ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

Type-Safe Complex Specifications in Kotlin: a DSL to Configure the Alchemist Simulator

Tesi di laurea in:
SOFTWARE PROCESS ENGINEERING

Relatore
Prof. Gianluca Aguzzi

Correlatori
Dott. Danilo Pianini

Candidato
Marco Frattarola

I Sessione di Laurea
Anno Accademico 2024-2025

Abstract

Max 2000 characters, strict.

Optional. Max a few lines.

Contents

List of Figures

List of Listings

Chapter 1

Introduction

The simulation of complex systems has become a fundamental tool in modern scientific research and engineering practice. Complex systems are characterized by the interaction of numerous components that give rise to emergent behaviors that cannot be easily predicted from the individual components alone. These systems span diverse domains, from biological networks and chemical reactions to distributed computing systems, social dynamics, and swarm robotics. Understanding and predicting the behavior of such systems requires sophisticated simulation frameworks capable of modeling spatial distributions, temporal dynamics, and stochastic interactions.

Alchemist is a simulation framework designed to address these challenges by providing a powerful platform for modeling spatially-distributed systems with temporal dynamics. However, as the complexity of simulated systems increases, the current YAML-based configuration system presents significant challenges that limit productivity and reliability. This thesis addresses these limitations by developing a type-safe Domain-Specific Language (DSL) in Kotlin for Alchemist configuration. The DSL leverages Kotlin's type system and IDE support to provide compile-time error detection, autocomplete capabilities, and improved developer experience, while maintaining semantic equivalence with the existing YAML configuration system.

The remainder of this thesis is organized as follows. Chapter 1.2 provides background information on Alchemist, including its application domains, use cases, and the motivation that led to this work. Chapter 2 analyzes the requirements and constraints for developing the DSL, examining the limitations of the current YAML approach and establishing the domain model. Chapter 3 presents the design of the DSL, including its architecture and key design decisions. Chapter 4 describes the implementation details and challenges encountered during development. Chapter 5 evaluates the DSL through case studies and comparison with YAML configurations. Finally, Chapter 6 summarizes the contributions and discusses future work.

1.1 Context and Motivation

1.1.1 Configuring Complex Systems

Configuration as a Fundamental Mechanism

Configuration constitutes a fundamental mechanism through which users specify the structure, behavior, and parameters of complex systems without modifying the underlying implementation. In the context of computational systems, configuration encompasses the specification of runtime characteristics, architectural topology, behavioral policies, and experimental parameters that govern system execution. Rather than encoding these aspects directly into program logic, configurations externalize them into separate artifacts that can be modified, versioned, and reused independently of the core codebase. This separation of concerns enables a single implementation to serve diverse use cases through different configurations. For example, in a simulation framework like Alchemist (add citation) a simulation can be defined by creating a configuration file that specifies the all the needed parameters such as the initial state and the behavior of its components. This configuration file can then be used to create a simulation instance that can be run.

Imperative and Declarative Configuration Paradigms

Configuration management systems can be classified according to two distinct paradigms that fundamentally differ in their approach to system specification and manipulation:

- **Imperative paradigm**
- **Declarative paradigm**

This distinction, rooted in the dichotomy between procedural and declarative programming models, has profound implications for how systems are configured, maintained, and reasoned about.

The *imperative configuration* paradigm focuses on specifying **how** a system should reach its desired state through explicit sequences of commands and operations. An imperative configuration defines the procedural **steps** required to transform the system from its current state to the target state. Each command represents an action that modifies system state, and the execution order is critical to achieve the intended outcome. Shell scripts, deployment automation tools like Ansible in its command-based mode and traditional system administration procedures exemplify this approach. An example of an imperative configuration is *Dockerfile* configuration file: Docker is an open platform for developing, shipping, and running applications (citation) A *Dockerfile* is the Docker's imperative configuration format for building container images. It is a text file containing instructions for building source code.

```
FROM ubuntu:22.04

# install app dependencies
RUN apt-get update && apt-get install -y python3 python3-pip
RUN pip install flask==3.0.*

# install app
COPY hello.py /

# final configuration
ENV FLASK_APP=hello
EXPOSE 8000
CMD ["flask", "run", "--host", "0.0.0.0", "--port", "8000"]
```

In this example, each `RUN`, `COPY`, and `CMD` instruction represents a step that modifies the container image state, and the sequence must be carefully ordered (e.g., installing packages before executing files that depend on them). Docker then, builds images by reading the instructions from a Dockerfile.

In contrast, the declarative configuration paradigm focuses on specifying **what** the desired system state should be, abstracting away the procedural details of how that state is achieved. A declarative configuration describes the target state as a set of constraints, properties, and resource specifications, delegating to the configuration management system the responsibility of determining and executing the necessary operations to realize that state. The system automatically computes the necessary set of changes required to achieve the desired state. Tools such as Puppet, Terraform, Kubernetes manifests, and Docker Compose exemplify this paradigm. Docker Compose, for example, allows developers to declare the desired state of multi-container applications:

```yaml
services:
  web:
    build: .
    ports:
      - "8000:5000"
  redis:
    image: "redis:alpine"
```

The declarative paradigm offers several compelling advantages that have driven its adoption across modern infrastructure management.

1. First, declarative configurations are inherently more readable and maintainable, as they document the intended system state without obscuring it with implementation details.

2. They are idempotent by design: applying the same configuration multiple times produces the same result, eliminating concerns about partial execution or repeated application.

3. They facilitate reasoning about system properties through formal verification and constraint validation before deployment.

4. They support more robust error handling, as the system can detect when the desired state cannot be achieved and report specific discrepancies.

Application Domains and Representative Systems

Declarative configuration systems have proliferated across diverse domains of software infrastructure, each addressing specific challenges while adhering to common architectural principles. Understanding these application domains and their representative technologies provides context for the subsequent discussion of configuration challenges and the motivation for type-safe configuration approaches.

Infrastructure Configuration Management. Puppet pioneered the declarative configuration management paradigm for physical and virtual server infrastructure. (citation here) Its resource abstraction model allows administrators to declare desired states for files, packages, services, users, and other system entities, with the Puppet agent responsible for ensuring these resources match their specifications. Chef followed a similar philosophy while embedding configuration logic within a Ruby domain-specific language, providing both declarative resource declarations and imperative programming constructs. These tools revolutionized system administration by replacing ad-hoc shell scripts with version-controlled, testable configuration artefacts.

Cloud Infrastructure Provisioning. Terraform extended declarative configuration to cloud resource provisioning, allowing operators to declare entire cloud environments, including compute instances, networks, load balancers, databases, and access policies, using a unified configuration language. Its provider ecosystem supports virtually all major cloud platforms and services, enabling multi-cloud deployments through consistent declarative specifications.

Container Orchestration. Kubernetes represents perhaps the purest expression of declarative configuration in modern infrastructure. The Kubernetes API defines a comprehensive schema of resource type such as Pods, Services, Deployments, ConfigMaps, Secrets, and dozens of others, that collectively describe application topology, networking, storage, and operational policies. Users express desired states through YAML or JSON manifests, and Kubernetes controllers continuously reconcile actual cluster state with declared specifications.

Application Deployment and Composition. Docker Compose applies declarative configuration to application deployment, allowing developers to specify multi-container applications as collections of services with defined dependencies, networking, and resource constraints.

Simulation and Experiment Configuration. Scientific simulation frameworks like Alchemist employ declarative configuration to specify complex experimental scenarios. This approach enables researchers to express sophisticated experimental designs without imperative programming, facilitating reproducibility and systematic parameter exploration.

These systems, while addressing distinct domains, share common architectural patterns:

- **Resource schema**: defines the universe of entities that the configuration system can manage, specifying the types of resources, their properties, validation rules, and semantic constraints.
- **Concrete syntax**: provides a human-readable, version-controllable format for expressing configurations, supporting necessary abstraction, parameterization, and composition. Some systems, like Puppet and Terraform, provide a domain-specific languages with built-in abstraction primitives such as modules, variables, and iteration constructs. Others, like Kubernetes, define a pure data format (JSON/YAML). Different concrete syntaxes can target the same underlying resource schema.
- **Configuration management engine**: it is responsible for determining a valid sequence of operations to achieve the desired state (respecting dependencies and constraints), and execute those operations while handling errors and partial failures.

1.1.2 Challenges and Limitations

Despite the architectural sophistication of declarative configuration systems, the concrete syntax layer has converged on a remarkably narrow set of data serialization formats. XML (Extensible Markup Language), JSON (JavaScript Object

Notation), YAML (YAML Ain't Markup Language) have emerged as one of the most used data formats for expressing configuration artifacts across different domains of modern software infrastructure. This convergence reflects their perceived advantages: human readability, widespread tool support, and interoperability with programming language data structures. However, these formats have fundamental limitations that become increasingly problematic as configuration complexity grows.

Lack of Type Safety

XML, JSON and YAML are fundamentally *untyped* data formats. They represent data structures (objects, arrays, strings, numbers, booleans, and null values) without any mechanism for expressing type constraints, required fields, or value ranges. Type information exists only implicitly through runtime validation performed by the consuming application, which must parse the configuration and verify that values conform to expected types and constraints. This validation occurs entirely at runtime, meaning configuration errors surface only when the system attempts to load or execute the configuration, potentially after significant development effort has been invested.

Beyond the absence of type constraints, XML, JSON and YAML are fundamentally limited to primitive data types supported by the serialization format itself. These formats natively support only basic scalar types along with composite structures (objects and arrays) composed of these primitives. This limitation becomes particularly problematic when configuration systems need to express domain-specific concepts that map to programming language classes, interfaces, or complex object types.

Consider simulation frameworks like Alchemist, where configurations must specify classes that will be instantiated at runtime to represent domain entities. The YAML configuration expresses these class specifications as string literals:

```yaml
incarnation: protelis
environment:
  type: Continuous2DEnvironment
```

```yaml
network-model:
  type: ConnectWithinDistance
  parameters: [6]

deployments:
  type: Rectangle
  parameters: [1, 0, 0, 100, 100]
```

Here, class names such as `Continuous2DEnvironment`, `ConnectWithinDistance` and `Rectangle` are specified as plain strings. The application must use reflection or factory patterns to resolve these string class names to actual Java or Kotlin classes at runtime, instantiate them with the provided parameters, and verify that the constructor signatures match. This approach introduces several critical problems such as:

- **No compile-time verification**: The class name is merely a string literal, so there is no way to verify at configuration authoring time that the specified class exists, is accessible, or implements the expected interface.
- **No constructor parameter validation**: The parameters array must match the constructor signature, but this compatibility cannot be checked statically. A mismatch between parameter count, types, or order is discovered only when instantiation fails at runtime.
- **No type inference or IDE support**: Since class names are strings, Integrated Development Environments (IDEs) cannot provide autocomplete suggestions, type-aware parameter hints, or navigation to class definitions.
- **Refactoring**: Renaming a class in the codebase requires manually updating all string references in configuration files, with no tooling support to ensure completeness.

This pattern of specifying class names as strings for runtime instantiation is not unique to Alchemist. Many frameworks employ similar approaches across different domains. The Spring Framework, for instance, allows developers to define beans and their dependencies in XML configuration files, where class names are specified as strings:

```xml
<bean id="item1" class="org.baeldung.store.ItemImpl1" />
<bean id="store" class="org.baeldung.store.Store">
    <constructor-arg type="ItemImpl1" index="0" name="item" ref="item1" />
</bean>
```

Spring's dependency injection container resolves these string class names at runtime, instantiating the classes and wiring dependencies. Moreover, when configurations involve nested object construction, where one class requires another class as a parameter, the problem compounds. Consider an Alchemist configuration specifying a time distribution that itself requires other time distribution objects:

```yaml
time-distribution:
  type: JaktaTimeDistribution
  parameters:
    sense:
      type: WeibullTime
      parameters: [1, 1]
    deliberate:
      type: DiracComb
      parameters: 0.1
    act:
      type: ExponentialTime
      parameters: 1
```

The absence of compile-time type checking means that refactoring configurations is inherently risky. Renaming a service, changing a parameter structure, or modifying nested configurations requires manual verification that all references have been updated correctly. In complex configurations spanning multiple files or involving hundreds of interdependent components, this manual process is error-prone and time-consuming. Type mismatches, missing required fields, and inconsistent parameter combinations are discovered only through runtime failures, forcing practitioners to debug through opaque error messages and stack traces rather than receiving immediate, actionable feedback during configuration authoring.

Limited Developer Experience

The lack of type information severely limits the capabilities of Integrated Development Environments (IDEs) and text editors. Without explicit schemas or type definitions, IDEs cannot provide meaningful autocomplete suggestions, inline documentation, or real-time error detection. Developers must rely on external documentation to understand what configuration options are available, what types are expected, and what constraints apply. This absence of IDE support increases the learning curve for new users and reduces productivity even for experienced practitioners.

Schema validation mechanisms exist for JSON, YAML, and XML formats, offering some degree of structural validation and improved IDE support. JSON Schema (citation here) provides a standardized way to define the expected structure of JSON documents, while XML Schema Definition (XSD)(citation here) offers similar capabilities for XML. YAML configurations can leverage JSON Schema or YAML-specific schema definitions to validate document structure, required fields, and basic type constraints.

However, schema validation provides only partial solutions to the fundamental limitations of untyped configuration formats. While schemas can validate *syntax*, ensuring that required fields are present, that values conform to expected primitive types, and that the document structure matches a defined pattern, they provide limited support for *domain logic* and *semantic validation*.

In principle, a schema author could enumerate all possible class names that may appear in a `type` field, allowing the schema to validate that the string represents one of the known classes. Similarly, a schema could specify the expected parameter count and types for each enumerated class, enabling validation of constructor compatibility. However, this approach requires the schema author to maintain a complete, exhaustive list of all classes that may be instantiated, along with their constructor signatures. For closed systems with a fixed set of types, this is feasible and can provide meaningful validation.

The fundamental challenge arises in extensible software systems like Alchemist, where the framework's plugin architecture allows users to define custom incarnations, environments, network models, and reaction types. In such systems, the set of valid class types is open-ended and cannot be fully captured in a static schema definition. A schema author attempting to enumerate all possible classes would face an insurmountable task: the schema would need to be updated every time a new plugin or extension is developed, and it would still fail to validate configurations that use custom classes unknown to the schema at the time of schema definition. Even if a schema were maintained to include all built-in classes, configurations using user-defined custom classes would remain unvalidated, defeating the purpose of comprehensive validation.

Moreover, schemas cannot validate semantic relationships that require understanding the runtime behavior of classes. For instance, schemas cannot ensure that YAML anchor(citation here) references resolve correctly, that variable dependencies form a valid acyclic graph, or that mathematical expressions in configuration values are well-formed and type-safe. These validations require understanding the semantics of the configuration language and the relationships between configuration elements, which goes beyond what structural schemas can express.

These schemas are typically external to the configuration file itself and require explicit configuration of IDE plugins or validation tools. The developer experience contrasts sharply with that of statically-typed programming languages, where IDEs provide comprehensive autocomplete, type checking, refactoring support, and inline documentation based on the language's type system.

Readability and Maintainability Challenges

As configuration complexity increases, JSON and YAML files become increasingly difficult to read, understand, and maintain. The hierarchical structure of these formats, while useful for simple configurations, becomes unwieldy when dealing with systems involving dozens of components, deeply nested structures, or complex interdependencies. It is not uncommon to see configurations with hundreds of lines of code, making it difficult to understand.

Another issue of untyped configuration formats is the lack of mechanisms to reduce duplication within a single file. Some formats do provide limited mechanisms to reduce duplication such as YAML that supports anchors and aliases (`&anchor` and `*alias`) (citation here) that allow defining a value once and referencing it multiple times within the same document. However, these mechanisms are fundamentally limited: they operate only within a single file, cannot reference external files, and provide no type safety or compile-time validation of references. JSON provides no built-in mechanism for reuse whatsoever, requiring complete duplication of common patterns.

More critically, none of these formats provide standardized mechanisms for dividing configurations across multiple files and reusing components across different configuration files, as one would expect in a programming language where classes and functions can be defined in separate modules and imported where needed. While XML provides standardized mechanisms for including external files (such as XInclude)(citation here), YAML and JSON have no such standard capabilities. Some tools provide non-standard include mechanisms (e.g., Ansible's `!include` directive)(citation here), but these are tool-specific, non-portable, and lack compile-time validation. This limitation forces users of the configuration system to either maintain monolithic configuration files or rely on external preprocessing tools that merge files before parsing, neither of which provides the type safety, IDE support, or refactoring capabilities that would be available if configurations could be structured as reusable, type-checked modules similar to programming language components. The limitations of JSON and YAML become particularly pronounced when configuring complex scenarios such as scientific simulations and experimental configurations. Expressing complex scenarios in YAML results in very long files that tends to grow more vertically than horizontally, where the semantic relationships between configuration elements are obscured by syntactic boilerplate.

```
incarnation: scafi

_constants:
  retentionTime: &retentionTime 15.0

variables:
  seed: &seed
    min: 0
    max: 1
    step: 1
    default: 0
  spacing: &spacing
    formula: 0.5
  longSideNodes: &xNodes
    formula: 3
  shortSideNodes: &yNodes
    formula: 4
  error: &error
    formula: 0
  partitioning: &partitioning
    language: scala
    formula: |
      import it.unibo.learning.model.IID
      IID
  experiment: &experiment
    formula: "\"MNIST\""
  batchSize: &batchSize
    formula: 32
  epochs: &epochs
    formula: 2
  areas: &areas
    formula: 9
seeds:
  scenario: *seed
  simulation: *seed

network-model:
  type: ConnectWithinDistance
  parameters: [1.5]

layers:
  - type: PhenomenaDistribution
    parameters: [0, 0, *xNodes, *yNodes, *areas,
   ↪  *partitioning, *experiment, 0.8, *seed]
    molecule: Phenomena

_gradient: &gradient
  - time-distribution:
      type: DiracComb
      parameters: [ 0.2, 1 ]
    type: Event
    actions:
      - type: RunScafiProgram
        parameters: [it.unibo.scafi.CentralizedClient,
       ↪  *retentionTime]
  - program: send

environment:
  type: Continuous2DEnvironment
  parameters: [ ]
  global-programs:
```

```
    - time-distribution:
        type: Trigger
        parameters: [0]
      type: PhenomenaToDataset
      parameters: []
    - time-distribution:
        type: Trigger
        parameters: [ 0.1 ]
      type: ModelAndSeedInitialization
      parameters: [*seed, *experiment]
    - time-distribution:
        type: DiracComb
        parameters: [ 0.3, 1 ]
      type: CentralServerFL
      parameters: [ ]

monitors:
  - type:
 ↪    it.unibo.alchemist.model.monitors.CentralizedTestSetEvaluation
    parameters: [*batchSize, *experiment, *areas, *seed]

deployments:
  type: Grid
  parameters: [0, 0, *xNodes, *yNodes, *spacing, *spacing,
 ↪    *error, *error]
  contents:
    - molecule: BatchSize
      concentration: *batchSize
    - molecule: Epochs
      concentration: *epochs
    - molecule: Seed
      concentration: *seed
#    - molecule: Experiment
#      concentration: *experiment
  programs: *gradient

terminate:
  type: AfterTime
  parameters: 50

export:
  - type: CSVExporter
    parameters:
      fileNameRoot: "experiment"
      interval: 1.0
      exportPath: "data"
    data:
      - time
      - molecule: TrainingLoss
        aggregators: [min, max, mean, variance]
        value-filter: onlyFinite
        precision: 3
      - molecule: ValidationLoss
        aggregators: [min, max, mean, variance]
        value-filter: onlyfinite
      - molecule: ValidationAccuracy
        aggregators: [min, max, mean, variance]
        value-filter: onlyfinite
```

Parameter references must be expressed as strings, making it impossible to verify referential integrity statically. Mathematical expressions and conditional logic must be encoded as strings or complex nested structures, losing the clarity and type safety that programming language constructs would provide. The lack of abstraction mechanisms beyond simple variable substitution means that common patterns must be repeated rather than abstracted into reusable components.

Error Handling and Runtime Validation

As configuration systems scale to complex scenarios, the error handling and validation mechanisms in YAML, JSON, and XML reveal fundamental limitations that make debugging difficult and error-prone. Unlike statically-typed programming languages where errors are caught at compile-time with precise location information and clear diagnostic messages, configuration formats provide only runtime validation, and the error messages they produce are often ambiguous, unhelpful, or point to internal parser mechanisms rather than the actual configuration problem.

One of the most insidious aspects of YAML's indentation-based syntax is that a misindented line can produce a syntactically valid document that parses successfully, but with a completely different semantic meaning than intended. Consider the following Alchemist configuration:

```yaml
incarnation: sapere
environment:
  type: Continuous2DEnvironment
  parameters: [100, 100]
network-model: {type: ConnectWithinDistance, parameters: [5]}
```

If a single extra space is accidentally added before `network-model`, the YAML parser will interpret it as a nested key under `environment`:

```yaml
incarnation: sapere
environment:
  type: Continuous2DEnvironment
  parameters: [100, 100]
  # Extra tab before this line
  network-model:  {type: ConnectWithinDistance, parameters: [5]}
```

This configuration is syntactically valid and will parse without error, but the `network-model` will be incorrectly nested under `environment` rather than being a top-level configuration key. In this case the application will produce two different simulation results:

- The first file will produce a simulation with the specified network model (*ConnectWithinDistance*).
- The second file will produce a simulation where the network model is ignored, and the default network model is used (*NoLinks*).

The user must manually inspect the file, and check indentation levels to identify the root cause, a time-consuming process that becomes even more difficult as configuration files grow to hundreds or thousands of lines. These kinds of errors are particularly insidious because they are not detected until the simulation is executed, potentially after significant computation has already occurred. The problem compounds when configurations reference values that are validated only at application runtime. For example, an Alchemist configuration might specify a class name that does not exist:

```yaml
environment:
  type: NonExistentEnvironmentClass
  parameters: [100, 100]
```

The YAML parser will accept this configuration as valid, since `NonExistentEnvironmentClass` is a perfectly valid string. The error will only surface when the application attempts to instantiate the class at runtime. As configuration complexity increases, with complex nested structures, and interdependent parameters, these error handling limitations become severe productivity bottlenecks. Practitioners spend significant time debugging configuration errors that could be caught immediately with compile-time validation, clear error messages, and IDE support that highlights problems as they type. The lack of early error detection means that configuration errors are discovered only after the system attempts to execute, potentially after significant computation has already occurred or after a deployment has been initiated, making the debugging process even more costly and frustrating.

1.1.3 The Need for a Type-Safe Configuration Language

Recent configuration languages demonstrate that declarativity can coexist with type safety and tooling support. CUE unifies data, schemas, and policies inside a single constraint system, enabling incremental validation, order-independent unification, and reusable templates that reduce boilerplate [?]. Dhall delivers a total, strongly typed language whose expressions normalize deterministically and can be exported to YAML or JSON without sacrificing safety guarantees [?]. The proposed solution aims to provide a type-safe configuration language for the Alchemist simulator, using Kotlin(citation here) as the implementation language. The need for a type-safe configuration language is motivated by the limitations of the current YAML-based configuration system, as described in the previous section, and by the requirement for a more powerful configuration mechanism that can accommodate the extensibility of the Alchemist simulator and the complex scenarios it enables.

This thesis proposes a Kotlin-based Domain-Specific Language as the solution. Kotlin was selected for several reasons: first, the language provides first-class support for building domain-specific languages through its expressive DSL builders, type-safe builders, and extension functions that enable creating fluent, readable configuration APIs. Second, Alchemist is a JVM-based framework originally implemented in Java, Kotlin, and Scala, which means Kotlin is already natively supported within the Alchemist ecosystem without requiring additional runtime dependencies or cross-language interoperability layers. This native integration ensures seamless compatibility with existing Alchemist codebases and enables direct access to the framework's APIs and domain models. Unlike YAML configurations where class names must be specified as strings (e.g., `type: Continuous2DEnvironment`), the Kotlin DSL allows users to directly reference simulation domain objects and classes as first-class language constructs, eliminating the need for string-based class name resolution and enabling compile-time verification that the referenced classes exist and are accessible.

Kotlin's static type system, combined with its DSL capabilities and IDE ecosystem, makes it possible to encode domain entities and relationships as first-class constructs that the compiler can verify. Type safety is therefore elevated from an ergonomic preference to a correctness requirement: configuration authors receive compile-time diagnostics, refactoring support, and reusable libraries, while the simulator benefits from semantically sound artifacts that can be validated before execution.

1.2 Background

This chapter provides the background and motivation for this thesis work. We begin by introducing the Alchemist simulator, describing what it is and how it is used across various application domains. We then present concrete use cases that demonstrate the framework's capabilities and importance. Finally, we examine the limitations of the current configuration system and explain why these limitations motivated the development of a type-safe Domain-Specific Language, establishing the foundation for this thesis contribution.

1.2.1 Alchemist

Alchemist is a simulation framework for modeling spatially-distributed systems with temporal dynamics. The project originated in 2010 within the context of the European SAPERE project[1] and has since developed into a general-purpose simulation platform. The framework employs a chemical-inspired computational model where system evolution emerges from reactions occurring between entities distributed in space.

The chemical metaphor provides a natural abstraction for modeling various phenomena. In chemical systems, molecules react according to rules that depend on their concentrations and spatial proximity. Alchemist adapts this model to computational settings: nodes represent entities that contain molecules (data) with associated concentrations (values), and reactions define how these entities interact and evolve over time. This approach has been applied to domains including biochemical reaction networks, distributed algorithms in wireless sensor networks, aggregate programming systems, crowd dynamics, epidemic spreading, and swarm robotics.

[1]Self-Aware Pervasive Service Ecosystems

The framework implements stochastic, event-driven simulation using variants of Gillespie's algorithm, enabling efficient execution even with large numbers of mobile entities.

Core Concepts and Meta-Model

Application Domains

Alchemist has been employed to simulate complex scenarios across multiple domains, demonstrating the framework's versatility and effectiveness:

- **Biochemical reaction networks**: Modeling the dynamics of chemical reactions in biological systems, where molecules represent chemical species and reactions model biochemical processes. These simulations enable researchers to understand how complex biochemical networks behave under different conditions and parameter settings.
- **Distributed algorithms in wireless sensor networks**: Simulating communication protocols and distributed algorithms where nodes represent sensors that communicate within limited ranges. The spatial distribution and mobility of nodes create complex interaction patterns that can be effectively modeled using Alchemist's reaction-based approach.
- **Aggregate programming systems**: Simulating field-based computing paradigms where computations emerge from local interactions between nodes. The framework supports incarnations for Protelis and ScaFi, enabling the simulation of aggregate programming algorithms in realistic spatial scenarios.
- **Crowd dynamics**: Modeling the movement and behavior of crowds in various environments. The spatial distribution of individuals and their local interactions create emergent behaviors that can be studied through simulation.
- **Epidemic spreading**: Simulating the spread of diseases through populations, where nodes represent individuals and reactions model infection and

recovery processes. The spatial distribution and mobility patterns significantly influence the epidemic dynamics.

- **Swarm robotics**: Modeling the collective behavior of robotic swarms where individual robots interact locally to achieve global objectives. The framework enables the study of how local interactions give rise to coordinated swarm behaviors.

These diverse applications demonstrate Alchemist's capability to model complex systems scenarios where spatial distribution, temporal dynamics, and stochastic interactions play crucial roles. The framework's flexibility allows researchers to adapt the simulation model to their specific domain requirements while leveraging a unified simulation engine.

Alchemist's meta-model consists of several core abstractions. A *molecule* identifies a data item, while its *concentration* represents the associated value. This terminology derives from chemistry but applies generically: molecules and concentrations can represent any data type, enabling the framework to model diverse domains.

Nodes are containers that hold molecules and reactions. Each node maintains local state through its molecules and executes reactions that modify this state. Nodes exist within an *environment*, which provides spatial services including position tracking, distance computation, and optional mobility support. Environments may be continuous (Euclidean spaces) or discrete (graphs, grids), accommodating different spatial modeling needs.

Linking rules determine connectivity between nodes based on the environment state. Each linking rule maps nodes to *neighborhoods*, consisting of a center node and its neighbors. This mechanism models communication constraints, such as distance-limited wireless networks where nodes interact only with nearby neighbors.

Reactions define system behavior. Each reaction comprises conditions, actions, and a time distribution. Conditions evaluate the environment state, returning both a boolean (enabling the reaction) and a numeric value (influencing the reaction rate). Actions modify the environment when the reaction fires. The time distribution determines reaction timing based on an instantaneous rate computed from static parameters and condition values. This enables stochastic modeling where reaction probabilities depend on current system state.

Alchemist's extensibility relies on *incarnations*, which define type systems for concentrations and specialize the framework for specific domains. The SAPERE incarnation, the original implementation, treats concentrations as numeric values representing chemical amounts. The Protelis incarnation integrates the Protelis aggregate programming language for distributed algorithm simulation. The Scafi incarnation supports the ScaFi framework for field-based computing. The Biochemistry incarnation provides detailed biochemical reaction modeling capabilities. This architecture enables a unified simulation engine to support multiple modeling paradigms.

The framework implements stochastic simulation using Gillespie's algorithm and variants for event scheduling. Environments track node positions and update neighborhoods dynamically, supporting mobile entities. The implementation targets efficiency, handling simulations with thousands of nodes. A plugin-based architecture enables extensions through new incarnations, environments, and reaction types. The system includes a graphical interface for real-time visualization, supports batch execution with parameter sweeps and variable dependencies, and provides data export capabilities for post-simulation analysis.

The limitations of the current YAML-based configuration system have become increasingly problematic as Alchemist is used for more complex simulations across diverse domains. This section examines these limitations in detail and explains why they motivated the development of a type-safe Domain-Specific Language, which constitutes the main contribution of this thesis.

Alchemist simulations are currently configured via YAML files specifying the incarnation type, environment configuration, network model, node deployments, initial molecule concentrations, reaction definitions, batch simulation variables, and export settings. While YAML offers human readability and a relatively simple syntax for basic configurations, it suffers from fundamental limitations that become increasingly problematic as simulation complexity grows.

The YAML-based configuration system exhibits several critical limitations that hinder productivity and reliability, particularly as simulation complexity increases.

One of the primary limitations of the YAML-based configuration system is the lack of early error detection. Configuration errors surface only at runtime, when the simulation is being loaded or executed. These runtime errors often provide opaque and difficult-to-interpret messages that make debugging challenging. For instance, a typo in a molecule name or an incorrect parameter type may not be detected until the simulation attempts to access that configuration, potentially after significant computation has already occurred. The error messages typically reference internal data structures and parsing mechanisms rather than the user's configuration, making it difficult to identify and fix the root cause of the problem.

This opacity is particularly problematic when working with complex simulations that involve numerous nodes, reactions, and dependencies. A single configuration error can prevent the entire simulation from running, but locating the specific error in a large YAML file can be time-consuming and frustrating. Researchers and practitioners lose valuable time debugging configuration issues instead of focusing on their simulation models and analysis.

As the complexity of the simulated system increases, YAML configuration files become increasingly difficult to read, understand, and maintain. The hierarchical structure of YAML, while useful for simple configurations, becomes unwieldy when dealing with simulations that involve:

- Hundreds or thousands of nodes with different initial states
- Multiple reaction types with complex conditions and actions
- Nested dependencies between variables and parameters
- Multiple deployment strategies and network models
- Complex batch simulation scenarios with interdependent variables

The indentation-based syntax of YAML, while visually structured, becomes error-prone as nesting levels increase. A misplaced space or incorrect indentation can completely alter the configuration's meaning, leading to subtle bugs that are difficult to detect. Moreover, YAML's lack of explicit type information means that developers must rely on documentation or trial-and-error to understand what types of values are expected for each configuration parameter.

When simulating complex systems, configuration files can grow to hundreds or even thousands of lines. In such scenarios, navigating the YAML structure, understanding relationships between different sections, and making modifications becomes increasingly challenging. The absence of IDE support for YAML means that developers cannot leverage features such as autocomplete, type checking, or inline documentation, further complicating the configuration process and increasing the learning curve for new users.

YAML's lack of type safety means that configuration errors related to incorrect types are only discovered at runtime. For example, a parameter that expects a numeric value might receive a string, or a list might be provided where a single value is expected. These type mismatches are not detected until the configuration is parsed and used, potentially after significant development time has been invested.

The absence of compile-time validation also means that refactoring configurations is risky. Renaming a molecule or changing a parameter structure requires manual verification that all references have been updated correctly. In complex simulations with many interdependent components, this manual process is error-prone and time-consuming. The lack of type checking also makes it difficult to ensure consistency across large configuration files, where the same concepts may be referenced in multiple places.

These limitations directly impact the productivity and reliability of Alchemist users. Researchers working on complex simulations spend significant time debugging configuration errors that could be caught at compile-time. The lack of IDE support increases the learning curve and reduces productivity, particularly for users new to the framework. As Alchemist continues to be adopted for increasingly complex scenarios, these limitations become more pronounced and hinder the framework's effectiveness.

The development of a type-safe Domain-Specific Language addresses these limitations by providing compile-time error detection, IDE support with autocomplete and inline documentation, and improved maintainability through explicit typing and better tooling. By moving from YAML to a programming language-based configuration approach, we enable early detection of configuration errors, improve code maintainability, and provide better tooling support for complex simulation scenarios. This work is motivated by the need to improve the developer experience and productivity when configuring Alchemist simulations, particularly as the framework is used for increasingly complex and sophisticated scenarios.

The DSL maintains semantic equivalence with the existing YAML configuration system, ensuring that simulations configured via the DSL behave identically to their YAML counterparts. This compatibility allows for gradual migration and preserves the ability to use YAML configurations when appropriate, while providing a more robust alternative for complex simulation scenarios.

1.2.2 Domain-Specific Languages

1.2.3 Kotlin DSLs

1.2.4 Kotlin Symbol Processing

Chapter 2

Analysis

This chapter analyzes the problem of developing a **type-safe** Domain-Specific Language for Alchemist simulations configuration. The analysis establishes the foundation for the design and implementation phases by identifying requirements, constraints, and the domain model underlying the simulation configuration.

The chapter begins by defining the high-level goals for the DSL. It then examines constraints that the solution must satisfy, including backward compatibility with the existing configuration system and limitations imposed by the **Kotlin** language. Moreover, the requirements are defined and analyzed in more detail, including functional and non-functional requirements.

The analysis then investigates the limitations of the current YAML-based approach and examines how the existing type system constrains the solution. Finally, the chapter models the simulation configuration domain, identifying key concepts and their relationships that guide the DSL design.

2.1 High-level Goals

The **primary goal** is to develop a type-safe Domain-Specific Language in Kotlin that will replace YAML as the main configuration format for Alchemist simulations. The DSL should eliminate the type safety and tooling limitations of YAML while maintaining **semantic equivalence** with existing YAML configurations. The adoption of a *Programming Language* for the simulation configuration should improve the overall developer experience and productivity: the DSL can provide compile-time type and syntax checking, enabling early detection of configuration errors before simulation execution. Moreover, the *Integrated Development Environment* (IDE) support, including autocomplete and inline documentation, should reduce the learning curve and improve productivity when creating simulation configurations. The user can leverage the IDE to focus on the simulation configuration and less on the language syntax.

The **second goal** concerns compatibility and integration. The DSL must integrate seamlessly with Alchemist's existing loading infrastructure, producing identical simulation models to those generated from YAML. This ensures that simulations configured via the DSL behave identically to their YAML counterparts, preserving reproducibility and allowing gradual migration from YAML to the DSL.

The **third goal** focuses on expressiveness and maintainability. The DSL should provide a natural, domain-appropriate syntax that makes simulation configurations more readable and maintainable than their YAML equivalents. However, the DSL should have the same semantic as the YAML configuration, allowing the user to leverage their existing knowledge of YAML to configure simulations. Moreover, the DSL, should be open to extensions and should be able to support new features or new incarnations as they are developed, without the need to modify the DSL itself.

Next sections will analyze, in more detail, the constraints and functional requirements that the DSL must satisfy.

2.2 Constraints

Constraints define **limitations** and **conditions** that the DSL solution must satisfy, restricting the design space and implementation choices. These constraints arise from the need to integrate with existing systems and comply with language specifications.

Two primary constraint categories affect the DSL design:

- **Backward compatibility**: ensures that the DSL integrates with Alchemist's existing configuration system without breaking existing functionality.

- **Language constraints**: the Kotlin language syntax and the available tooling capabilities limit the implementation choices.

These two constraints will be analyzed in more detail in the next sections.

2.2.1 Backward Compatibility

The backward compatibility constraint ensures that the DSL integrates with Alchemist's existing configuration system without breaking existing functionality. This constraint is crucial to ensure that the YAML configuration files can still be used to configure simulations, even after the DSL is adopted. The Kotlin DSL should be an **alternative** to YAML, not a replacement. At the same time, the DSL should support all the existing YAML features and should be able to generate the same simulation model as the YAML configuration.

The development of the Kotlin DSL should not require any changes to the existing YAML configuration files. The same is true for the loading system of Alchemist, it should be able to load both YAML and Kotlin DSL configuration files. The actual system should be extended but not replaced.

2.2.2 Kotlin Language Constraints

Kotlin provides powerful DSL-building capabilities through receiver types and extension functions, enabling concise syntax for **builder** patterns. However, compared to languages like Scala, Kotlin imposes several constraints that affect DSL design and expressiveness.

Function naming restrictions limit DSL expressiveness. Kotlin requires function names to be valid identifiers, preventing the use of arbitrary symbols or special characters. While Kotlin supports infix functions and operator overloading, the set of overloadable operators is fixed and limited. This restricts the ability to create domain-specific operators that might make configurations more intuitive.

Kotlin 1.6.20 introduced experimental context parameters as a mechanism for context-dependent declarations. However, this feature remains experimental and has some limitations: context parameters cannot be used at class level or in constructors, restricting their applicability to property and function declarations only.

2.3 Functional Requirements

Functional requirements specify what the DSL must accomplish to achieve the high-level goals while satisfying the identified constraints. These requirements define the essential capabilities and behaviors that the DSL implementation must provide. Three primary functional requirements govern the DSL design:

- **YAML equivalence**
- **Loading system integration**
- **Type safety**

These three functional requirements will be analyzed in more detail in the next sections.

2.3.1 YAML Equivalence

The YAML equivalence functional requirement ensures that the DSL can express all simulation configurations currently representable in YAML and produces semantically identical simulation models. Since it is assumed that the current YAML configuration system is working correctly, it can be used as a reference to ensure that the DSL can produce the same simulation model. The Alchemist YAML specification can be used as a reference to define the semantic of the DSL, since we want the DSL to have the same semantic as the YAML configuration. The user should be, in theory, able to convert an existing YAML configuration file to a DSL configuration file and vice versa, and the two configurations should produce the same simulation model. Moreover, a user should expect to find in the DSL configuration system all the features of the YAML configuration system.

2.3.2 Loading System Requirements

The loading system requirements mandate integration with Alchemist's existing configuration loading infrastructure, enabling the DSL to work alongside YAML configurations. The loading system should be able to load both YAML and Kotlin DSL configuration files. Currently, the loading system is able to load the simulation model from the YAML configuration file that is a file with a *.yaml* extension. The loading system should be extended to be able to support also the new Kotlin DSL configuration file that is a file with a *.kts* extension.

- **Via Gradle**
- **Via Standalone Application**

Now the Alchemist current loading system will be analyzed in more detail, to understand how it works and how it can be extended to support the new Kotlin DSL configuration file.

Existing Loading System

The Alchemist loading system operates through a two-phase architecture that separates configuration template preparation from actual simulation instantiation. This design enables batch execution scenarios where multiple simulations must be generated from a single configuration template with varying parameter values.

The loading process begins when a configuration file is parsed into an internal map-based representation. The system then enters the **Loader creation phase**, during which it analyzes the configuration structure and prepares a reusable template. This phase performs several critical operations:

- **Variable identification**: The system traverses the configuration to identify three distinct categories of variables:
  - **Constants**: Variables whose values can be immediately evaluated without dependencies on other variables. These are computed during loader creation and remain fixed across all simulation instances.
  - **Dependent variables**: Variables whose values can be computed once the values of free variables are known. These are registered during loader creation but evaluated later during simulation instantiation.
  - **Free variables**: Variables that require explicit user-provided values. These variables define the parameter space for batch simulations and must be instantiated with ground values before simulation construction.

- **Dependency resolution**: The system resolves dependencies between variables, ensuring that constants are evaluated first, followed by dependent variables that can be computed from constants. This process iterates until no further constants can be identified, establishing a partial ordering of variable dependencies.

- **Launcher configuration**: The system loads and instantiates the launcher component, which manages simulation execution lifecycle. This configuration is independent of individual simulation parameters and remains constant across batch runs.

- **Remote dependencies loading**: External dependencies required for the configuration are identified and prepared, enabling the system to fetch and incorporate external resources when needed.

Upon completion of the loader creation phase, a `Loader` object is produced that encapsulates the configuration template along with metadata about variables, constants, and execution parameters. This loader serves as a factory for generating simulation instances.

The **simulation instantiation phase** occurs when `getWith()` is invoked with a map of variable values. This phase transforms the configuration template into an executable simulation model through the following steps:

- **Variable reification**: Free variables receive their ground values from the provided map, with defaults used for unspecified variables. The system validates that all provided variable names correspond to known free variables.

- **Dependent variable computation**: Once all free variables have ground values, dependent variables are evaluated using their registered computation functions. The system ensures that all dependencies are satisfied before evaluation.

- **Value injection**: All computed values (constants, free variables, and dependent variables) are injected into the configuration map, replacing variable placeholders with concrete values throughout the configuration structure.

- **Simulation model construction**: The system constructs the complete simulation model, including:
  - Environment instantiation with specified dimensions and properties
  - Node creation and deployment according to deployment descriptors
  - Molecule and concentration assignment to nodes
  - Reaction and program attachment to nodes
  - Layer configuration for spatial data structures
  - Linking rule establishment for network topology
  - Monitor and exporter setup for data collection
  - Simulation engine initialization

This two-phase architecture provides several advantages. First, it enables efficient batch execution: a single loader can generate multiple simulation instances with different parameter combinations without re-parsing the configuration. Second, it supports parameter space exploration by allowing systematic variation of free variables across simulation runs. Third, it separates concerns between configuration management (loader phase) and simulation construction (instantiation phase), improving maintainability and extensibility.

The distinction between constants, dependent variables, and free variables is fundamental to this architecture. Constants represent configuration elements that remain invariant across all simulation instances, such as physical constants or fixed simulation parameters. Dependent variables enable derived parameters that depend on free variables, allowing configurations to express relationships between parameters without requiring explicit enumeration of all combinations. Free variables define the parameter space that can be varied during batch execution, enabling systematic exploration of simulation behavior across different configurations.

2.3.3 Type Safety

Type safety represents a fundamental requirement for the DSL, addressing one of the most significant limitations of the YAML-based configuration approach. Unlike

YAML, where configuration errors remain undetected until runtime, the DSL must leverage Kotlin's type system to identify and report errors during development, before the simulation is executed.

The primary advantage of type safety lies in the immediate feedback it provides to developers. Modern IDEs can perform continuous type checking as the user writes code, highlighting errors in real-time through visual indicators such as red underlines and error messages. This immediate feedback transforms the development experience: instead of discovering a typo in a molecule name or an incorrect parameter type only after running the simulation, developers receive instant notification that something is wrong. For instance, if a user attempts to assign a string value to a parameter that expects a numeric type, the IDE will immediately flag this as an error, preventing the mistake from propagating further into the development process.

Beyond error detection, type safety enforces a stricter interpretation of the configuration specification. The DSL's type system acts as a formal contract that constrains what constitutes valid syntax and structure. This enforcement mechanism guides developers toward correct configurations by making invalid constructs impossible to express within the type system. This contrasts sharply with YAML, where such structural requirements are only enforced through runtime validation.

Finally, the compile-time validation provided by type safety contributes to improved confidence in configuration correctness. Developers can trust that if their code compiles successfully, the configuration adheres to the type constraints, eliminating entire classes of syntax errors. This feature should reduce the amount of time spent on fixing typos and other common errors in the configuration file.

2.4 Non-functional Requirements

While functional requirements define *what* the DSL must accomplish, non-functional requirements specify *how well* the system should perform and what quality attributes it should exhibit.

These requirements address the qualitative aspects of the DSL that determine its effectiveness, maintainability, and long-term viability. Non-functional requirements typically involve subjective assessments of quality, usability, and performance characteristics.

For a Domain-Specific Language, non-functional requirements play a crucial role in determining its adoption and success. A DSL that fulfills all functional requirements but is difficult to use, lacks extensibility, or performs poorly will likely fail to gain traction among developers. These requirements ensure that the DSL not only works correctly but also provides a positive user experience and can evolve to meet future needs.

The next sections will analyze in more detail the non-functional requirements.

2.4.1 DSL API Usability

2.4.2 Extensibility

2.4.3 Performance

2.5 Requirements Analysis

2.6 Problem Analysis

2.6.1 Limitations of YAML Configuration

2.6.2 Legacy Type System Limitations

2.7 Simulation Configuration Domain

Chapter 3

Design

3.1 Architecture

3.1.1 Scoped Information Contexts

3.1.2 Loading System

3.2 Detailed Design

3.2.1 Task-Based System

3.2.2 DSL Builder Structure

Chapter 4

Implementation

4.1 DSL Implementation

4.1.1 Builder Functions

4.1.2 Variables and Constants

4.1.3 Batch Mode Support

4.2 KSP Implementation

4.2.1 Symbol Processing Pipeline

4.2.2 Code Generation Strategy

4.2.3 Context Injection Mechanism

Chapter 5

Evaluation

5.1 Testing

5.1.1 Unit Testing

5.1.2 Integration Testing

5.1.3 DSL Equivalence Testing

5.1.4 Performance Testing

5.2 Experimental Evaluation

5.2.1 DSL Usability

5.2.2 KSP Processor Benefits

Chapter 6

Conclusion and Future Work

Bibliography

Acknowledgements

Optional. Max 1 page.