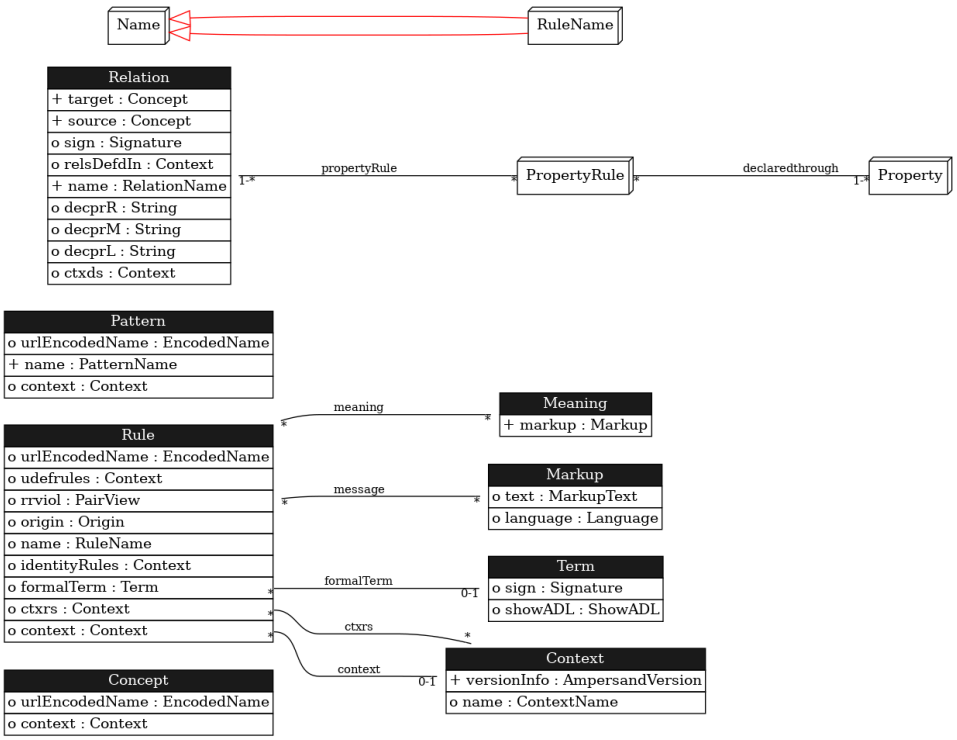

FIGURE D.6: Relevant metamodel Rule

Semantics

TABLE D.4: The semantics related to Rule

Rule	Description	Implication
Each rule must have a unique name (RuleName).	Identifies the rule distinctly within the system.	Facilitates easy referencing and discussion of specific rules, essential for maintenance and compliance monitoring.
Rules must be defined within a context (udefrules).	Specifies the scope or domain where the rule is applicable.	Ensures rules are applied correctly in relevant contexts, enhancing system integrity and functionality.
Rules must have a (formalTerm).	Defines the logical condition or computation the rule enforces.	Provides the mechanism for enforcing constraints or behaviours, crucial for maintaining business logic and data integrity.
Rules may include a user-friendly message (message).	Offers an explanation or error message when the rule is triggered.	Enhances user experience by providing clear feedback on rule violations, aiding in correct data entry and operations.
Rules can influence or be linked to specific properties (PropertyRule).	Connects rules to specific attributes or properties of entities.	Directs the impact of rules to particular aspects of the model, specifying how and where rules enforce constraints.

Full data model

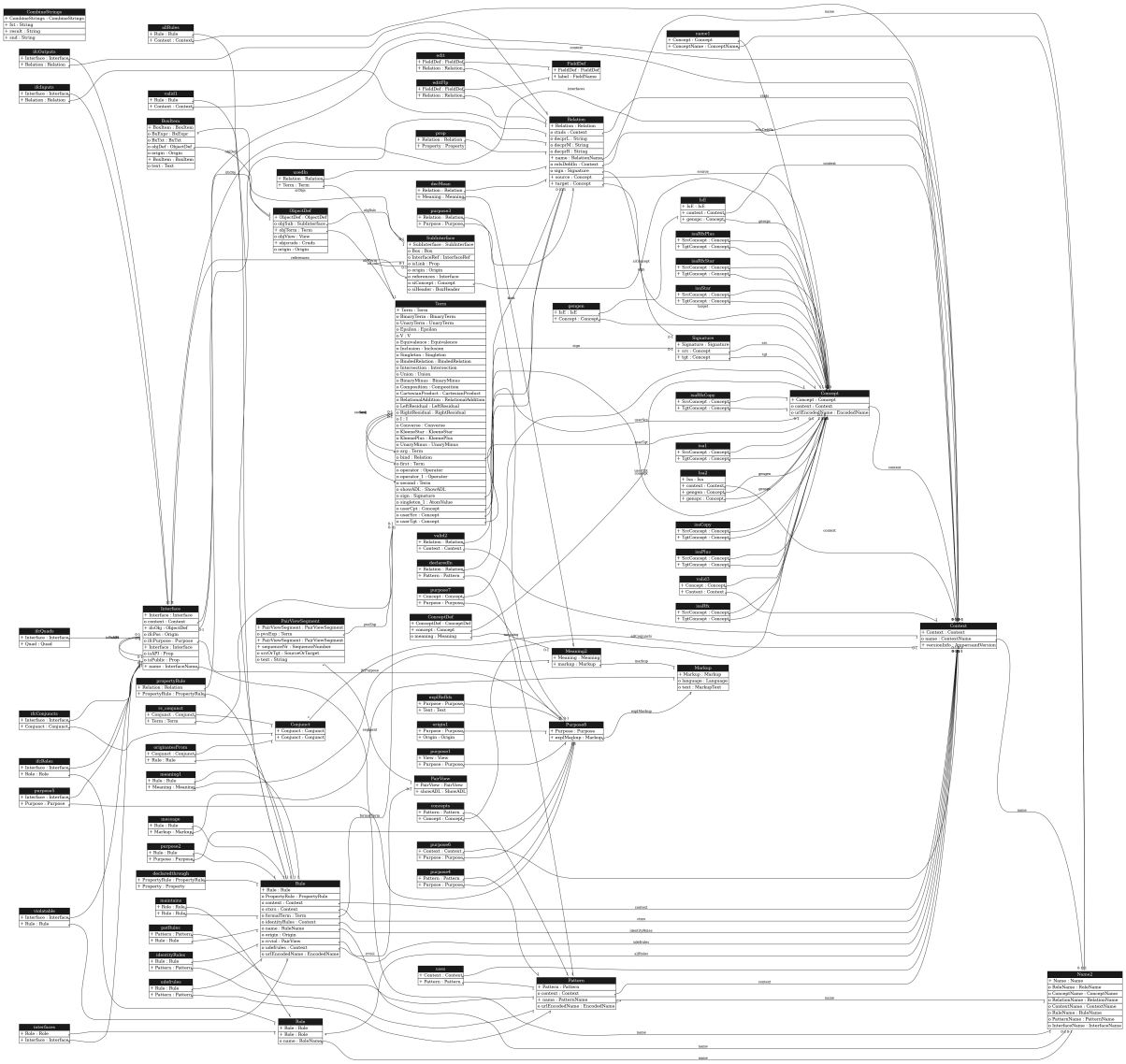

FIGURE D.7: Full model of the Ampersand metamodel

Appendix E

Design of the RAP environment

The revision of the RAP and Ampersand platform is critical to integrating textual and graphical syntaxes into a cohesive development environment. As highlighted in the design and decision phases, the RAP environment needs functionalities to generate textual scripts directly from its database, in addition to the generation of the RAP database from the scripts. This capability would ensure that any modifications made in the database, whether through graphical or textual interfaces, are automatically updated in the script. This integration is vital for maintaining the integrity and accuracy of the system, as it allows for a seamless flow of data and rules across different components of the DSL.

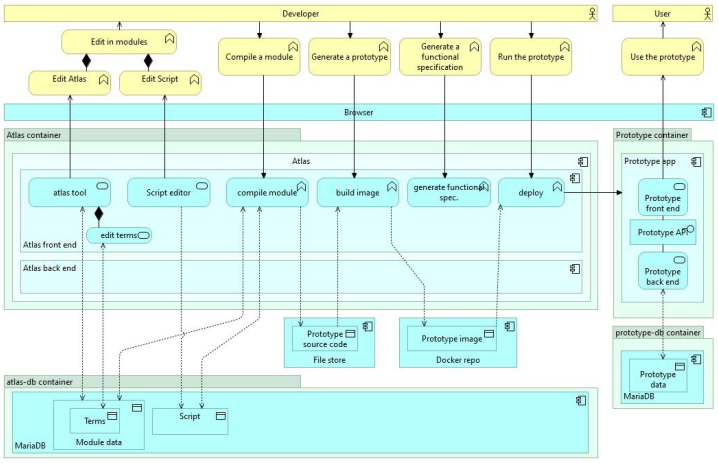

FIGURE E.1: An overview of the workings of the redesigned RAP environment to fit the requirements for the HDSL

The revised RAP architecture, depicted in the figure E.1, illustrates how both the script editor and the ATLAS tool interface with their respective script or database. After modifications, a compilation process via the 'compile module' can be triggered. This setup ensures that both databases—the script and the ATLAS database—are synchronized, preventing loss of edits when switching between the two systems. Each editing tool can independently compile data into the system, where editing in one tool requires recompilation before switching to the other, maintaining consistency across the environments.

To enable this change, certain technical architectural modifications should be made. The RAP environment must allow the future interfaces to alter the underlying database instead of just displaying the data from the database This capability is crucial for enabling ATLAS to function not just as a visualization tool but also as an active editing tool within the development environment. Additionally, an interface must be developed to send the data to the Ampersand

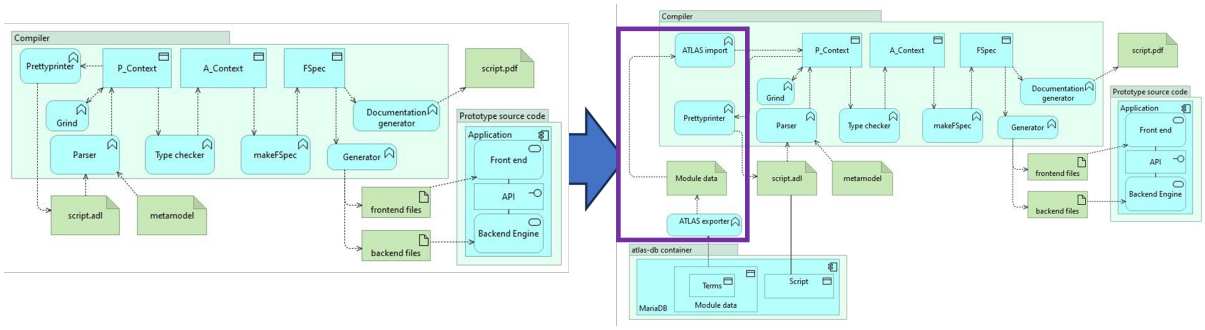

FIGURE E.2: The change in architecture in the development of the Ampersand platform

platform. Next to that, the Ampersand platform must have an interface to receive the data from RAP. This data must in turn be printed upon an .ADL script. To integrate ATLAS with Ampersand effectively, a new pathway from ATLAS to the Ampersand platform is necessary. This involves several components:

- *ATLAS Exporter:* This component is responsible for extracting module data from ATLAS and storing it in a temporary file (the module data file). This file acts as an intermediary storage that holds the data before it is processed by the compiler in the Ampersand platform. For this file, the .json format is chosen.
- *ATLAS Import:* A new parser that reads from the module data file and integrates this data into the Ampersand platform. This allows the graphical data manipulated within ATLAS to be translated into textual script within the Ampersand platform.
- *PrettyPrinter:* A printer which prints the data generated by the compiler in the Ampersand platform to the script editor.

After redesigning the architecture of the RAP environment, the architecture of the functioning of the ATLAS tool itself could be addressed. It was decided to use multiple views to display the elements, resulting in the design shown in Figure E.3.

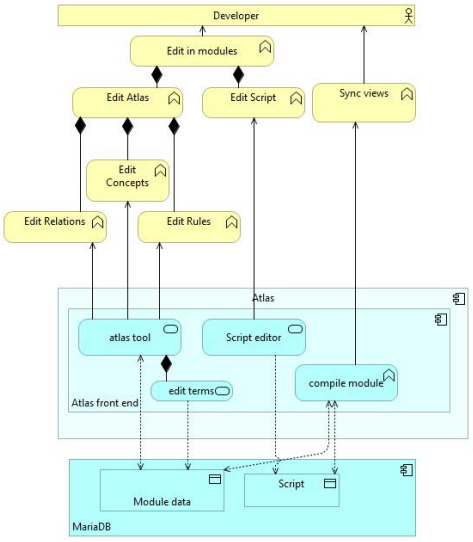

FIGURE E.3: Enter Caption

Appendix F

Development of the RAP environment

The initial step in implementing the design involved developing the environment by creating three transformation functionalities. Although the PrettyPrinter already existed, it required enhancements to meet the new requirements. Additionally, an exporter needed to be developed within RAP to facilitate data exportation.

Atlas Exporter

To export information from ATLAS via a .json file, modifications to the original RAP system were necessary. A new .adl interface was created specifically to retrieve data from the database. This interface is invoked through an API call, structured as follows:

`api/v1/resource/ScriptVersion/[SCRIPTVERSION]/atlas_32_population?"`

This API call enables the extraction of data from the RAP environment, transforming it into a .json format for external use.

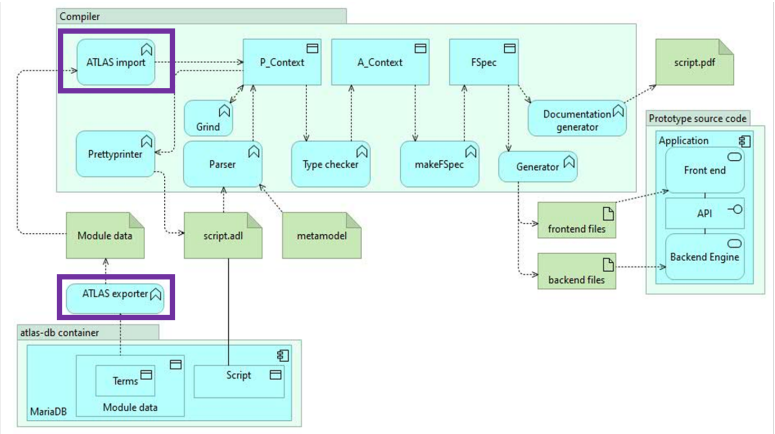

FIGURE F.1: The change in the development of the Ampersand platform

Atlas Exporter

The ATLAS import function involves a Haskell function designed to read the instances from the .json input file using the parseJSON functions. These instances are then parsed into the P_structure of Ampersand through the Build function, after which the data enters the Ampersand platform.

Routing

To facilitate the functionality of these API calls, modifications were made to the ExecEngine of RAP. This involved implementing additional functionalities that allow API calls to store data in a .json file within the database, resulting in the creation of the ATLAS_DATA.json file. This file is structured similarly to the aforementioned interface. Additionally, a command was created in the Ampersand platform to process the .json file using the AtlasImport function. These steps are orchestrated within the ExecEngine of RAP by adding new functions to handle these operations.

Semantics and Grammar Rules

In the RAP environment, the semantic and grammar rules described in the analysis phase were implemented. Since RAP is essentially an Ampersand-generated application, these rules had to be implemented using the Ampersand Language. This was achieved by creating 'rules' that are either executed or enforced by the ExecEngine or the front-end system, referred to as 'User'. Rules executed by the ExecEngine are authoritative and will trigger an error if violated. If a rule enforced by 'User' is violated, it will only generate a notification. Additionally, rules can be defined to trigger specific actions upon violation. For example, the rule depicted in Figure F.2 ensures that any CONCEPT without a defined name receives a temporary name, ensuring compliance with semantic rules without reducing user convenience.

```
38  -- Each (new) CONCEPT is given a Temporary ConceptName when empty
39  -- ConceptName
40  ROLE ExecEngine MAINTAINS conceptTempFillName
41  RULE conceptTempFillName : I[Concept] |- name[Concept*ConceptName];V
42  MESSAGE "Type of CONCEPT was not defined, automatically set to OBJECT"
43  VIOLATION (TXT "{EX} InsPair;ConceptName;Concept;", SRC I, TXT ";ConceptName;TEMPNAME")
```

FIGURE F.2: Example of a rule in Ampersand Language

Appendix G

Usability Testing

The Enhanced System Usability Scale is an advanced version of the traditional SUS [3], developed by Thamilararasan et al. [35] to provide a more comprehensive evaluation of system usability. It adds nine additional questions to the original ten, resulting in a total of 19 questions.

Table G.1: Enhanced SUS Questionnaire

No	Questions	Category
1	I think that I would like to use this system frequently	Satisfaction, Understandability, Learnability
2	I found the system unnecessarily complex	Operability
3	I thought the system was easy to use	Satisfaction, Understandability
4	I think that I would need the support of a technical person to be able to use this system	Operability
5	I found the various functions in this system were well integrated	Understandability, Usability Compliance
6	I thought there was too much inconsistency in this system	Operability
7	I would imagine that most people would learn to use this system very quickly	Satisfaction, Learnability
8	I found the system very cumbersome to use	Operability
9	I felt very confident using the system	Satisfaction
10	I needed to learn a lot of things before I could get going with this system	Satisfaction
11	Tasks can be performed in a straightforward manner using this software	Effectiveness
12	I'm unable to complete my work effectively using this system	Effectiveness
13	I found the interface design of the system follows the usability standards	Usability Compliance
14	I found this system does not fulfil the usability standards	Attractiveness, Usability Compliance
15	I can use it successfully every time	Effectiveness
16	I found this system's colour and graphical design is not attractive enough	Attractiveness
17	I found this system's user interface is very user-friendly	Attractiveness
18	This system responds too slowly to inputs	Efficiency
19	This system helps me to do my job efficiently	Efficiency

Calculating the Score for Each Attribute

To calculate the scores for each usability category, the following steps were undertaken:

1. **Collect Responses:** Each of the 19 questions in the Enhanced SUS has five response options ranging from "Strongly agree" to "Strongly disagree," scored from 1 to 5.
2. **Convert Scores:** For positively worded questions, subtract 1 from the response value to get the adjusted score (0 to 4). For negatively worded questions, subtract the response value from 5 to get the adjusted score (0 to 4).
3. **Sum the Scores:** Add up the adjusted scores for all questions to get a total score.
4. **Multiply by 2.5:** Multiply the total score by 2.5 to convert it to a scale of 0 to 100. This gives the overall usability score.
5. **Calculate Attribute Scores:** To calculate scores for specific attributes, sum the adjusted scores for the questions corresponding to each attribute and then multiply by 2.5. The mapping of questions to attributes is as follows:
    - **Efficiency:** UQ18, UQ19
    - **Effectiveness:** UQ11, UQ12, UQ15
    - **Satisfaction:** UQ1, UQ3, UQ7, UQ9, UQ10
    - **Understandability:** UQ1, UQ3, UQ5
    - **Learnability:** UQ1, UQ7
    - **Operability:** UQ2, UQ4, UQ6, UQ8
    - **Attractiveness:** UQ14, UQ16, UQ17
    - **Usability Compliance:** UQ5, UQ13, UQ14
6. **Interpret Scores:** Higher scores indicate better usability for each attribute. The overall score and the attribute-specific scores can help identify strengths and areas for improvement in the system.

G.1 Participants demographics

For this usability test, the primary focus was on recruiting newer programmers who already have some coding experience. The aim was to ensure that participants had a technical or mathematical background, enabling them to grasp the concepts behind Ampersand effectively. This demographic was chosen to evaluate how well the HDSL would be understood and utilized by individuals who are not experts but have a foundational understanding of programming and technical systems.

These questions helped to identify participants' levels of programming expertise, the nature of their experience, and their familiarity with different programming languages. The gathered data provided a comprehensive understanding of the participants' backgrounds, which is crucial for interpreting the results of the usability testing. An overview of demographics can be seen in Table G.2.

Table G.2: Programming Experience of the Participants

Level	Elaborate on the type of experience	Languages
Competent	Ik heb econometrie gestudeerd, hier heb ik in Java, R en python data analytics en machine learning geleerd. Bij m'n studie Technische Geneeskunde heb ik in Matlab leren beeld bewerken. Nu werk ik in het AI-team van PwC. Ik zit in een intern development team waarin we bouwen aan een AI tool om ESG wetgeving automatisch te checken voor bedrijven.	Python;Java;R;SQL;Matlab
Competent	Work and study	JavaScript;C;SQL
Competent	Study	Python;R
Proficient	Within my studies, we have had multiple projects to utilise modern techniques, such as machine learning and artificial intelligence, to find solutions to difficult problems. Using a variety of neural networks for the use of financial forecasting has been the biggest example of this. Additionally, this also returns in my day-to-day work in my job	Python;R;SQL
Competent	I use it for study regularly	Python;VBA
Novice	Study	Python

G.2 results

The data collected from the Enhanced SUS) questionnaires can be found in table G.3 to provide a clear picture of the usability performance of both the TDSL and HDSL. "Strongly agree" to "Strongly disagree," scored from 4 to 0.

Table G.3: All data of the tests

No	Type	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	TOT
1	TDSL	2	1	2	3	1	1	1	2	1	1	0	1	1	1	0	0	0	0	0	
	HDSL	3	1	3	0	3	2	3	1	2	0	3	1	3	1	3	1	3	1	3	
2	TDSL	3	3	1	3	3	1	2	2	3	3	3	4	3	1	1	0	2	1	3	
	HDSL	3	3	1	2	3	1	3	1	2	1	3	1	2	3	1	1	2	1	3	
3	TDSL	1	4	4	3	4	4	4	4	4	4	4	4	4	3	4	4	4	2	2	
	HDSL	1	2	2	3	2	1	3	1	2	2	3	1	3	1	2	2	3	1	2	
4	TDSL	3	4	4	3	3	4	4	4	4	3	3	3	3	4	2	0	3	2	3	
	HDSL	1	1	3	1	3	1	3	2	1	1	1	1	3	1	1	3	3	3	3	
5	TDSL	3	3	3	3	4	4	4	3	3	3	4	4	3	3	3	2	3	2	3	
	HDSL	3	0	3	1	4	1	3	1	3	0	4	1	3	1	3	2	3	2	3	
6	TDSL	1	3	2	1	3	3	1	3	2	3	1	2	3	3	1	3	3	1	1	
	HDSL	3	1	3	2	4	1	3	0	4	1	3	2	2	2	1	3	3	3	2	

Bibliography

[1] Leif Andersen, Michael Ballantyne, and Matthias Felleisen. Adding interactive visual syntax to textual code. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–28, 2020.

[2] Kevin Brennan et al. *A guide to the Business Analysis Body of Knowledger*. Iiba, 2009.

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

[4] Margaret M Burnett and David W McIntyre. Visual programming. *COmputer-Los Alamitos-*, 28:14–14, 1995.

[5] Justin Cooper and Dimitris Kolovos. Engineering hybrid graphical-textual languages with sirius and xtext: Requirements and challenges. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 322–325. IEEE, 2019.

[6] Jasper Denkers, Louis van Gool, and Eelco Visser. Migrating custom dsl implementations to a language workbench (tool demo). In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering*, pages 205–209, 2018.

[7] DHI Group (Firm). Dice tech job report: the fastest growing hubs, roles and skills. 2020.

[8] William Frakes, Ruben Prieto-; Diaz, and Christopher Fox. Dare: Domain analysis and reuse environment. *Annals of software engineering*, 5(1):125–141, 1998.

[9] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Monticore: a framework for the development of textual domain specific languages. In *Companion of the 30th international conference on Software engineering*, pages 925–926, 2008.

[10] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Derivation and refinement of textual syntax for models. In *Model Driven Architecture-Foundations and Applications: 5th European Conference, ECMDA-FA 2009, Enschede, The Netherlands, June 23-26, 2009. Proceedings 5*, pages 114–129. Springer, 2009.

[11] James A Hess, E William, and A Novak. Feature-oriented domain analysis (foda) feasibility study kyo c. kang, sholom g. cohen. *no. September*, 1990, 2020.

[12] Doortje Hoogsteen and Hans Borgman. Empower the workforce, empower the company? citizen development adoption. 2022.

[13] Stef Joosten. Relation algebra as programming language using the ampersand compiler. *Journal of Logical and Algebraic Methods in Programming*, 100:113–129, 2018.

[14] Niklas Krieger. Hylimo: a textual dsl and hybrid editor for efficient modular diagramming. In *SE 2024-Companion*, pages 185–186. Gesellschaft für Informatik eV, 2024.

[15] Mohammad Amin Kuhail, Shahbano Farooq, Rawad Hammad, and Mohammed Bahja. Characterizing visual programming approaches for end-user developers: A systematic review. *IEEE Access*, 9:14181–14202, 2021.

[16] Shaoying Liu. A gui-aided approach to formal specification construction. In *International Workshop on Structured Object-Oriented Formal Language and Method*, pages 44–56. Springer, 2015.

[17] Salome Maro. A dsl supporting textual and graphical views. 2015. doi:http://hdl.handle.net/2077/40135.

[18] 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.

[19] Daniel Moody. The "physics" of notations: toward a scientific basis for constructing visual notations in software engineering. *IEEE Transactions on software engineering*, 35(6):756–779, 2009.

[20] Fumiko Nagoya and Shaoying Liu. A case study of a gui-aided approach to constructing formal specifications. In *Structured Object-Oriented Formal Language and Method: 6th International Workshop, SOFL+ MSVL 2016, Tokyo, Japan, November 15, 2016, Revised Selected Papers 6*, pages 74–84. Springer, 2017.

[21] Fumiko Nagoya and Shaoying Liu. A comparative study of a gui-aided formal specification construction approach. In *Computational Science and Its Applications–ICCSA 2017: 17th International Conference, Trieste, Italy, July 3-6, 2017, Proceedings, Part I 17*, pages 273–283. Springer, 2017.

[22] Tomohiro Oda, Keijiro Araki, and Peter Gorm Larsen. Viennatalk and assertch: building lightweight formal methods environments on pharo 4. In *Proceedings of the 11th edition of the International Workshop on Smalltalk Technologies*, pages 1–7, 2016.

[23] Tomohiro Oda, Keijiro Araki, and Peter Gorm Larsen. A formal modeling tool for exploratory modeling in software development. *IEICE TRANSACTIONS on Information and Systems*, 100(6):1210–1217, 2017.

[24] University of North Carolina at Chapel Hill Libraries. Prisma: Transparent reporting of systematic reviews and meta-analyses. https://guides.lib.unc.edu/prisma, 2024. Accessed: 2024-6-20.

[25] Open University of the Netherlands and Ordina. Ampersandtarski: Building information systems. https://ampersandtarski.github.io/, 2024. Accessed: 2024-07-28.

[26] Richard F Paige, Dimitrios S Kolovos, and Fiona AC Polack. A tutorial on metamodelling for grammar researchers. *Science of Computer Programming*, 96:396–416, 2014.

[27] Ken Peffers, Tuure Tuunanen, Marcus A Rothenberger, and Samir Chatterjee. A design science research methodology for information systems research. *Journal of management information systems*, 24(3):45–77, 2007. doi:10.2753/MIS0742-1222240302.

[28] Francisco Pérez Andrés, Juan De Lara, and Esther Guerra. Domain specific languages with graphical and textual views. In *Applications of Graph Transformations with Industrial Relevance: Third International Symposium, AGTIVE 2007, Kassel, Germany, October 10-12, 2007, Revised Selected and Invited Papers 3*, pages 82–97. Springer, 2008. doi:10.1007/978-3-540-89020-1_7.

[29] Ionut Predoaia. Towards systematic engineering of hybrid graphical-textual domain-specific languages. In *2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 153–158. IEEE, 2023.

[30] Ionut Predoaia, Dimitris Kolovos, Matthias Lenk, and Antonio García-Domínguez. Streamlining the development of hybrid graphical-textual model editors for domain-specific languages. *Journal of Object Technology*, 22(2):2:1–14, July 2023. The 19th European Conference on Modelling Foundations and Applications (ECMFA 2023). URL: http://www.jot.fm/contents/issue_2023_02/article8.html, doi:10.5381/jot.2023.22.2.a8.

[31] Chris Rupp. Requirements templates: The blueprint of your requirements. https://www.sophist.de/fileadmin/user_upload/Bilder_zu_Seiten/Publikationen/RE6/Webinhalte_Buchteil_3/Requirements_Templates_-_The_Blue_Print_of_your_Requirements_Rupp.pdf, n.d. Accessed: 2024-09-14.

[32] Apurvanand Sahay, Arsene Indamutsa, Davide Di Ruscio, and Alfonso Pierantonio. Supporting the understanding and comparison of low-code development platforms. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 171–178, 2020. doi:10.1109/SEAA51224.2020.00036.

[33] Markus Scheidgen. Textual modelling embedded into graphical modelling. In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 153–168. Springer, 2008.

[34] Richard N Taylor, Will Tracz, and Lou Coglianese. Software development using domain-specific software architectures: Cdrl a011—a curriculum module in the sei style. *ACM SIGSOFT Software Engineering Notes*, 20(5):27–38, 1995.

[35] Yarshini Thamilarasan, Raja Rina Raja Ikram, Mashanum Osman, Lizawati Salahuddin, Wan Yaakob Wan Bujeri, and Kasturi Kanchymalay. Enhanced system usability scale using the software quality standard approach. *Engineering, Technology & Applied Science Research*, 13(5):11779–11784, 2023.

[36] Marcel Toussaint and Thomas Baar. Enriching textual xtext-dsls with a graphical gef-based editor. In *Perspectives of System Informatics: 11th International Andrei P. Ershov Informatics Conference, PSI 2017, Moscow, Russia, June 27-29, 2017, Revised Selected Papers 11*, pages 394–401. Springer, 2018.

[37] Simon van Roozendaal. Methodology for the development of domain specific languages with both graphical and textual elements. https://docs.google.com/presentation/d/111nQGnt7z6Agtp1l8oWyyqLS-GIKrV3z/edit#slide=id.p5, 2024.

[38] Joost F Wolfswinkel, Elfi Furtmueller, and Celeste PM Wilderom. Using grounded theory as a method for rigorously reviewing literature. *European journal of information systems*, 22(1):45–55, 2013.