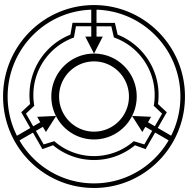
Your Thesis Title Here



Rafael Tanzer

Cyber-Physical-Systems, ..., ...
TU Vienna

Supervisor: Michele Chiaria, Simone , Francesco Pontigia and Ezzio Bartocci

May 2025

This thesis is submitted in partial fulfillment of the requirements for the degree of *Bachelor of Science*.

Abstract

Contents

I Foundations 3

1 Introduction 4
1.1 Motivation and Problem Statement . . . 4
1.2 Objectives and Scope . . . 4

2 Background on Languages & Grammars 5
2.1 Formal Language Theory . . . 5
2.1.1 Backus–Naur Form (BNF) and Variants . . . 5
2.1.2 Grammar Formalisms and Parsing Models . . . 5
2.2 Language Structure and Paradigms . . . 5
2.2.1 Abstract vs. Concrete Syntax . . . 5
2.2.2 Declarative Languages . . . 5
2.3 Domain-Specific Languages (DSLs) . . . 5
2.3.1 Characteristics and Use Cases . . . 5
2.3.2 Declarative DSLs . . . 5

3 Probabilistic Programming & Cyber-Physical Systems 6
3.1 What Is Probabilistic Programming? . . . 6
3.2 Inference and Sampling Methods . . . 6
3.3 Cyber-Physical Systems: Definitions and Examples . . . 6
3.4 Motivation for DSLs in P-P and CPS Contexts . . . 6

4 The *Miniprob* DSL 7
4.1 Domain and Purpose . . . 7
4.2 Syntax Overview . . . 7
4.3 Semantics and Example Models . . . 7

5 Language Servers, Parsers & Type Checking 8
5.1 Language Server Protocol (LSP) Basics . . . 8
5.2 Parser and Syntax Checking . . . 8
5.3 Type Checking Functionality . . . 8
5.4 Overview of Langium . . . 8

II Design & Implementation 9

6 Requirements & Technology Survey 10
6.1 Functional Requirements . . . 10
6.2 Non-functional Requirements . . . 10
6.3 Alternative Technologies . . . 10

6.4 Justification for Langium & VS Code . . . 10

7 Architectural Design 11
7.1 High-Level Architecture Diagram . . . 11
7.2 Module Decomposition . . . 11
7.3 Data Flow and Control Flow . . . 11

8 Implementation Details 12
8.1 Langium Grammar Definition for *Miniprob* . . . 12
8.2 Semantic Checks and Type System . . . 12
8.3 VS Code Extension Points . . . 12

9 Testing & Validation 13
9.1 Unit Tests . . . 13
9.2 Integration Tests . . . 13
9.3 Case Studies & Examples . . . 13

III Evaluation & Discussion 14

10 Performance & Usability 15
10.1 Parsing Speed . . . 15
10.2 Editor Responsiveness . . . 15
10.3 User Feedback . . . 15

11 Discussion 16
11.1 Meeting the Requirements . . . 16

A Full Grammar Specification 18

B Expanded Code Listings 19

C Test Data 20

Part I

Foundations

Introduction

1.1 Motivation and Problem Statement

CPS team - waht do they need it for code growing user base - interconnecticty os , other qualities speaking for the usage of vs code - already in possesion of Haskell Checker but

1.2 Objectives and Scope

The main goal of the thesis was to create a *Language Extension* <title> for *MiniProb*. Language extensions are a big part of VS Code's extension ecosystem and enable the editor to utilize custom language tooling. Such extensions support the user during the implementation process while writing code, by providing language assistance like syntax validation or code completion. While this implementation targets VS Code, the underlying logic and functionality are not limited to it — through the use of the *Language Server Protocol (LSP)*, the developed tooling can be reused across various editors and IDEs that support the protocol. The core functionality of these extensions stems from *parsers* which, against a formal language definition, convert written text into abstracted parts, validating the code in the process. These parts can then be used to extend the capabilities of the tooling to encompass referential validation, type checking, code completion, diagnostics, and other language services that enhance the development experience.

In order for parsers to function accordingly, they require formal language definitions or models. These definitions range from various types of grammars to abstract machines such as automata, which together provide the structural and syntactic rules necessary for correct interpretation and validation of source code. Based on these definitions, parsers are able to systematically identify the hierarchical structure of a program by recognizing patterns, token sequences, and nested constructs, ensuring that the code adheres to the expected form before any further processing or analysis occurs. In this thesis, a generative context-free grammar is used to formally describe *MiniProb*, a domain-specific language (DSL) designed for authoring *POMC* files. Based on this grammar, a parser - serving as the foundation - enables the implementation of a fully functional language extension to aid developers writing POMC files.
Ultimately, the resulting extension, titled <title>, is intended to provide comprehensive language support for MiniProc. This includes syntactic analysis through syntax highlighting and validation, accurate resolution of symbol references and code completion as well as the implementation of type checking mechanisms to ensure semantic correctness during development. //maybe basic code completion for expressions

An appropriate set of regression tests was developed to ensure the continued correctness and stability of the language tooling. These tests include parsing tests, which verify that valid input is correctly recognized and structured according to the grammar; validation tests, which ensure that semantic rules are properly enforced and errors are accurately reported; and linking tests, which check that references between symbols or declarations are correctly resolved across different parts of a program. Together, these test categories help maintain the integrity of the parser and language services as the implementation evolves.

Lastly, this thesis includes an evaluation of the newly implemented parser, focusing on its correctness and performance in practical usage scenarios. The evaluation is based on metrics collected from representative input samples, measuring factors such as parsing speed, memory usage, and error handling. Where relevant, comparisons are also drawn against the existing *Haskell*-based parser for *.pomc* files, offering a point of reference to assess improvements or trade-offs introduced by the new implementation.

Background on Languages & Grammars

2.1 Formal Language Theory

Formal language theory deals with the study of languages – sets of strings constructed from alphabets – and the formal grammars that generate them. In contrast to natural languages, which have evolved over centuries under the influence of diverse cultural, historical, and environmental factors, formal languages do not inherently relate to any perceived constructs of our environments and are generally not intuitively understood. Additionally, formal languages do not share the rich evolutionary progression — a lengthy process of gradual adaptations and refinements spanning generations — of their natural counterparts. Instead, they employ sets of axiomatic *rules* that describe each language individually. The field of formal language theory sprung from linguist Noam Chomsky's attempts during the 1950s to definitively characterize the structure of natural languages using mathematical rules.[Jia+09] ...

nalasndlsandas;daskdk

which describe the language. produciton rules and close inspection of models for context sensitive and regular lang as they are used in this.

2.1.1 Backus–Naur Form (BNF) and Variants

2.1.2 Grammar Formalisms and Parsing Models

2.2 Language Structure and Paradigms

2.2.1 Abstract vs. Concrete Syntax

2.2.2 Declarative Languages

2.3 Domain-Specific Languages (DSLs)

2.3.1 Characteristics and Use Cases

2.3.2 Declarative DSLs

DSL or General (do api prog lang have to be haskell complete? C3, java haskell -»)

Probabilistic Programming & Cyber-Physical Systems

3.1 What Is Probabilistic Programming?

3.2 Inference and Sampling Methods

3.3 Cyber-Physical Systems: Definitions and Examples

3.4 Motivation for DSLs in P-P and CPS Contexts

The *Miniprob* DSL

4.1 Domain and Purpose

4.2 Syntax Overview

4.3 Semantics and Example Models

Language Servers, Parsers & Type Checking

https://code.visualstudio.com/api/language-extensions/overview

5.1 Language Server Protocol (LSP) Basics

5.2 Parser and Syntax Checking

5.3 Type Checking Functionality

5.4 Overview of Langium

Part II

Design & Implementation

Requirements & Technology Survey

6.1 Functional Requirements

6.2 Non-functional Requirements

6.3 Alternative Technologies

6.4 Justification for Langium & VS Code

Architectural Design

7.1 High-Level Architecture Diagram

7.2 Module Decomposition

7.3 Data Flow and Control Flow

Implementation Details

8.1 Langium Grammar Definition for *Miniprob*

8.2 Semantic Checks and Type System

8.3 VS Code Extension Points

Testing & Validation

9.1 Unit Tests

9.2 Integration Tests

9.3 Case Studies & Examples

Part III

Evaluation & Discussion

Performance & Usability

10.1 Parsing Speed

10.2 Editor Responsiveness

10.3 User Feedback

Discussion

11.1 Meeting the Requirements

Bibliography

[Jia+09] Tao Jiang et al. “Formal Grammars and Languages”. In: (Nov. 2009). DOI: 10.1201/9781584888239-c20.

Full Grammar Specification

Expanded Code Listings

Test Data