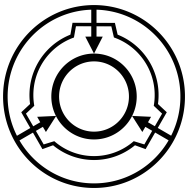
Your Thesis Title Here



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 . . . 6
2.1.2 Grammar Formalisms and Parsing Models . . . 6
2.2 Language Structure and Paradigms . . . 6
2.2.1 Abstract vs. Concrete Syntax . . . 6
2.2.2 Imperative vs. Declarative Languages . . . 7
2.3 Domain-Specific Languages (DSLs) . . . 7
2.3.1 Imperative DSLs . . . 8

3 Cyber-Physical Systems and Formal Analysis 9
3.1 Cyber-Physical Systems: Definition and Examples . . . 9
3.2 Formal Methods and Model Checking . . . 10
3.3 Motivation for PPLs in CPS Contexts . . . 10

4 Probabilistic Programming 12
4.1 What Is Probabilistic Programming? . . . 12
4.2 Inference and Sampling Methods . . . 12
4.3 Symbolic Semantics and Model Checking . . . 13
4.4 Symbolic Semantics and Model Checking for Probabilistic Programs . . . 13

5 The *MiniProb* DSL 15
5.1 Domain and Purpose . . . 15
5.2 Syntax Overview . . . 15
5.3 Semantics and Example Models . . . 15

6 Language Servers, Parsers & Type Checking 16
6.1 Language Server Protocol (LSP) Basics . . . 16
6.2 Parser and Syntax Checking . . . 16
6.3 Type Checking Functionality . . . 16
6.4 Overview of Langium . . . 16

II Design & Implementation 17

7 Requirements & Technology Survey 18
7.1 Functional Requirements . . . 18
7.2 Non-functional Requirements . . . 18
7.3 Alternative Technologies . . . 18
7.4 Justification for Langium & VS Code . . . 18

8 Architectural Design 19
8.1 High-Level Architecture Diagram . . . 19
8.2 Module Decomposition . . . 19
8.3 Data Flow and Control Flow . . . 19

9 Implementation Details 20
9.1 Langium Grammar Definition for *Miniprob* . . . 20
9.2 Semantic Checks and Type System . . . 20
9.3 VS Code Extension Points . . . 20

10 Testing & Validation 21
10.1 Unit Tests . . . 21
10.2 Integration Tests . . . 21
10.3 Case Studies & Examples . . . 21

III Evaluation & Discussion 22

11 Performance & Usability 23
11.1 Parsing Speed . . . 23
11.2 Editor Responsiveness . . . 23
11.3 User Feedback . . . 23

12 Discussion 24
12.1 Meeting the Requirements . . . 24

A Expanded Code Listings 27

B Test Data 28

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* ?NAME? 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 ?NAME?, 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 determine and 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 *production 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] This analytical approach led to development of the *Chomsky hierarchy*, which proved to be a vital theoretical foundation for later discoveries and applications, as it was found that all information(photos, videos, numbers, axioms) can be represented as finite strings.

The *Chomsky Hierarchy* is a hierarchical classification of formal grammars, that labels them to four groups. The grammars are ranked based on the individual *expressive power* of the languages they produce, with each class including the less expressive ones. The whole hierarchy consists of four groups of grammars (and corresponding language classes), that are identified by inspecting the production rules, which get progressively less restrictive.

Type-3 grammars, or *regular* grammars, generate exactly the class of regular languages. Their productions are restricted to the two equivalent styles:

$$right-reg.: \quad A \to aB \quad or \quad A \to a \qquad and \qquad left-reg.: \quad A \to Ba \quad or \quad A \to a,$$

where A, B are nonterminals and a is a terminal. Regular expressions provide an alternative, declarative notation for these languages — each expression can be mechanically transformed into a regular grammar, and vice versa. More broadly, every grammar in the Chomsky hierarchy admits an equivalent acceptor automaton; in the case of Type-3 grammars, this correspondence yields deterministic or nondeterministic finite automata. Leveraging these conversions makes regular grammars invaluable in practice (for example, in lexical analysis, pattern matching, and protocol verification), even though their expressive power is the most limited. Type-2 grammars, or *context-free* grammars (CFGs), produce all context-free languages. Their rules take the general form

$$A \to \alpha$$

where A is a single nonterminal and α is any string of terminals and nonterminals. This additional flexibility captures nested, hierarchical structures—like balanced parentheses or most programming-language syntaxes — that regular grammars cannot. Each CFG is accepted by a corresponding pushdown automaton: the grammar expansions map naturally onto push and pop operations on the PDA's stack, making the grammar-automaton equivalence at this level another cornerstone of formal language theory.

for choosing between replacing or recalibrating components—but measured data often reflect intertwined effects and closed-loop corrections. titcontext-sensitive and *recursively enumerable* languages respectively, and are the most expressive of all with Type-0 grammars placing absolutely no constraints on the production, making them only acceptable by Turing-Machine.

2.1.1 Backus–Naur Form (BNF) and Variants

Even though production rules, alphabet specifications, and automaton definitions suffice to unambiguously define the set of valid strings/sentences in a language, their notation tends to be opaque and cumbersome in practice, prompting interest in forming intuitively comprehensible grammar definitions. The Backus-Naur Form (BNF), created by John Backus with contributions by Peter Naur and released in their Algol-60 report. [Bac+60], was a pivotal introduction furnishing a concise, human-readable syntax for language generation. By expressing each rule as

$$\langle \text{nonterminal} \rangle \ ::= \ expansion_1 \mid expansion_2 \mid \ldots$$

BNF allows both sequencing and alternation of multiple expansions, enabling language designers to articulate complex, nested structures without exposing the underlying automaton states or transition tables. This declarative approach not only supports rigorous specification but also facilitates mechanical parsing, thereby advancing the practice of compiler development and language tooling.

While BNF provides a clear, formal way to describe language syntax, it suffers from verbosity and redundancy — pure BNF grammars often become bloated when encoding common patterns like optionals or repetitions requiring auxiliary nonterminals for a sufficient description and, over time, has been extended and modified for various use-cases spawning a new family of grammar notations. The **Extended Backus-Naur Form** extends BNF by adding more expressive meta-syntax, introducing operators similar to *Regular Expressions* allowing the use of optional or repeatable expressions, commonly indicated by brackets but not limited to '[...]' and '{...}', and the use of comments. Such extensions make grammars more compact and easier to read without changing the fundamental class of languages they describe.[] Multiple distinct instances of EBNF's have been development, all with minor syntactic differences, yet there really is no on-for-all EBNF used in practice, although a standardized ISO/IEC 14977 version exists. [Jinb; Jina]
Within the scope of my thesis, *?NAME?*, Extended Backus–Naur Form (EBNF) is particularly significant, since Langium employs its own custom EBNF dialect to define the grammars underpinning its language-assistance features. 9

2.1.2 Grammar Formalisms and Parsing Models

..programming languages

2.2 Language Structure and Paradigms

2.2.1 Abstract vs. Concrete Syntax

Depednming on subsection Grammar Formalisms, mentino the importance of grammars in connectino with programming langs.??

The terms *abstract* and *concrete* syntax are primarily encountered in the context of programming language design. While the two concepts follow the common relationship of abstract and concrete instances — where one embodies a generalized, meta-level blueprint, while the other encapsulates the concrete, instance-level details — and the term "syntax" broadly applies to all languages, the idea of pinning meta-information onto language constructs, somewhat implies further calculations based on the language itself.

When designing programming languages, the abstract syntax is of particular interest, as its streamlined view of the language is ideally suited for developing validation constraints or type systems and for general interpretation. This is done most commonly by encoding the syntax into a **Abstract Syntax Tree** (**AST**), a tree-like structure containing nodes for each high-level construct (such as expressions, declarations, or statements) connected according to their logical hierarchy, while deliberately omitting lexical details to yield a canonical. [SK95] The **Concrete Syntax Tree** (**CST**) is the specialized counterpart to the AST and consists of a full parse-tree, preserving every terminal and nonterminal token(keywords, delimiters, etc.) and mirrors each grammar production in its node structure, providing the complete syntactic context possibly needed for precise error reporting and tooling. [Aho+06]

2.2.2 Imperative vs. Declarative Languages

Languages used for further computation can be analyzed from multiple perspectives and classified with different qualities in mind. [Van12] For this thesis we focus on slotting languages into two overarching paradigms, **imperative** and **declarative** languages, which differentiate the by inspecting way of reaching the desired outcome.?NOLIKETHIS? Declarative languages describe *what* the computation should accomplish by adhering to sets of expressions, constraints or logical statements. [WGM08] These formulas set the rules and goals characterizing the desired result or relationship of input and output values. The actual control flow and low-level decisions of the execution is left to the specific language implementations and runtime engines.

Generally, because of the vague wording used, the paradigms themselves are not always completely and accurately describing the behaviour of a programming language, possibly leading to different concepts overlapping. This difference is especially clear in modern languages, which often implement functionality by borrowing constructs from the opposite paradigm — for example, imperative C# incorporates declarative features through its LINQ-Library, while Haskell, functional by design, supports imperative-style programming. [**netHaskellClaims**]

Imperative languages, in contrast to declarative ones, describe *how* a computation will accomplish the desired end-state by following explicit sequences of commands. These commands consist of statements that actively manipulate the computations state, which is typically managed through assignments to variables and taking charge of the control flow with provided language constructs (loops, conditionals, calls, procedures, etc.). Consequently, correct execution is no longer responsibility of the underlying system, but the implementation and therefore the programmer itself. The imperative paradigm reflect von Neuman machine — accessing memory and hardware directly. This fined grained control enables the use of low-level optimization techniques [**lowLvelOpt**]. As programmers instruct the machine how to do something, certain performance characteristics and resource constraints can be impelled depending on the context of the computation. Such techniques encompass memory and cache management, *Instruction-Set* optimization or compiler optimization.

Verbose is missing?

However, while this explicit control provides developers with great functional freedom, it also places a greater burden on them to manage every detail of execution. Because the programmer must specify each step — from how data is stored in memory to the order in which operations occur — imperative code can become verbose and cluttered obscuring the high-level intent. Moreover, mutable state and side effects introduce the risk of subtle bugs, especially in concurrent or parallel settings where unintended interactions between state changes can lead to race conditions. Also, formal reasoning and verification are more complex, since proving correctness requires tracking every state transition rather than relying on *referentially transparent* expressions.

Typical use cases for imperative programming include algorithmic or process-oriented problems where an explicit series of steps is natural. Low-level system programming, embedded development, and performance-sensitive routines frequently rely on imperative constructs to manage hardware resources directly. In scenarios that demand fine-grained stateful interactions — such as updating user interfaces incrementally, reading and writing files in a specific order, or controlling physical systems via commands — imperative languages shine by giving developers precise command over execution. Simulation engines, game logic, and scripting or "recipe"-style automation languages also favor imperatively describing each step to achieve the desired outcome.

2.3 Domain-Specific Languages (DSLs)

for choosing between replacing or recalibrating components—but measured data often reflect intertwined effects and closed-loop corrections. to also domain experts. Moreover, DSLs declutter code, rather than describing a subproblem of the domain with multiple lines of code, fewer are needed because constructs inherently carry domain specific information, also contributing to broader comprehensibility for those familiar with the domain. DSLs solutions are also easier to formally verify, as they are restricted to a limited context, than GLPs for which proving correctness is extremely hard.

However, DSLs can suffer from limited scope and flexibility even within their target domain, forcing developers to fall back on general-purpose languages when they need functionality outside the DSL's narrow focus. Furthermore, if the domain itself is complex or rapidly evolving, creating and maintaining a DSL may still de-

mand deep domain expertise, because the abstractions must accurately capture all necessary domain rules and nuances.

2.3.1 Imperative DSLs

Whichever paradigm a DSL follows is mostly guided by the specifics of the domain to describe. Even though DSLs usually adhere to a declarative approach [DKV00], which follows from a major motivation: wanting to describe *what* is expected of the domain in high-level abstraction, a domain may emit qualities that invoke more imperative thinking. If the nature of a domain is inherently composed of sequential actions, consists of procedures or stateful operations, the imperative paradigm is more appropriate. Imperative DSLs require the user to script solutions in step-by-step fashion using domain-specific commands and statements. Essentially,the user embeds domain concepts as a sequential programming model, where the control flow and state transitions of the domain computations have to be managed.

In domains where fine-grained control and explicit ordering matter, imperative DSLs provide maximal flexibility, because every aspect of execution is adjustable. This level of control becomes crucial when timing or order of operations yield different outcomes, or when fine-tuned performance optimizations are required. Furthermore, building an imperative DSL is often simpler than creating a declarative one: typically, an interpreters process each command in sequence, or a translator converts DSL instructions into a host general-purpose language. []

Imperative DSLs are well suited for scripting and defining macros, where users sequence domain-specific commands to drive behavior. They also shine in combination with testing frameworks and scenario specifications, where user interfaces describe interactions by specifying a timely sequence of user inputs. In robotics and industrial automation, imperative DSLs allow to program custom action sequences — such picking up an object, then placing it precisely — where each step must follow a strict order to ensure safety and accuracy. More broadly, any process-control domain that relies on precise sequencing and stateful interactions benefits from the clarity and control that an imperative DSL provides.

Cyber-Physical Systems and Formal Analysis

3.1 Cyber-Physical Systems: Definition and Examples

Cyber-Physical System (CPS) is an umbrella term broadly referring to any systems which tightly integrate both computational logic (cyber) and physical components, allowing interplay through designated interfaces. [Lee15] The two parts of a CPS have to be seamlessly interconnected enabling efficient communication and consistent information relying?? [MS20], as both sides heavily depend on one another. The cyber-physical relationship can be generally describes by three roles *controller/agent*, *sensors* and *activators*, which are connected by an underlying network. Controllers, representing the 'cyber', are occupied with computation and relying fitting instructions based on te observational data provided by the physical sensors. Sensor and Actuators embody the 'physical' and are the ones interacting with the environments in which the system resides. A sensors jobs is to gather intel from its surroundings communicating it in order for the agent to make correct decisions and predictions and to monitor the system. Without sufficient data, the controller is left blind and possibly unable to perform as intended. While sensors passively interact with the environment by observing, actuators do so actively by altering the systems state — carrying out tasks instructed by the controller. ??Looping behaviour??

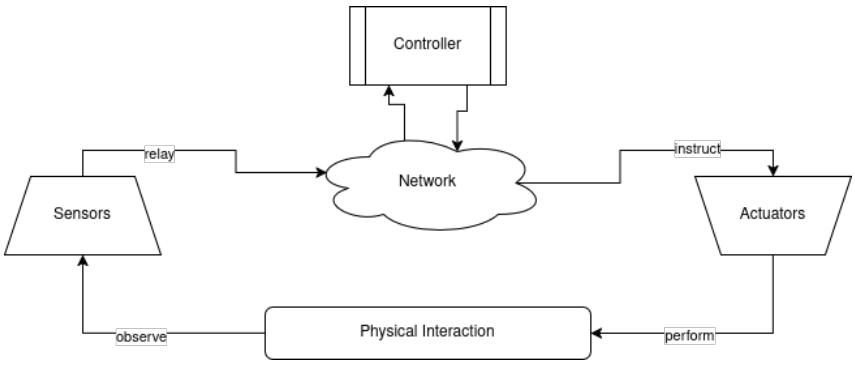


Figure 3.1: General architecture of common CPS

Controllers may range from simple reactive routines to advanced inference models grounded in probabilistic programming, or even intelligent agents capable of autonomous decision-making. Actuators, in turn, translating these decisions into physical action can be represented by basic devices such as motors that move parts, valves that regulate flow, or relays that control circuits. They can also act as more complex assemblies — robotic joints, programmable hydraulic systems, or smart actuators that adjust based on local conditions. At the foundation, sensors continuously monitor the environment, capturing data through modalities such as motion, orientation, or spatial mapping. Together, these elements form an integrated feedback system, enabling CPS to respond intelligently and flexibly to their physical context, regardless of application domain.

CP-Systems are inherently multidisciplinary structures intersecting engineering domains (mechanical, electrical, civil, etc.) with computer science. Because these domains employ differing models for ... , effectively combining them is non-trivial. Challenges arise during development and operation of CPSs for which finding solutions is imperative. [MS20] First and foremost is the concern of **reliability and safety**. Occurrences of failures can have dire real-world consequences with catastrophic end results. This means systems must handle

unexpected conditions gracefully and robustly. But ensuring safety through extensive verification and validation is complicated by the combination of continuous physical dynamics and discrete decision logics. Another challenge is **real-time performance**, as computing elements must keep pace with the physical processes, which usually require low-latency responsiveness including the network. Since CPSs are used in infrastructure and other critical sites that may be targets of cyber attacks, guarantying a high standard of **Security** is essential. Additionally, sensors are not completely reliable — possibly introducing noise to measurements — and environmental conditions can vary drastically leading to **uncertainty**, which a CPS has to cope with to be resilient. [Thr02]

3.2 Formal Methods and Model Checking

CPSs are often deployed in safety-critical environments where utmost importance is placed on reliable operation and functional correctness. Yet reaching high levels of confidence in a systems behavior remains extremely inside cyber-physical domains.[Ask+19] This difficulty arises from the complex interactions of all the individual integral components of the system, which can not be effectively tested only employing traditional methods. The inherent entanglement of these systems warrants the use of specialized procedures that are able to cover and validate the acting components. Unlike conventional simulation based testing, *formal methods* are able to prove correctness against formal specifications, thereby providing more reliable results and eliminating potential error sources, such as race conditions or violations of safety properties(unnoticed states).

In formal methods, *theorem proving*, *model checking*, and *runtime verification* are widely recognized as the three main approaches for assuring system correctness[Kul+22; Ask+19]. All three practices are distinguishable from one another and are tailored for certain scenarios. Runtime verification is a lightweight technique that verifies systems by inspecting their execution trace - often employing automata to detect violations. Theorem proving, in contrast, is an offline and often time intensive process relying on mathematical inference rules to statically validate correctness with a high degree of rigor.

Model checking is of most interest in this case, as the MiniProb DSL 4.4 was developed with this context in mind. [PBC25] It statically verifies a system by exhaustively exploring the state-space and is considered one of the most successful verification techniques. [Ask+19] The thorough analysis of reachable states, combined with specifications which are expressible in various forms of modal logic, offers a strong expressive foundation for property verification [BK08]. However, the systematic exploration of states leads to poor performance, especially for model with complex and big state spaces. Moreover, as most model checking implementation employ automata-based solutions, systems are usually converted into equivalent state machines which can result in an explosion of states.[] Model checking is generally only used in conjunction with finite-state machines for aforementioned reasons. Still, theres been advances trying to apply it to infinite state spaces. [Cla+00; BGP97]. The resulting verdicts for infinite-state models are typically approximations or are restricted by bounded analysis, such as checking only a finite number of steps or abstracting unbounded data domains. As a result, the verification outcomes may not provide absolute guarantees for all possible executions, but rather assurances that hold under specific assumptions, abstractions, or within predefined limits of a system's behavior.

These approaches often rely on *symbolic evaluation* rather than naive enumeration of states. By representing potentially infinite sets of states using logical formulas or data structures such as Binary Decision Diagrams (BDDs) or SMT formulas, symbolic model checking can reason about system behavior without explicitly generating every state.

3.3 Motivation for PPLs in CPS Contexts

Probabilistic programming has multiple fitting applications within CPSs, especially where reasoning under uncertainty is essential. Generally, probabilistic programming allows controllers to plan even in unpredictable environments and settings.

When measurements contain noise or are incorrect, PPL models can still be applied to extract useful information. Instead of treating the raw reading of a sensor as the "truth," the readings are considered samples for a *sensor's model*. Not only is it possible to reduce the noise in the measurements by calculating the posterior, but those noise characteristics are also used to flesh out the predictive model itself. This enables PPLs to adaptively adjust noise models online when, for example, sensors deteriorate over time. Additionally — with most CPSs employing multiple sensors — a PPL naturally supports *sensor fusion*, [cpsSensorFusion] combining data from heterogeneous sources to produce a coherent estimate. By defining each sensor's observation as a conditional distribution, a probabilistic program weights each reading according to its uncertainty: more reliable sensors contribute more heavily, while inconsistent or faulty readings are down-weighted or flagged. Inference then yields a joint posterior over the true state, allowing the system to reconcile disagreements between sensors, detect potential sensor failures by inferring latent bias variables, and maintain a quantified confidence.

Which leads us into robotics, a quintessential cornerstone of CPS. Instead of working on raw observational data and making assumptions in incomplete cases, robots can adopt a *belief* based on an observational model [Thr02]. The observational model couples motion dynamics (how actions translate into state transitions) with existing sensor models, providing a full posterior distribution that the robot can use to imply different levels of confidence for its actions. Furthermore, robots often operate in changing environments or with components that degrade over time. Within a PPL, unknown parameters, such as friction coefficients or sensor calibration offsets, can be treated as latent variables and perform Bayesian learning to infer them from incoming data. The same inference mechanism used for state estimation thus also adapts the model online, continuously refining both motion dynamics and sensor mappings.

A PPL expresses system dynamics probabilistically so that sampling yields a distribution of possible futures. [Hei+18] By enforcing "chance constraints" (e.g., "limit the probability of exceeding a safety threshold to 1%"), inference or Monte Carlo sampling identifies control inputs that optimize expected performance while respecting risk. When the current state is uncertain, planning occurs in belief space — each action triggers a Bayesian update of the state distribution, and planners choose sequences that minimize expected cost given that uncertainty. Simultaneously, online Bayesian learning refines model parameters (like friction or drag) over time, tightening future predictions; as uncertainty shrinks, control can push closer to limits, whereas rising uncertainty leads to more conservative actions.

Recent research is looking at ways to utilize PPLs to gather diagnostics of a CPS. [PB24] They authors propose a way to infer potential error causes, that might be overlooked otherwise, due to the . This of of great interest to system operators for deciding if either parts have to be replaced or only have to be recelebrated. Because measured data on system deviations often reflects complex intertwining causes and *closed-loop corrections*, identifying root causes becomes a matter of inference. The proposed two-step approach first builds a generative model (based on control software and expert insights) that simulates observations from hypothesized error causes. This simulator is then recast as a probabilistic program, allowing Bayesian inference to estimate latent causes (with confidence intervals) from actual measurements.

Probabilistic Programming

4.1 What Is Probabilistic Programming?

Probabilistic programming is a paradigm that aims to perform statistical analysis using tools yielded by computer science.[Mee+21] Unifying general-purpose programming and probabilistic modelling enables the presentation of statistical models as a program. Defined by underlying program code, the model is embodied by the relations between variables and calculations. The established capabilities of programming languages are often augmented with probabilistic functionality, in ... creating probabilistic programming languages (PPL),[Goo13] that are closely associated with this paradigm. The enriched syntax enables the creation of more compact models and fosters conciser abstractions of the underlying probabilistic structure, usually providing constructs that allow declaring random variables and conditioning of observed data, typical for probabilistic modelling. These constructs form the foundation for specifying core elements of statistical models. *Priors* are pre-known and usually drawn from distributions reflecting the initial belief over what value *latent variables*/model parameters might take on. *Likelihoods* are also known beforehand representing how data is assumed to be generated given the latent variable for which the *posterior* is to be inferred. Together, these components form the backbone of a probabilistic model and provide the information required to compute the posterior distribution, which is grounded in **Bayesian theory**.[Mee+21]

Bayes' Theorem provides the foundational framework for probabilistic reasoning under uncertainty. It describes how to update prior beliefs about unknown quantities - known as latent variables - based on new evidence, resulting in a posterior distribution. Formally, it states that the posterior is proportional to the product of the prior and the likelihood:

$$P(\theta \mid x) \propto P(x \mid \theta) \cdot P(\theta)$$

In probabilistic programming, this principle is applied through models that define prior distributions over latent variables and likelihood functions for observed data. The probabilistic programming language then applies Bayes' Theorem, typically via automated inference algorithms, to compute or approximate the posterior. This makes it possible to express complex probabilistic models declaratively, while delegating the computation of posterior beliefs to the underlying inference engine.

4.2 Inference and Sampling Methods

Inference is the process by which probabilistic models are "executed". Given a model and observed data, the task is to compute the posterior distributions of latent variables. Because probabilistic models often include uncertainty and hidden variables, inference is the process that turns the model into concrete answers. This process is abstracted and automated, but remains computationally nontrivial. Standard approaches to inference are generally grouped into three main categories- *exact inference*, *sampling-based methods*, and *variational inference*.[Bis06; BKM17; GS14].

1. Exact Inference

Exact inference computes the posterior distribution analytically or through complete enumeration. This is possible only in models with limited complexity where the entire state space can be traversed. Although conceptu-

ally straightforward, exact inference quickly becomes infeasible as model size increases, due to the combinatorial explosion of possible states.

2. Sampling-Based Inference

Sampling-based inference approximates the posterior by drawing samples from it, typically using Markov Chain Monte Carlo (MCMC) methods. Continuously, random samples are generated from a given distribution - usually the prior or a conditional- and used to progressively approximate the posterior with the model being executed repeatedly. These approaches are flexible and can handle complex models, but they are often computationally expensive and require many samples to ensure accuracy. The resulting samples can then be used to estimate expectations, marginal distributions, or event probabilities.

3. Variational Inference

Variational inference formulates inference as an optimization problem. Instead of sampling, it produces a family of approximate distributions and seeks the member that is closest to the true posterior. This involves defining an objective function - often the *evidence lower bound (ELBO)* - and optimizing it using gradient-based methods. Variational inference tends to be faster and more scalable than sampling, though it trades off exactness for speed by approximating the posterior rather than sampling from it directly [BKM17].

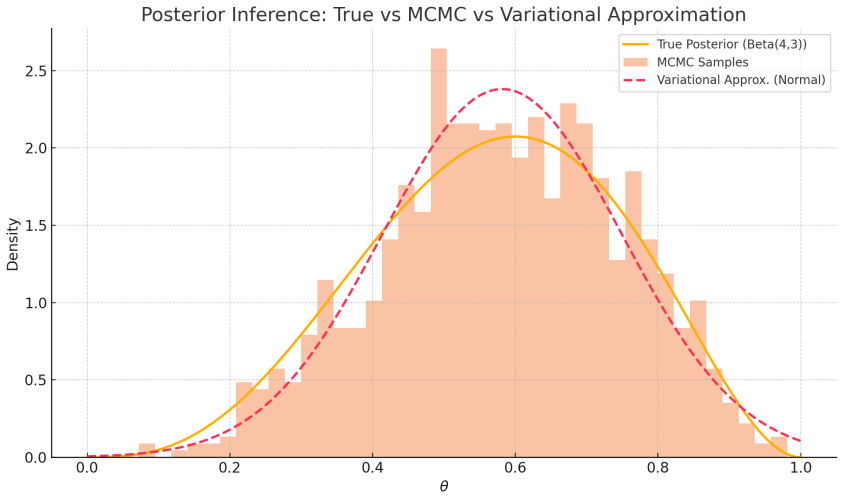


Figure 4.1: Comparison sample of exact posterior to inference methods

4.3 Symbolic Semantics and Model Checking

pOPA ... ref to CPS Model Checking or put chapter after CPS

Inlcude graphic displaying sampling approximations compared to a symbolic state evaluation

4.4 Symbolic Semantics and Model Checking for Probabilistic Programs

While probabilistic programming is typically associated with statistical inference and sampling-based evaluation, there exists an alternative perspective that treats probabilistic programs as formal models subject to verification 3.1. In contrast to traditional inference, symbolic evaluation and model checking aim to rigorously analyze all possible executions of a program against formal specifications.

Symbolic approaches differ in that they do not enumerate individual executions or samples, but instead reason over entire sets of possible states and paths using abstract representations. This enables analysis of program

behavior in a more in-depth and automated way, making it possible to answer questions like: "Does this property always hold, regardless of sampling outcomes?" or "What is the probability that this condition is eventually satisfied?"

Such techniques are particularly relevant for verifying probabilistic systems where correctness, safety, or reachability must be guaranteed rather than estimated. They also enable reasoning about programs that would be challenging to analyze through sampling alone, such as those involving recursion, nondeterminism, or infinite state spaces.

A growing body of foundational work has explored how symbolic and model checking techniques can be adapted to probabilistic settings [Bai+97; EKM06; EY12]. This includes approaches based on probabilistic automata, logics for probabilistic reasoning, and symbolic representations of state spaces.

Building upon these foundations, the POPACheck tool [PBC25] represents a pioneering approach to verifying probabilistic programs with recursive structure. By translating programs into probabilistic operator precedence automata - a formal model that can represent infinite state spaces - POPACheck allows for the analysis of properties expressed in structured temporal logics. This enables verification of systems that lie beyond the scope of most traditional probabilistic programming frameworks.

In this context, *MiniProb* is used not for inference, but as a front-end language for describing probabilistic behavior that is ultimately verified through model checking. The distinction between sampling-based execution and symbolic verification is presented below where an explanation is provided on how MiniProb programs are evaluated and verified using POPACheck.

The *MiniProb* DSL

5.1 Domain and Purpose

5.2 Syntax Overview

5.3 Semantics and Example Models

Language Servers, Parsers & Type Checking

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

6.1 Language Server Protocol (LSP) Basics

6.2 Parser and Syntax Checking

6.3 Type Checking Functionality

6.4 Overview of Langium

Part II

Design & Implementation

Requirements & Technology Survey

7.1 Functional Requirements

7.2 Non-functional Requirements

7.3 Alternative Technologies

7.4 Justification for Langium & VS Code

Architectural Design

8.1 High-Level Architecture Diagram

8.2 Module Decomposition

8.3 Data Flow and Control Flow

Implementation Details

9.1 Langium Grammar Definition for *Miniprob*

9.2 Semantic Checks and Type System

9.3 VS Code Extension Points

Testing & Validation

10.1 Unit Tests

10.2 Integration Tests

10.3 Case Studies & Examples

Part III

Evaluation & Discussion

Performance & Usability

11.1 Parsing Speed

11.2 Editor Responsiveness

11.3 User Feedback

Discussion

12.1 Meeting the Requirements

Bibliography

[Aho+06] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools*. 2nd. Reading, Massachusetts: Addison–Wesley, 2006. ISBN: 978-0321486813.

[Ask+19] Mehrnoosh Askarpour et al. "Formal Methods in Designing Critical Cyber-Physical Systems". In: Oct. 2019, pp. 110–130. ISBN: 978-3-030-30984-8. DOI: 10.1007/978-3-030-30985-5_8.

[Bac+60] J. W. Backus et al. "Report on the algorithmic language ALGOL 60". In: *Commun. ACM* 3.5 (May 1960), pp. 299–311. ISSN: 0001-0782. DOI: 10.1145/367236.367262. URL: https://doi.org/10.1145/367236.367262.

[Bai+97] Christel Baier et al. "Symbolic Model Checking for Probabilistic Processes". In: *Proceedings of the 24th International Colloquium on Automata, Languages and Programming (ICALP)*. Vol. 1234. LNCS. Springer, 1997, pp. 61–73. DOI: 10.1007/BFb0035319.

[BGP97] Tevfik Bultan, Richard Gerber, and William Pugh. "Symbolic model checking of infinite state systems using presburger arithmetic". In: *Computer Aided Verification*. Ed. by Orna Grumberg. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 400–411. ISBN: 978-3-540-69195-2.

[Bis06] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. Vol. 26202649. Jan. 2008. ISBN: 978-0-262-02649-9.

[BKM17] David M Blei, Alp Kucukelbir, and Jon D McAuliffe. "Variational Inference: A Review for Statisticians". In: *Journal of the American Statistical Association* 112.518 (2017), pp. 859–877.

[Cla+00] Edmund Clarke et al. "Counterexample-Guided Abstraction Refinement". In: *Computer Aided Verification*. Ed. by E. Allen Emerson and Aravinda Prasad Sistla. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 154–169. ISBN: 978-3-540-45047-4.

[DKV00] Arie Deursen, Paul Klint, and Joost Visser. "Domain-Specific Languages: An Annotated Bibliography". In: *SIGPLAN Notices* 35 (Jan. 2000), pp. 26–36.

[EKM06] Javier Esparza, Antonín Kučera, and Richard Mayr. "Model Checking Probabilistic Pushdown Automata". In: *Logical Methods in Computer Science* 2.1 (2006), 2:1–2:31. DOI: 10.2168/LMCS-2(1:2)2006.

[EY12] Kousha Etessami and Mihalis Yannakakis. "Model Checking of Recursive Probabilistic Systems". In: *ACM Transactions on Computational Logic (TOCL)* 13.2 (2012), 12:1–12:40. DOI: 10.1145/2159531.2159534.

[Goo13] Noah D. Goodman. "The principles and practice of probabilistic programming". In: *SIGPLAN Not.* 48.1 (Jan. 2013), pp. 399–402. DOI: 10.1145/2480359.2429117. URL: https://doi.org/10.1145/2480359.2429117.

[GS14] Noah D. Goodman and Andreas Stuhlmüller. *The Design and Implementation of Probabilistic Programming Languages*. http://dippl.org. Accessed: 2024-01-01. 2014.

[Hei+18] Tor Aksel N. Heirung et al. "Stochastic model predictive control — how does it work?" In: *Computers and Chemical Engineering* 114 (2018). FOCAPO/CPC 2017, pp. 158–170. DOI: 10.1016/j.compchemeng.2017.10.026. URL: https://doi.org/10.1016/j.compchemeng.2017.10.026.

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

[Jina] Pete Jinks. *BNF/EBNF Variants*. https://www.cs.man.ac.uk/~pjj/bnf/ebnf.html. Last modified 12 March 2004. Accessed 26 May 2025.

[Jinb] Pete Jinks. *Notations for Context-Free Grammars: BNF, Syntax Diagrams, EBNF*. https://www.cs.man.ac.uk/~pjj/bnf/bnf.html. Last modified 12 March 2004. Accessed 26 May 2025.

[Kul+22] Tomas Kulik et al. “A Survey of Practical Formal Methods for Security”. In: *Formal Aspects of Computing* (2022). Published Version. ISSN: 1433-299X. DOI: 10.1145/3522582. URL: https://eprints.whiterose.ac.uk/194011/.

[Lee15] Edward A. Lee. “The past, present and future of cyber-physical systems: a focus on models”. In: *Sensors (Basel)* 15.3 (Feb. 2015), pp. 4837–4869. DOI: 10.3390/s150304837.

[Mee+21] Jan-Willem van de Meent et al. *An Introduction to Probabilistic Programming*. 2021. DOI: 10.48550/arXiv.1809.10756. URL: https://arxiv.org/abs/1809.10756.

[MS20] Abha Mahalwar and Rishabh Sharma. “Cyber-Physical Systems: Challenges and Future Directions”. In: *Turkish Journal of Computer and Mathematics Education (TURCOMAT)* 11 (Dec. 2020), pp. 2865–2870. DOI: 10.61841/turcomat.v11i3.14651.

[PB24] A. Piedrafita and L. Barbini. “PHM Society European Conference”. In: vol. 8. 1. June 2024, p. 7. DOI: 10.36001/phme.2024.v8i1.4055.

[PBC25] Francesco Pontiggia, Ezio Bartocci, and Michele Chiari. *POPACheck: a Model Checker for Probabilistic Pushdown Automata*. arXiv preprint arXiv:2502.03956. Conditionally accepted at CAV 2025. TU Wien, 2025.

[SK95] Kenneth Slonneger and Barry L. Kurtz. “Specifying Syntax”. In: *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. Reading, Massachusetts: Addison–Wesley, 1995. Chap. 1, pp. 1–30. ISBN: 0-201-65697-3.

[Thr02] Sebastian Thrun. “Probabilistic robotics”. In: *Communications of the ACM* 45.3 (Mar. 2002), pp. 52–57. DOI: 10.1145/504729.504754. URL: https://doi.org/10.1145/504729.504754.

[Van12] Peter Van Roy. “Programming Paradigms for Dummies: What Every Programmer Should Know”. In: (Apr. 2012).

[WGM08] Hui Wu, Jeff Gray, and Marjan Mernik. “Grammar-driven generation of domain-specific language debuggers”. In: *Softw., Pract. Exper.* 38 (Aug. 2008), pp. 1073–1103. DOI: 10.1002/spe.863.

Expanded Code Listings

Test Data