ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

Design and Implementation of a Prototype Open Benchmarking Platform for Collective Adaptive Systems

Tesi di laurea in:
Laboratorio di Sistemi Software

Relatore
Prof. Danilo Pianini

Candidato
Paolo Penazzi

Correlatore
Prof. Lukas Esterle

IV Sessione di Laurea
Anno Accademico 2022-2023

Abstract

In every domain of scientific research, the comparison between innovative solutions and the state of the art is crucial. This practice enables the evaluation of whether the system under examination outperforms the established reference, either comprehensively or in specific aspects. In various fields of computer science, tools have been developed to benchmark new and existing solutions. On the other hand, in the domain of collective adaptive systems, a conspicuous gap exists in software designed to facilitate such comparisons.

The primary objective of this thesis is to create a prototype for a benchmarking platform focused on Collective Adaptive Systems (CAS). By making use of existing simulators available in the market, the aim is to establish a comprehensive framework for testing, validating, and comparing these dynamic systems. The presented platform is designed to allow users to define benchmarks, execute them, and extract results of interest - all while preserving flexibility and extensibility. This inherent adaptability allows for the incorporation of additional simulators into the testbed.

An experiment has been executed to validate the framework's anticipated functionalities and understand its strengths and weaknesses. This analysis serves the purpose of identifying areas for future improvement within the tool.

Contents

Abstract **iii**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Background . . . 2
1.2.1 Collective Adaptive Systems . . . 2
1.2.2 Testing and Simulation . . . 4
1.2.3 Alchemist . . . 5
1.2.4 NetLogo . . . 8
1.3 Objectives . . . 9

2 Design **11**
2.1 Domain Analysis . . . 11
2.1.1 Ubiquitous Language . . . 11
2.1.2 User Stories . . . 12
2.2 Requirements . . . 13
2.2.1 User Requirements . . . 13
2.2.2 Functional Requirements . . . 13
2.2.3 Non-Functional Requirements . . . 14
2.3 Architecture . . . 14
2.3.1 Benchmark Configuration . . . 18
2.3.2 Benchmark Results . . . 20
2.4 Extension . . . 22
2.5 Simulators . . . 23

3 Implementation **25**
3.1 Framework . . . 25
3.2 Technologies . . . 32
3.2.1 Framework technologies . . . 32
3.2.2 DevOps technologies . . . 33

4 Validation **39**
4.1 Testing 39
4.1.1 Unit Testing 39
4.1.2 Integration Testing 40
4.2 Evaluation 42

5 Conclusion and Future Work **47**

51

Bibliography **51**

List of Figures

1.1 Some examples of CAS. 3
1.2 Alchemist meta-model. 6
1.3 Alchemist reaction. 6
1.4 A grid of nodes in Alchemist. 7
1.5 NetLogo interface. 8

2.1 Abstract architecture of the testbed. 15
2.2 Abstract execution of a benchmark. 15
2.3 Detailed architecture of the system. 16
2.4 Detailed benchmark execution. 17
2.5 Simulator's output processing. 21

3.1 Benchmark Model. 26
3.2 Git Flow. 33

4.1 NetLogo simulation launched by the framework. 41
4.2 Alchemist simulation launched by the framework. 42
4.3 Case of study: Benchmark result A. 45
4.4 Case of study: Benchmark result B. 45

List of Listings

2.1 Benchmark configuration file structure: Strategy section. 18
2.2 Benchmark configuration file structure: NetLogo simulator section. 19
2.3 Benchmark configuration file structure: Alchemist simulator section. 20
3.1 Benchmark model. 25
3.2 SupportedSimulator enum. 26
3.3 Parsing of the input file. 27
3.4 ConfigFileHandler interface. 27
3.5 Listener interface. 28
3.6 CSV file cleaning in Alchemist. 28
3.7 ScenarioOutput implementation. 29
3.8 BenchmarkOutput implementation. 29
3.9 BenchmarkResult and ScenarioResult implementation. 29
3.10 Controller implementation. 30
3.11 Controller: createExecutor method. 30
3.12 Executor interface. 31
3.13 NetlogoExecutor class. 31
3.14 Custom task to generate the JAR file. 34
3.15 CI/CD workflow: build job. 35
3.16 CI/CD workflow: release job. 35
4.1 Parser tests. 39
4.2 Case of study: benchmark configuration file. 42
4.3 Case of study: Alchemist input file. 43
4.4 Case of study: gradient program A. 43
4.5 Case of study: output processing function. 44
4.6 Case of study: gradient program B. 45

Chapter 1

Introduction

1.1 Motivation

In scientific research, new solutions are developed in order to improve what is defined as the state of the art. This implies that, at some point, two different solutions must be compared to define which one performs better, either overall or in specific aspects. The result of this comparison must be a clear and objective metric, that does not leave room for personal interpretation. Additionally, it is essential to use a widely adopted, standard test protocol that executes two different algorithms on the same problem, under the same conditions, with the possibility of reproducing experiments. The lack of such a protocol raises concerns regarding the reproducibility of scientific publications, as happened in some fields of Computer Science [13, 19]. The creation of a benchmarking and testing framework can solve these problems, by setting a standard for the comparison of different solutions.

In some application domains, such as security [17], IoT [15], and many more [14, 32], tools have been developed for benchmarking. In other fields, like Autonomic, Organic Computing, and Collective Adaptive Systems (CAS), such frameworks have not been created yet [8, 18]. This absence led us to focus on Collective Adaptive Systems (CAS) [24].

1.2 Background

1.2.1 Collective Adaptive Systems

Collective Adaptive Systems (CAS) are a complex type of distributed network which are composed of many heterogeneous entities, each with its own capabilities and goals. These systems are characterized by the ability to adapt their behavior to dynamically changing open-ended environments [9] and by the pursuit of a collective goal. The latter is achieved through the collaboration of the systems' entities, without a specific external or internal central control [20, 21]. In fact, CAS often adopts cooperative operating strategies to run distributed decision-making mechanisms [2]. The properties and behavior of these systems make them particularly challenging to test and evaluate their performance [3].

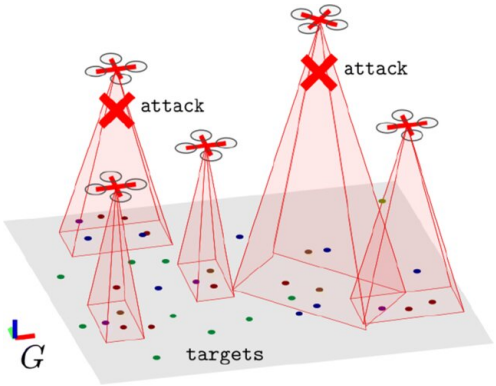
Nowadays, many systems are adaptive and collective. Examples include drone swarms tasked with monitoring an area (as Figure 1.1a shows), wearable devices to manage crowd congestion during a public event (represented in Figure 1.1b), cars on streets connected to handle traffic [24].

Figure 1.1b is provided by [1].

CAS programming The spread of these systems has led to a shift in computation, which is now divided and distributed across various devices in the network, introducing an additional level of complexity in programming these systems. Developers must consider issues such as communication between devices, concurrency, or failures. Furthermore, as these systems grow in complexity, it becomes challenging to create solutions that are extensible, modular, and easily testable [10]. Several approaches have been identified over the years for programming CAS, or more generally, adaptive distributed systems.

Agent-based models [22] associate each device with the behavior of an agent [29], which has sensors and actuators to interact with the environment and the ability to communicate with other agents. This is the paradigm used in the NetLogo simulator.

[1] https://www.unibo.it/it/studiare/dottorati-master-specializzazioni-e-altra-formazione/insegnamenti/insegnamento/2023/483706



(a) Drone swarm.

(b) Crowd management.

Figure 1.1: Some examples of CAS.

The SCEL language [25], the first to use the paradigm defined as attribute-based programming, establishes a formal approach to the interaction between devices. This paradigm defines the system as a series of devices, each with a set of attributes representing the properties of the component.

Aggregate programming is a new approach to developing complex distributed systems that abstract from individual devices, focusing on programming the collective. Through a layer that handles and hides some problematic aspects of these networks, such as communication between devices and details of individual entities, it is possible to simplify the design and maintenance of these systems [6, 7]. Aggregate programming is based on the concept of a computational field, which is a global map associating each device in the network with its local value, and on that of field calculus, a minimal core that provides basic constructs for working with fields [34]. Other notable approaches exist, including SOTA [1] and TOTA [23].

1.2.2 Testing and Simulation

In every field of engineering, testing is a fundamental part of the development process. It refers to the process carried out to verify and validate a system, according to its requirements [30]. Testing is important to evaluate the behavior of newly developed algorithms against state-of-the-art solutions. This allows us to understand whether a newly developed solution is better than an existing one in a certain scenario.

Adaptive systems testing introduces a series of challenges and difficulties, many of which stem from the intrinsic nature of these systems. Given their complexity, the use of a single simulator is not sufficient to test all their features. In fact, it is often necessary to employ multiple simulators and combine the results obtained to understand the system's behavior. This technique is termed co-simulation and introduces various issues, such as communication delays, approximations, and difficulties in synchronizing simulators [31]. Since these systems are adaptive and react to mutations in their environment, it is natural to want to support the injection of changes [8]. Therefore, a complete testbed must be able to command different existing tools, support various execution environments, and allow the user to test the system in its entirety.

In Computer Science, the simulation is the process of executing software in a controlled environment to evaluate its behavior. Simulations can be used to test the correctness of a program, evaluate its performance, or understand its behavior in a specific scenario. The key point of a simulation is to execute the software under controlled and repeatable conditions to compare different executions [13]. This cannot be done without a simulator, which provides the user with all the tools needed to run the simulation [4, 5]. The importance of simulators becomes clear when testing CAS. Creating a real environment to test a program, such as a network of 100 drones or a crowd of 1000 people, is not feasible. Simulators allow us to deploy a virtual environment where we can run the program and evaluate its behavior.

Within the domain of CAS, numerous simulators provide an environment for testing these systems, either in some aspects or in their entirety. This is the case for

ns-3 [2], a discrete-event network simulator, the Repast suite [26], a family of agent-base modeling platforms, and OMNeT++ [3], a modular, component-base C++ simulation library. Given their central role in this study, the subsequent sections will extensively examine other simulators, specifically Alchemist and NetLogo.

1.2.3 Alchemist

Alchemist [27] is an open-source tool for simulating complex distributed systems. It is termed a meta-simulator because it is based on generic abstractions. The meta-model of Alchemist, as Figure 1.2 and Figure 1.3 show, is inspired by biochemistry and consists of various entities [4]:

- **Molecule** The name of a data item.
- **Concentration** The value associated with a molecule.
- **Node** A container of molecules and concentrations. Disposed inside the environment.
- **Environment** The abstraction for the space. It contains nodes and can tell the position of a node in the space and the distance between two nodes.
- **Linking Rule** A rule that defines the relation between nodes.
- **Reaction** Events fired according to a time distribution and set of conditions.
- **Condition** A function that takes the current environment as input and outputs a boolean and a number. The output influences the execution of the corresponding reaction.
- **Action** Models a change in the environment.

The key to the Alchemist's extensibility is the very generic interpretation of molecules and concentrations. An incarnation maps this generic chemical abstraction to a specific use case. Alchemist supports four incarnations: SAPERE [12],

[2] https://www.nsnam.org/
[3] https://omnetpp.org/
[4] https://alchemistsimulator.github.io/

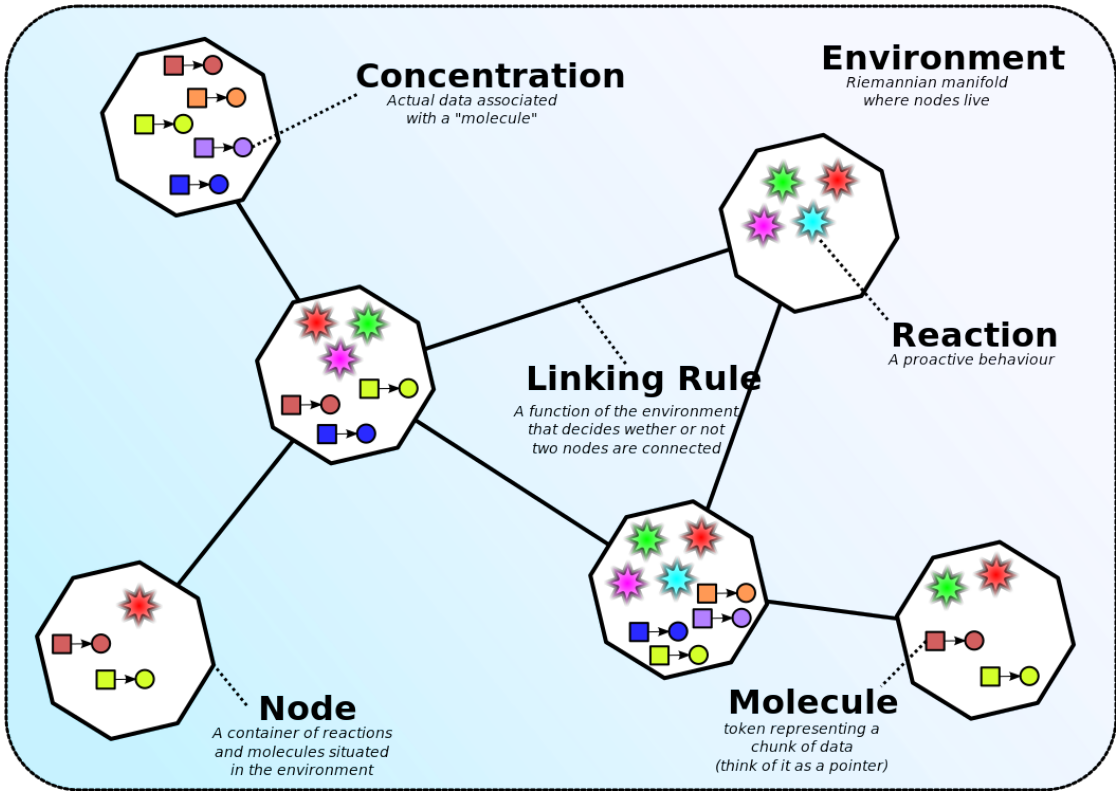


Figure 1.2: Alchemist meta-model.

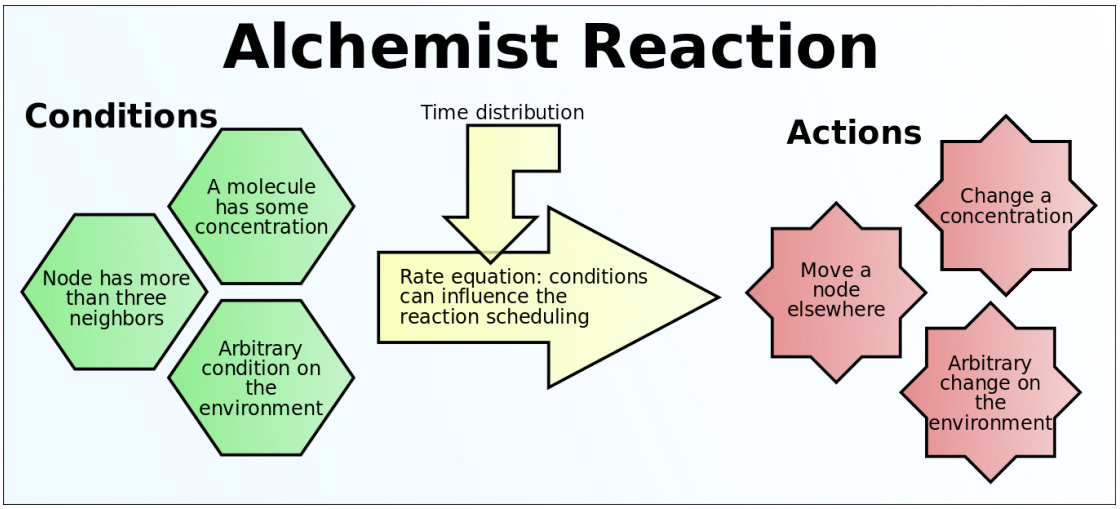


Figure 1.3: Alchemist reaction.

the first supported incarnation, based on the concept of Live Semantic Annotation

(LSA), ScaFi [11], which is a Scala-based library and framework for Aggregate Programming, Protelis [28], a programming language for aggregate computing, and Biochemistry incarnation. Figure 1.4 shows some nodes in the Alchemist simulator interface.

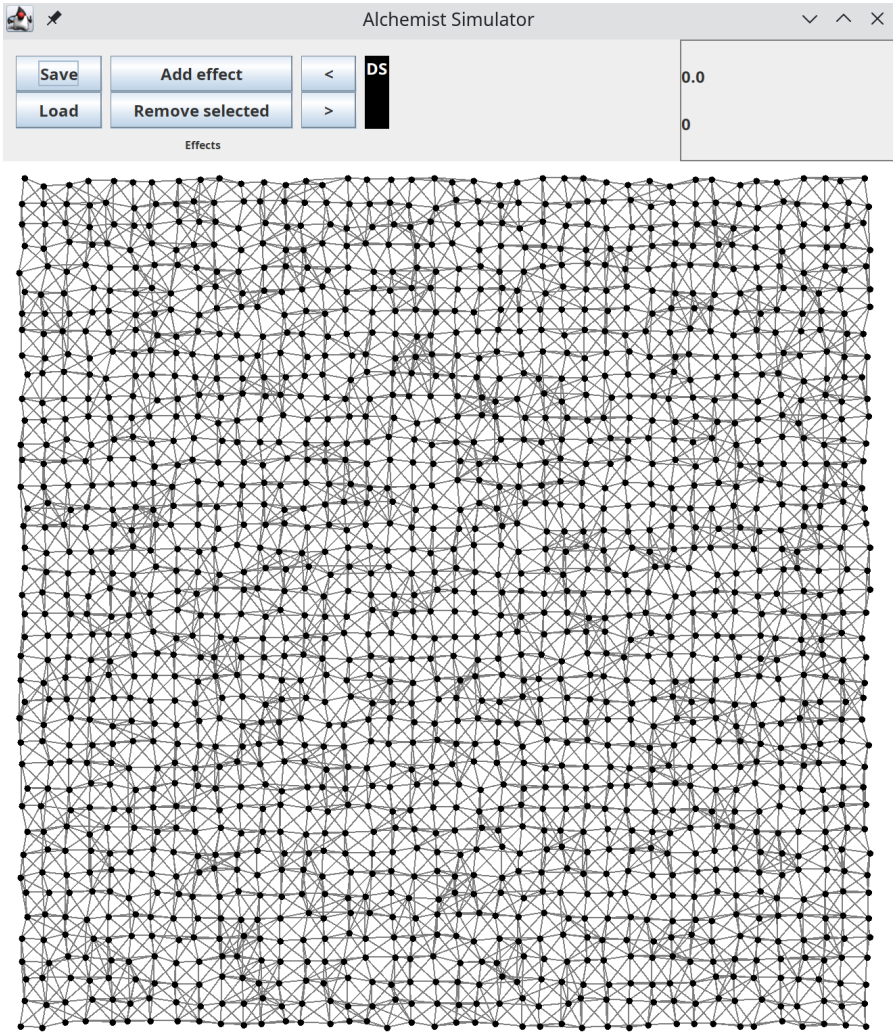


Figure 1.4: A grid of nodes in Alchemist.

1.2.4 NetLogo

NetLogo[5] is a programmable modeling environment for simulating natural and social phenomena. It was created at the Center for Connected Learning and Computer-Based Modeling (CCL) at Northwestern University, directed by Uri Wilensky.

NetLogo is particularly well suited for modeling complex systems developing over time. Users can program the behavior of thousands of independent agents to see how the system-level behavior emerges from the interactions of the agents.

It also comes with the Models Library, a large collection of pre-written simulations that can be used and modified. These simulations can be explored to observe their behavior under various conditions.

An example of a NetLogo simulation is depicted in Figure 1.5. It uses the Fire model which simulates the spread of a fire through a forest. It shows that the fire's chance of reaching the right edge of the forest depends critically on the density of trees.

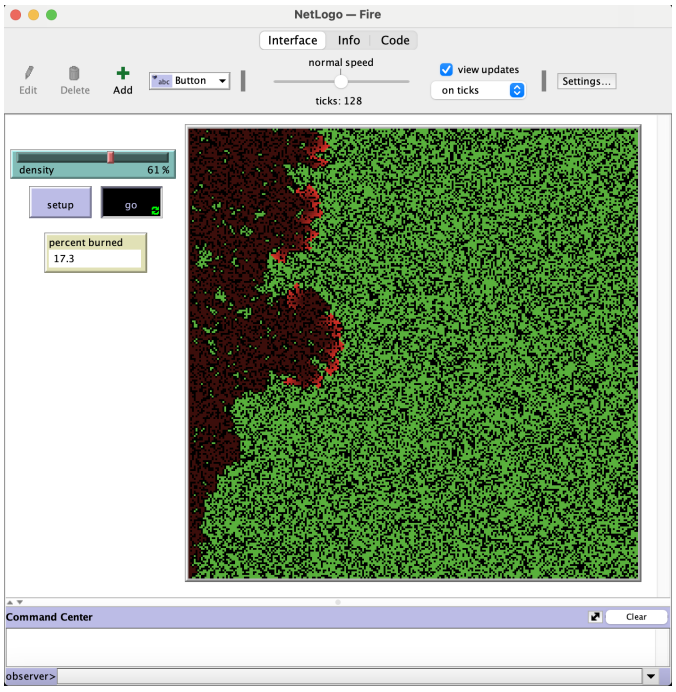


Figure 1.5: NetLogo interface.

[5]https://ccl.northwestern.edu/netlogo/

1.3 Objectives

This thesis aims to partially address the absence of a benchmarking platform by developing a prototype [33, 35]. The testbed must be designed to allow the user to define a benchmark, execute it, and compare the result to state-of-the-art solutions. To assess the system's behavior, the framework relies on existing simulators in the market. Given the large number of these tools, the testbed must be flexible and extensible to include support for new simulators later on without compromising the functionality of those already integrated or the framework as a whole [16]. Recognizing that users may focus on various aspects of CAS, it is essential to establish generic and customizable metrics for their evaluation. With this work, the aim is to provide the foundations for a standard way to test and compare various solutions in the CAS domain.

Thesis structure In the discussion of this work, we will start with the design of the framework, as discussed in Chapter 2. In particular, we conduct a domain analysis, define the requirements, and then provide a detailed description of the system architecture. Chapter 3 delves into the implementation details, describing how various components of the framework were developed and the technologies employed. Chapter 4 outlines the testing of the platform, verification of integration with supported simulators, and presents a case study to validate adherence to the requirements defined in the analysis phase. Finally, in Chapter 5, we analyze the work done and explore potential future developments.

Chapter 2

Design

2.1 Domain Analysis

A domain-driven approach was employed in the development of this work.

2.1.1 Ubiquitous Language

To better understand the problem domain and to avoid confusion, a ubiquitous language was defined. These concepts were then utilized in the framework development and can be found in the work.

Term	**Meaning**
Testing	The overall process carried out to verify and validate a system, according to requirements, to promote the desired internal and external quality and to mitigate risks in development and products.
Testbed	A platform for rigorous, transparent and replicable environment for experimentation and testing
Solution	A set of algorithms leading to achieving goals and overcoming the problem posted
Scenario	Contains all the information about the test execution: the simulation platform, the metrics, the input parameters
Simulator	A software that allows the user to see how its program would behave in a real environment

Table 2.1: Domain Ubiquitous Language

2.1.2 User Stories

User Stories were also defined to clarify the users' needs and thus what features the framework should support.

User Story
...As a user, I want to be able to create a benchmark.
...As a user, I want to be able to use different simulators.
...As a user, I want to be able to define and execute different scenarios.
...As a user, I want to be able to define a solution.
...As a user, I want to be able to define how the output of the benchmark will be processed.
...As a user, I want to be able to compare my solution to others.

Table 2.2: Domain Ubiquitous Language

It is worth noting that the expected users of the framework are researchers and developers, i.e., people with a strong technical background and knowledge of the domain.

2.2 Requirements

2.2.1 User Requirements

User requirements express the needs of the user and identify which actions the user should be able to perform. The following requirements are extracted from the previous domain analysis:

- It should be possible to define a *benchmark*.
- It should be possible to define a *scenario*.
- It should be possible to apply a *solution* to an existing scenario.
- It should be possible to download and use different *simulators*.
- It should be possible to execute a benchmark.
- It should be possible to define which *metric* to extract from the benchmark's output.
- It should be possible to compare the results of different solutions.
- It should be possible to extend the framework to support new simulators.

2.2.2 Functional Requirements

Functional requirements define the features and the functions of the framework. These derive from the user requirements.

- The framework should allow the user to define a benchmark.
- The framework should allow the user to define a scenario.
- The framework should allow the user to run a scenario with any solution.

- The framework should allow the user to use different simulators, providing a way to download them.
- The framework should allow the user to execute a benchmark.
- The framework should allow the user to define which metric to extract from the benchmark's output.
- The framework should allow the user to compare the results of different solutions.
- The framework should allow the user to add support for new simulators.

2.2.3 Non-Functional Requirements

Non-functional requirements define the quality attributes of the framework.

- The framework should facilitate the user in testing collective adaptive systems.
- The framework should not limit the user in any way, to the extent that specific simulators permit.
- The framework must provide an easy and clean way to define a benchmark and all its components.
- The framework must provide an easy and clean API to add support for new simulators.

2.3 Architecture

The testbed is a framework that sits between the user and the various simulators. The user specifies which benchmark to run. The execution of the benchmark is then handled by the testbed, which takes care of running the various scenarios in the respective simulators and collecting the results.

Figure 2.1 depicts the architecture of the testbed at the highest level.

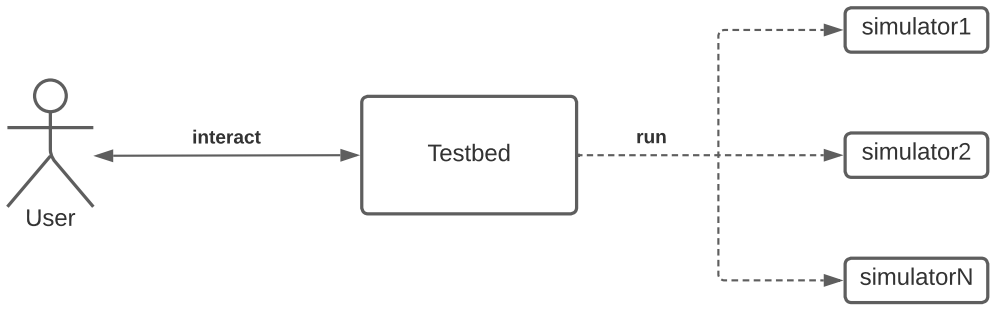


Figure 2.1: Abstract architecture of the testbed.

The typical user interaction with the testbed is shown in Figure 2.2. The user starts the testbed, which executes all the scenarios specified in the benchmark configuration file, and then displays the results to the user.

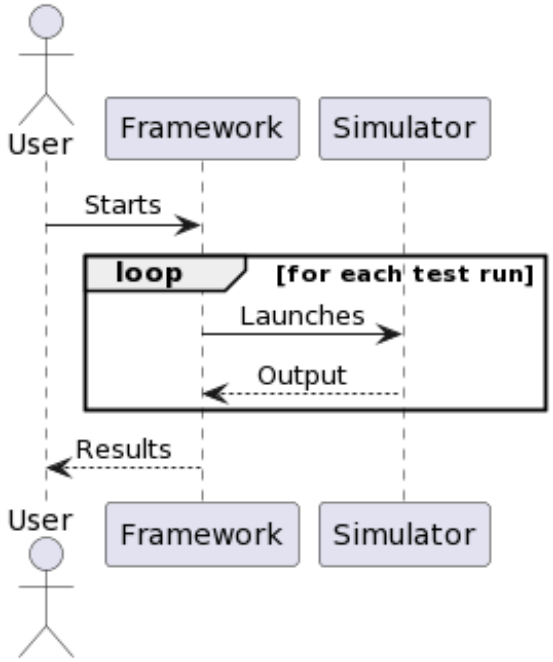


Figure 2.2: Abstract execution of a benchmark.

Diving deeper into the architecture, we can see that the testbed consists of different components, each with a specific role. Figure 2.3 is a more detailed

image of the system's architecture.

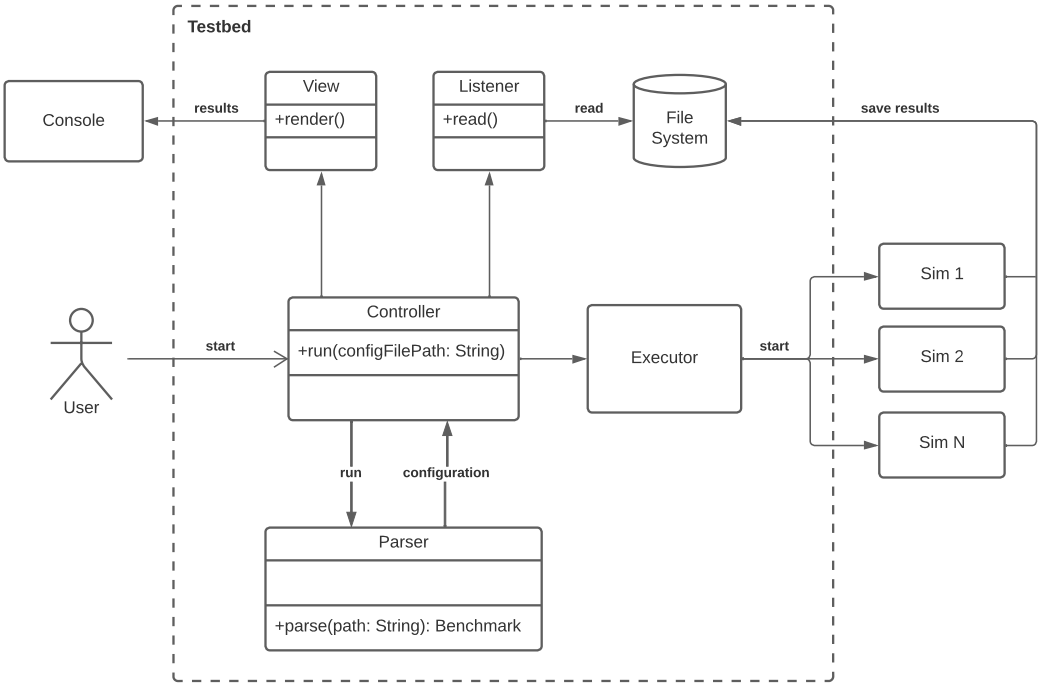


Figure 2.3: Detailed architecture of the system.

The main component of the Testbed is the controller, which is the entry point of the framework and handles the entire benchmark execution. The Parser is the component responsible for translating user-written specifications in YAML into a data structure that represents the benchmark model. If there are manipulations to be made on a configuration file, they will be performed by a Parser component, specific to each simulator, before the actual parsing. The Executor is responsible for starting the simulator and generating the correct command to invoke the simulator. For each started simulator, the corresponding Listener is then launched. The latter reads the simulator's output, cleans the file from unnecessary elements, and saves it in a data structure. Once the benchmark execution is complete, a post-processing function is applied to the benchmark output to obtain a result of interest, and it is displayed to the user by the View.

To better understand the testbed's functioning, it is useful to analyze the exe-

cution of a benchmark, which is depicted in Figure 2.4.

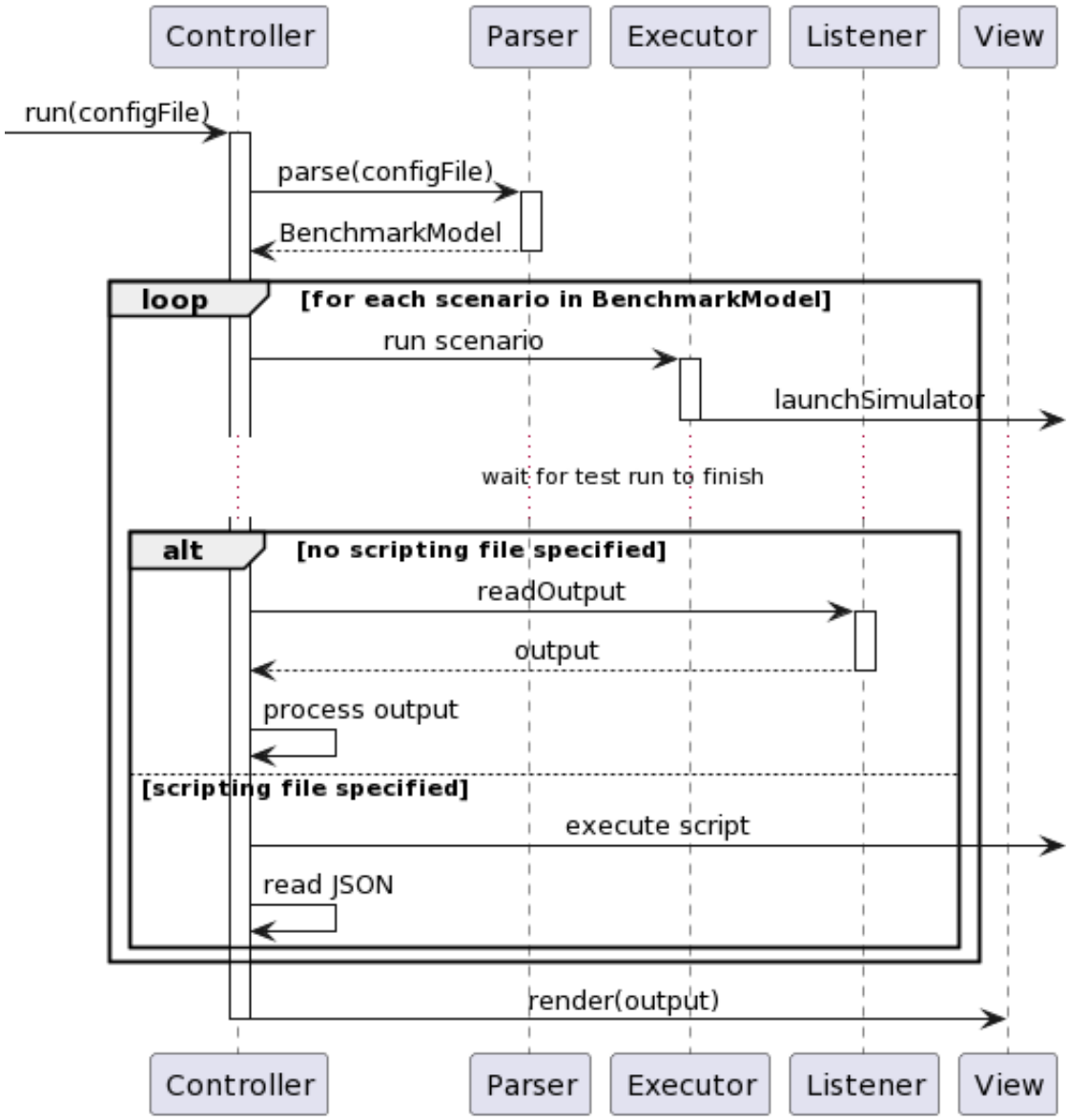


Figure 2.4: Detailed benchmark execution.

The configuration file is given as input to the controller, which performs some checks on the file's integrity. Once passed, the controller hands the configuration file to the parser, which, after modifying the file (if necessary), returns to the controller a data structure called BenchmarkModel. The model contains all the information about the benchmark to be executed. At this point, each scenario

defined by the user must be launched. The execution is carried out in the order specified by the user. For each scenario, a command to start the simulator is generated and launched. The framework then waits for the simulation to finish. Once it is done, the framework reads the results returned in output by the simulator and saves them in a data structure. When the data from all scenarios has been collected, a user-defined transformation is applied to extract results of interest. The latter are finally displayed to the user, either through the console or in a GUI.

2.3.1 Benchmark Configuration

The design of the input file system is a crucial aspect of the framework. It should enable the user to define a benchmark simply and intuitively, without limiting the user in any way. It also needs to be flexible enough to allow the addition of new simulators without breaking the existing structure.

The input file is composed of two main sections, namely *Strategy* and *Simulators*.

Strategy The strategy section provides generic information about the testbed configuration, rather than simulator-specific instructions. Currently, it only includes the execution order of the scenarios, a mandatory parameter that defines the sequence in which the scenarios will be executed. Additional strategy parameters could be added in the future to support other features, such as multi-threaded execution.

Listing 2.1 shows a possibile definition of the *Strategy* section in a benchmark configuration file.

Listing 2.1: Benchmark configuration file structure: Strategy section.

```
1 strategy:
2  executionOrder:
3   - Alchemist-sapere-tutorial
4   - NetLogo-tutorial
5   - Alchemist-protelis-tutorial
```

Simulators The simulators section contains the configuration of the simulators used in the benchmark. Each simulator has a name, a path, and a list of scenarios.

The name is mandatory and must be written exactly as it appears in the testbed documentation. The path is optional and is used to specify the path of the executable simulator. If not specified, the framework will assume that the simulator is in the same directory as the testbed.

Scenario The scenario configuration is more complex, as it depends on which simulator is used to run the scenario. This section contains:

- **name** the name of the scenario. This is mandatory and should match the name in the execution order list.
- **description** a brief explanation of the scenario. This is optional.
- **input** a list of all the input files needed to run the scenario. This parameter is optional to take into account a scenario that does not require any input file.
- **postProcessing** the script that will be used to process the scenario output. This parameter is optional.
- **repetitions** the number of times that the scenario should be run. This parameter is optional and defaults to 1.
- **duration** the duration of the simulation. This parameter can be used to overwrite the value present in the simulator-specific configuration file, if the simulator supports it. This parameter is optional.

Listing 2.2 shows a possible definition of the *Simulators* section in a benchmark configuration file using NetLogo as a simulator.

Listing 2.2: Benchmark configuration file structure: NetLogo simulator section.

```
simulators:
 - name: NETLOGO
  simulatorPath: "./NetLogo 6.4.0/"
  scenarios:
   - name: NetLogo-tutorial
    description: A tutorial to NetLogo
    input:
     - "../src/main/resources/netlogo/netlogo-tutorial.xml"
     - "./models/IABM Textbook/chapter 4/Wolf Sheep Simple 5.nlogo"
```

```
    repetitions: 3
    postProcessing: "./processing/netlogo-tutorial.py"
```

Listing 2.3 depicts the configuration of two scenarios run by the Alchemist simulator.

Listing 2.3: Benchmark configuration file structure: Alchemist simulator section.

```
simulators:
 - name: Alchemist
  simulatorPath: "./"
  scenarios:
   - name: Alchemist-protelis-tutorial
     description: A tutorial to Alchemist and Protelis incarnation
     input: ["src/main/resources/alchemist/protelis-tutorial.yml"]
     repetitions: 1
     duration: 10
   - name: Alchemist-sapere-tutorial
     description: A tutorial to Alchemist and Sapere incarnation
     input: ["src/main/resources/alchemist/sapere-tutorial.yml"]
     repetitions: 1
     duration: 100

```

2.3.2 Benchmark Results

A critical part of the framework design is related to what is presented to the user at the end of the benchmark execution.

As Figure 2.5 shows, each simulator has its own method of providing simulation results. Certain simulators return a CSV file, others a text file, and some even utilize snapshots. Managing all these diverse cases within the framework is not feasible, which is why two methods are provided to obtain the desired results: through an external scripting file or by implementing a listener and a processing function.

In the first case, an external scripting file is used to process the simulator's output. Once the simulation is complete, the controller invokes the script specified in the Scenario configuration. Upon execution completion, the framework reads the results from a predefined JSON file, namely *result.json*. This file must contain data in the form of *ScenarioResult*, which will be defined below.

In the second case, the Listener interface is employed. It takes the file outputted

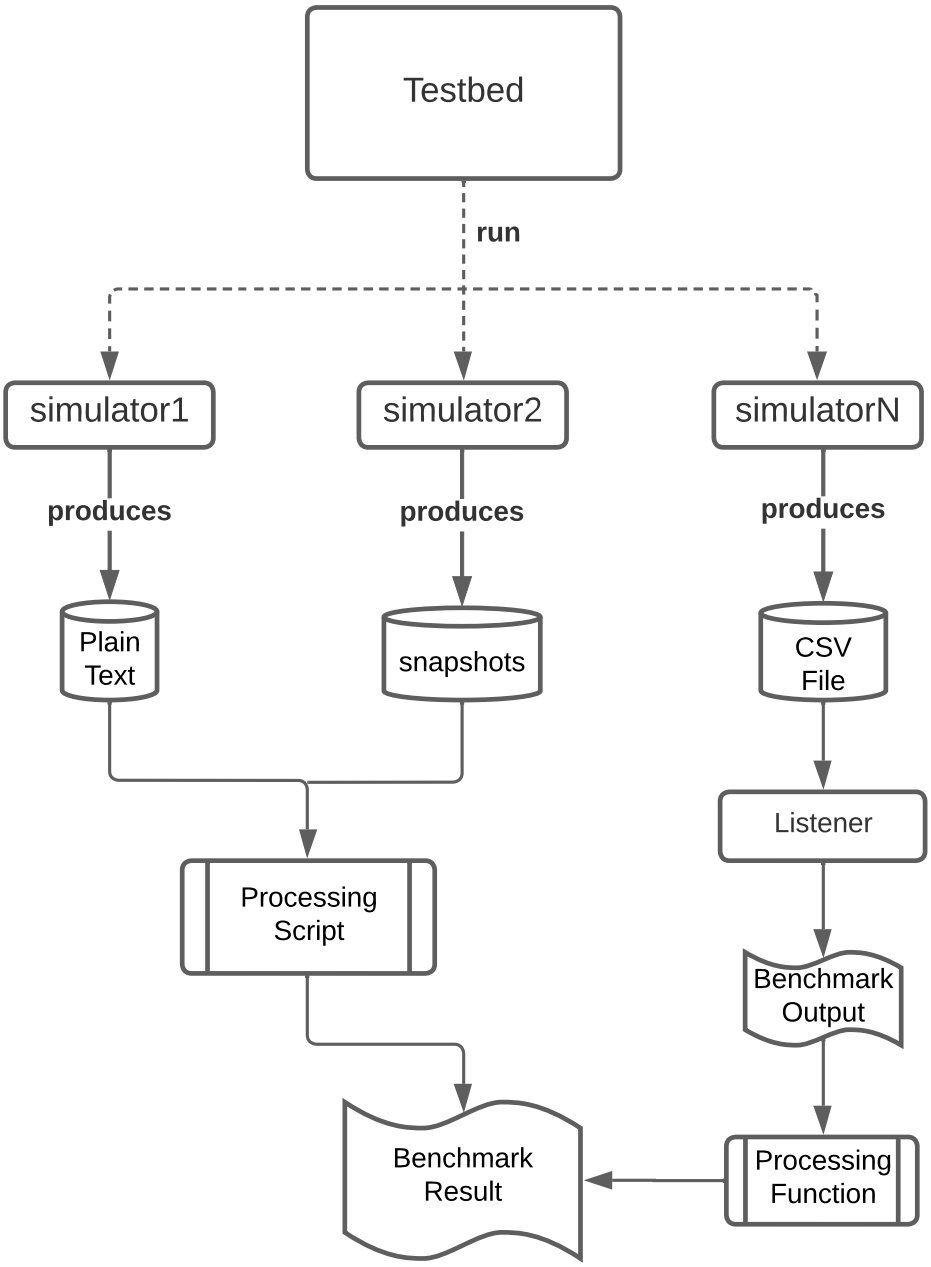


Figure 2.5: Simulator's output processing.

by the simulator and transforms it into an internal framework data structure, the *BenchmarkOutput*. A post-processing function must then be implemented, taking the *BenchmarkOutput* as input and returning a *BenchmarkResult*. This solution split the process into two phases, the read of the output simulator and the transformation of the data in a significant result. This allows the implementation of a method to read the output of a simulator and to use it in combination with various processing functions to obtain results of different natures.

We define *output* concepts:

- **Scenario Output** the output of a single scenario. It is a map that associates each metric with its value.
- **Benchmark Output** the output of the entire benchmark. It is a map that associates each scenario with its *Scenario Output*.

These concepts represent the data returned at the end of the benchmark execution as generated by the simulator. This data must be processed to extract useful information for the user.

We define *result* concepts, which represent the data the user desires, obtained by processing the benchmark's output.

- **Scenario Result** the result of a single scenario. It contains a description of the result and its value.
- **Benchmark Result** the result of the entire benchmark. It is a list of *Scenario Result*.

2.4 Extension

One of the main goals of this work is to create a flexible system that can be extended to support different simulators. The architecture was designed considering this objective. Each component of the system has a general behavior, independent of the simulator but incomplete. This will be then integrated with the specific behavior related to the simulator defined in a subclass.

Users interested in adding support for a new simulator must do the following steps:

- Implement the *Executor* interface.
- Implement the *Listener* interface. Optional.
- If some manipulations on the input file are needed, implement the *ConfigFileHandler* interface.
- Extend the *SupportedSimulator* enum by adding the new simulator.
- Update the *Controller* to take into account the new simulator.

2.5 Simulators

Since the framework relies on external simulators to run a benchmark, the users must have these simulators installed on their machines. Various solutions were evaluated to address this issue, each with its pros and cons.

- Include all supported simulators in the application: while this solution ensures that the simulators are present, it has drawbacks. Forcing the user to download all supported simulators may be impractical, especially if they only need a few of them or already have them installed. Additionally, this would significantly increase the application size.
- Let the user download the simulators: this is the simplest solution, as it shifts the responsibility of downloading the required simulators to the user. However, it degrades the user experience as users cannot immediately use the framework after downloading it.
- Hybrid solution: simulators are not included in the framework. However, if the testbed does not detect them during benchmark execution, it automatically downloads them. This solution is more complex but has proved to be the most effective after the analysis.

Chapter 3

Implementation

In this chapter, we will dive deeper into the implementation of the framework.

3.1 Framework

Benchmark Model The benchmark model is the data structure that represents the benchmark to be executed. It matches the structure of the input file to allow easy parsing, as shown in Figure 3.1.

Each concept of the model is implemented as a data class in Kotlin, which is a class that only contains data and does not have any functionality. The *Serializable* annotation is used to allow the model to be serialized and deserialized from YAML. This annotation is provided by the *"org.jetbrains.kotlinx:kotlinx-serialization-json"* library. Listing 3.1 shows the implementation of the benchmark model.

Listing 3.1: Benchmark model.

```
1  @Serializable
2  data class Benchmark(
3      val strategy: Strategy,
4      val simulators: List<Simulator>
5  )
6  @Serializable
7  data class Strategy(
8      val executionOrder: List<String> = emptyList(),
9  )
10 @Serializable
11 data class Simulator(
12     val name: SupportedSimulator,
```

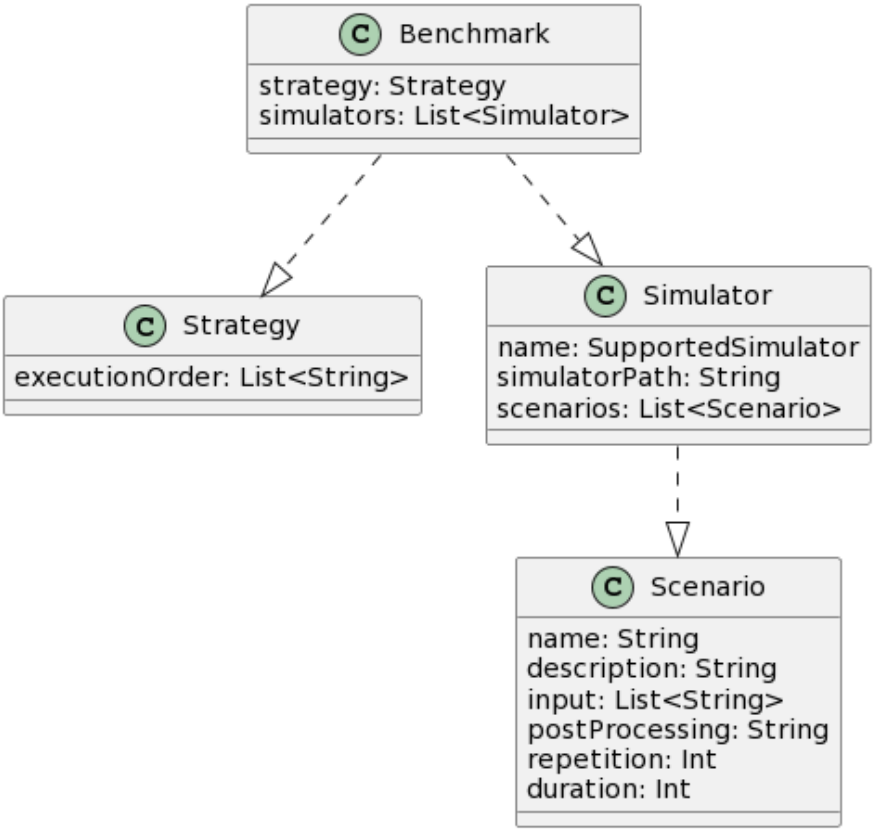


Figure 3.1: Benchmark Model.

```
13      val simulatorPath: String = "",
14      val scenarios: List<Scenario>,
15  )
16  @Serializable
17  data class Scenario(
18      val name: String,
19      val description: String = "",
20      val input: List<String> = listOf(),
21      val postProcessing: String = "",
22      val repetitions: Int = 1,
23      val duration: Int = 0,
24  )
```

To account for the possibility of adding new simulators, the *SupportedSimulator* enum was created.

Listing 3.2: SupportedSimulator enum.

```
1   enum class SupportedSimulator {
```

```
    ALCHEMIST,
    NETLOGO,
}
```

As Listing 3.2 shows, the enum contains all the simulators supported by the framework. When a new simulator is added, it must be added to this enum. This forces the user to extend each component of the system to support the new simulator.

Parser The parser is the component responsible for translating the user-written specifications in YAML into a data structure that represents the benchmark model. The library *"com.charleskorn.kaml:kaml"* is used to provide an easy and clean way to parse the input file as shown in Listing 3.3.

Listing 3.3: Parsing of the input file.

```
  val inputFile = File(path)
  val benchmark = Yaml.default.decodeFromString(Benchmark.serializer(), inputFile.
      readText())
```

In certain scenarios, the configuration file for a scenario must be modified before being returned to the controller. This is the case for the *duration* parameter, which can be specified in the benchmark specification file. This parameter may change the duration of the execution of a scenario, overriding the simulator's input file. To achieve this, a *ConfigFileHandler* interface, which is implemented by each simulator-specific component, was created. This interface contains a single method, *handle*, which takes the scenario configuration as an input. Listing 3.4 shows the *ConfigFileHandler* interface.

Listing 3.4: ConfigFileHandler interface.

```
  interface ConfigFileHandler {
    fun editConfigurationFile(scenario: Scenario)
}
```

For example, the Alchemist implementation of the *ConfigFileHandler* interface reads the *duration* parameter from the scenario configuration and injects it to the Alchemist input file, overriding it.

Listener The listener is implemented as an interface with a *read* method, as Listing 3.5 shows, which takes the path to the CSV file as input and returns a ScenarioOutput. Saving simulation results in a CSV file is a standard and widely used approach, which is why this is the preferred method to read the simulator's output. In case it is not available as an option, the listener interface can be extended to implement a custom read function. The logic for reading from a CSV file is implemented directly in the Listener interface, while the *clearCSV* method is not implemented and must be handled by the specific simulator's listener. In general, the *clearCSV* function should clean the file from all unnecessary information, leaving the CSV file only with headers and values; otherwise, the reading will not be performed.

Listing 3.5: Listener interface.

```
interface Listener {
  fun read(path: String = ""): ScenarioOutput
  fun clearCSV(path: String)
}
```

The Alchemist implementation for the *ClearCSV* method is depicted in Listing 3.6

Listing 3.6: CSV file cleaning in Alchemist.

```
override fun clearCSV(path: String) {
    val lines = Files.readAllLines(Paths.get(path), StandardCharsets.UTF_8)
    val regexPatterns = listOf(
        Regex("#.*#"),
        Regex("#$"),
        Regex("# $"),
        Regex("# T.*"),
    )
    val dataLines = lines.filter { line ->
        !regexPatterns.any { pattern -> pattern.matches(line) }
    }
    val finalLines = dataLines.map { line ->
        val modifiedLine = line.replace(Regex("# "), "")
        modifiedLine
    }
    Files.write(Paths.get(path), finalLines, StandardCharsets.UTF_8)
}
```

Output and Result The concept of *Scenario Output* is implemented as a map that associates each metric with its value. The metric is represented as a *String*, while the value is represented as a list of *Any*. An instance of *ScenarioOutput* is shown in Listing 3.7.

Listing 3.7: ScenarioOutput implementation.

```
typealias ScenarioOutput = Map<String, List<Any>>
// "timeSteps" -> [10, 20, 30, 40]
// "meanError" -> [0.54, 0.32, 0.12, 0.05]
```

The *BenchmarkOutput* is implemented as a map that associates each scenario with its *Scenario Output*. The scenario is represented as a *String*, generated using the pattern *scenarioName-runNumber*. Listing 3.8 shows an instance of *BenchmarkOutput*.

Listing 3.8: BenchmarkOutput implementation.

```
typealias ScenarioOutput = Map<String, List<Any>>
// "NetLogo-Tutorial-1" -> ScenarioOutput(...)
// "NetLogo-Tutorial-2" -> ScenarioOutput(...)
// "Alchemist-Protelis-1" -> ScenarioOutput(...)
```

The *Scenario Result* is implemented as a data class, which contains a description of the result, its values, and the type of visualization to be used to show the data. The *Benchmark Result* is implemented as a list of *Scenario Result*. Listing 3.9 shows the implementation of the *ScenarioResult* along an instance of it and the *BenchmarkResult*.

Listing 3.9: BenchmarkResult and ScenarioResult implementation.

```
data class ScenarioResult(
  val description: String,
  val value: List<Any>,
  val visualisationType: VisualisationType,
)
typealias BenchmarkResult = List<ScenarioResult>
```

Data types as generic as possible are employed to support the widest range of data types. To enhance code quality and readability, the discussed data structures are defined using type aliases. These allow one to define a custom name for a data type and use it anywhere in the project, instead of the original definition. In addition, this allows application domain concepts to be tied into the code.

Controller After parsing the benchmark configuration file, the *Controller* begins to execute every Scenario in the order specified by the user.

Listing 3.10: Controller implementation.

```
val scenarioNameOrder: List<String> = benchmark.strategy.executionOrder
val scenarioMap: Map<String, Triple<Simulator, Scenario, Int>> = benchmark.
    simulators
    .flatMap { simulator ->
        simulator.scenarios.map { scenario ->
            scenario.name to Triple(simulator, scenario, scenario.repetitions)
        }
    }.toMap()
scenarioNameOrder.forEach { scenarioName ->
    val (simulator, scenario, repetitions) = scenarioMap.getOrElse(scenarioName)
         {
        throw IllegalArgumentException("Scenario $scenarioName not found")
    }
    for (i in 1..repetitions) {
        runBlocking {
            println("[TESTBED] Running scenario $scenarioName in ${simulator.
                name} simulator. Run number $i")
            createExecutor(simulator.name, simulator.simulatorPath, scenario)
            readSimulatorOutput(simulator, scenario, i)
        }
    }
}
```

As Listing 3.10 shows, the *Controller* creates a map containing all the scenarios and their respective simulators, and then iterates over the map, executing each scenario. It is important to note that each scenario execution is strictly sequential. A scenario is launched, the controller waits for the execution to finish, creates the Listener to read the results and only then moves on to the next scenario. All the results are saved in a data structure that will be displayed to the user by the *View*.

Listing 3.11: Controller: createExecutor method.

```
private fun createExecutor(simulator: SupportedSimulator, simulatorPath: String,
     scenario: Scenario) {
    val executor: Executor = when (simulator) {
        SupportedSimulator.ALCHEMIST -> executors.AlchemistExecutor()
        SupportedSimulator.NETLOGO -> executors.NetLogoExecutor()
    }
    executor.run(simulatorPath, scenario)
}
```

Listing 3.11 shows the implementation of the *createExecutor* method. This

function is responsible for creating the correct executor for the simulator. This is one of the cases where the framework's extensibility is evident. There must be an executor for each supported simulator, therefore, this method must be updated every time a new simulator is added.

Executor The *Executor* component is implemented as an interface, which is extended by each specific simulator's executor. To launch a simulator, a command must be generated and executed. As shown in Listing 3.12, the generation and the execution of the command are split into two methods, namely *getCommand* and *run*. The execution of the command does not depend on the simulator, therefore it is implemented in the *Executor* interface. The generation of the command, on the other hand, is specific to each simulator. It is delegated to the classes that extend the *Executor* interface, which are specific to each simulator.

Listing 3.12: Executor interface.

```
interface Executor {
  fun run(simulatorPath: String, scenario: Scenario) {
    val processBuilder = getCommand(simulatorPath, scenario)
    val process = processBuilder.start()
    val reader = BufferedReader(InputStreamReader(process.inputStream))
    var line: String?
    while (reader.readLine().also { line = it } != null) {
        println(line)
    }
    process.waitFor()
  }

  fun getCommand(simulatorPath: String, scenario: Scenario): ProcessBuilder
}
```

Listing 3.13 shows the implementation of the *getCommand* method for the NetLogo simulator.

Listing 3.13: NetlogoExecutor class.

```
class NetLogoExecutor : Executor {
  override fun getCommand(simulatorPath: String, scenario: Scenario):
      ProcessBuilder {
      return ProcessBuilder(
          "./NetLogo_Console",
          "--headless",
          "--model",
          scenario.input[1],
```

```
            "--setup-file",
            scenario.input[0],
            "--table",
            "./export.csv",
        )
            .directory(File(simulatorPath))
            .redirectErrorStream(true)
    }
}
```

3.2 Technologies

3.2.1 Framework technologies

Kotlin Kotlin[1] is a cross-platform, strong statically typed, general-purpose high-level programming language with type inference. It took inspiration from several programming languages including Java, Scala and others. Born as an object-oriented programming language, it includes a lot of functional programming elements such as first-class support for higher-order functions and lambda literals. The rise of Kotlin is testified by Google's choice to adopt it as the official language for Android development, replacing Java.

Originally, Kotlin was developed as a JVM language. The support for multi-platform development was added recently. This has great advantages as it reduces the time spent writing and maintaining the same code for different platforms while retaining the flexibility and benefits of native programming.

YAML To provide the user with the ability to write a benchmark configuration file, various options were considered, such as using JSON, YAML, Google Protobuf or even developing a DSL. The choice fell on YAML[2], a human-readable data serialization language. It is a superset of JSON, which means that any valid JSON file is also a valid YAML file. YAML is commonly used in applications where data needs to be represented in a format that is easy for humans to read and write, as well as for machines to parse and generate. It is used as a configuration language in different projects, such as Kubernetes, Docker, GitHub Actions, and many more.

[1] https://kotlinlang.org/
[2] https://yaml.org/

3.2.2 DevOps technologies

DevOps engineering is a software development methodology that aims at communication, collaboration and integration among all workers around an IT project. This set of techniques responds to interdependencies between software development and relative IT operations, allowing a faster and more efficient organization of software products and services. The following paragraphs describe which DevOps techniques have been used in the making of the system, focusing on the advantages that each procedure has brought.

Repository management The work is hosted on GitHub[3] and uses Git[4] as a DVCS (Distributed Version Control System). A DVCS is a version control system that allows multiple users to work on the same codebase at the same time and keeps track of every change made. The project was developed following a customized version of Git Flow, depicted in Figure 3.2.

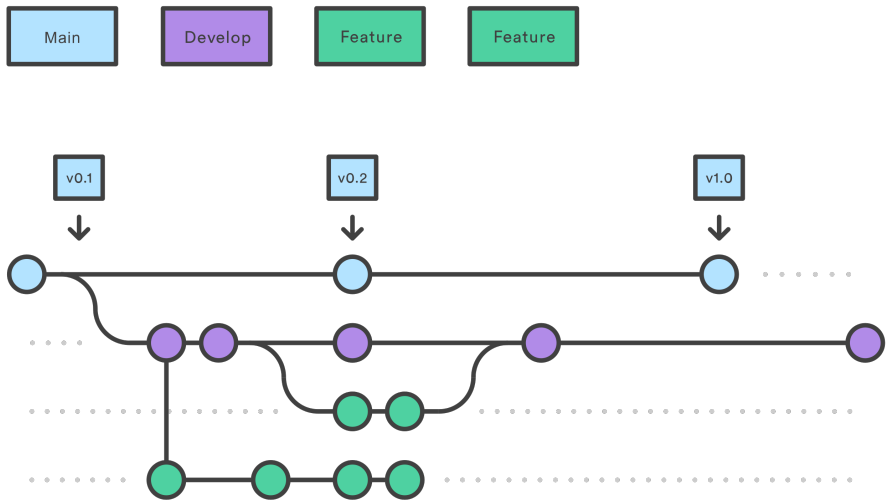


Figure 3.2: Git Flow.

The repository consists of a master branch and several feature branches. The master branch is the development reference, meaning that all the feature branches originate from it, and at the end of their existence, they are merged into the

[3]https://www.github.com/
[4]https://www.git-scm.com/

master branch. The feature branches are related to the developed features. For each feature to be implemented, a feature branch was created, and at the end of the development, a pull request was made to merge the content into the master branch. All feature branches followed the following naming convention: *feature/feature-name*

Build automation Build Automation refers to the automation of the build lifecycle of a project, which is the process from source code to product release and distribution. This allows automating operations that were previously done manually, making software deployment more efficient and less error-prone. In this work, Gradle[5], one of the most famous and widely used build systems, was used. Gradle is primarily used to manage dependencies with external libraries used in the project (e.g., the library for parsing YAML files). With Gradle, it is possible to define custom tasks, which are essentially atomic operations on the project that have input and output files and can depend on other tasks. This functionality has been leveraged to define the task that creates the framework's JAR file, which is then uploaded to GitHub during the release phase. Listing 3.14 shows the implementation of the custom task to generate the JAR file.

Listing 3.14: Custom task to generate the JAR file.

```
tasks.withType<ShadowJar> {
  archiveFileName.set("testbed.jar")
  manifest {
      attributes(
          mapOf(
              "Implementation-Title" to "Testbed",
              "Implementation-Version" to rootProject.version.toString(),
              "Main-Class" to "testbed.Testbed",
          ),
      )
  }
}
```

Continuous Integration Continuous Integration is a software development practice in which developers regularly merge their code changes into a central

[5]https://gradle.org/

repository, after which automated builds and tests are run. The key goals of continuous integration are to find and address bugs quicker, improve software quality, and reduce the time it takes to validate and release new software updates. In this work, GitHub Actions is used to automate the process of building, testing, and deploying the framework. The workflow of a GitHub Action is defined in a YAML file, which is stored in the repository under the path *.github/workflows*.

Listing 3.15 shows the build job of the CI/CD workflow, which runs the tests on different operating systems.

Listing 3.15: CI/CD workflow: build job.

```yaml
name: CI/CD Process

on:
 workflow_call:
 workflow_dispatch:

jobs:
 build:
  strategy:
   matrix:
    os: [ windows-2022, macos-12, ubuntu-22.04 ]
  runs-on: ${{ matrix.os }}
  concurrency:
   group: build-${{ github.workflow }}-${{ matrix.os }}-${{ github.event.number ||
     github.ref }}
   cancel-in-progress: true
  steps:
   - name: Checkout
    uses: DanySK/action-checkout@0.2.14
   - name: Test
    run: ./gradlew test
```

Listing 3.15 depicts the release job of the CI/CD workflow. To run this step, the *build* job must be completed successfully. This ensures that the tests are passed before releasing the software. Then it proceeds to install Node, which is required to run the semantic release tool, and finally runs the npm release command.

Listing 3.16: CI/CD workflow: release job.

```yaml
jobs:
 release:
  concurrency:
   # Only one release job at a time. Strictly sequential.
   group: release-${{ github.workflow }}-${{ github.event.number || github.ref }}
```

```
6     needs:
7      - build
8     runs-on: ubuntu-latest
9     if: >-
10     !github.event.repository.fork
11     && (
12      github.event_name != 'pull_request'
13      || github.event.pull_request.head.repo.full_name == github.repository
14     )
15    steps:
16     - name: Checkout
17      uses: actions/checkout@v4.1.1
18      with:
19       token: ${{ secrets.GH_TOKEN }}
20     - name: Find the version of Node from package.json
21      id: node-version
22      run: echo "version=$(jq -r .engines.node package.json)" >> $GITHUB_OUTPUT
23     - name: Install Node
24      uses: actions/setup-node@v4.0.2
25      with:
26       node-version: ${{ steps.node-version.outputs.version }}
27     - name: Release
28      env:
29       GH_TOKEN: ${{ secrets.GH_TOKEN }}
30      run: |
31       npm install
32       npx semantic-release
33
```

Versioning and Releasing For commits, Conventional Commits[6] is used, a convention that provides a set of possible commits, each with a different semantic, allowing the definition of the software version number based on the commit history.

The following set of commits is used:

- **Major Release**
  - Any commit terminating *!* causes a *breaking change*
- **Minor Release**
  - Commit type *feat* with any scope.
- **Patch Release**

[6] https://www.conventionalcommits.org/en/v1.0.0/

  - Commit type *fix* with any scope.
  - Commit type *docs* with any scope.
  - Commit type *chore* with scope *core-deps*.

- **No Release**
  - Commit type *test* with any scope.
  - Commit type *ci* with any scope.
  - Commit type *chore* with scope *deps*.
  - Commit type *refactor* with scope *deps*.

Thanks to the use of semantic release, it was possible to automate the versioning and releasing of the software. Every time a push is made to the master branch, semantic release calculates the version number and creates a release on GitHub, uploading the necessary assets.

The version number is defined in the format $X.Y.Z$ where X is a major version, Y is the minor version and Z is the patch version.

Chapter 4

Validation

4.1 Testing

4.1.1 Unit Testing

Unit testing is a software testing method to test individual units or components of software. In this work, we used it to test the behavior of the various components of the framework.

Given the nature of the work, most components require human judgment to be tested. This is the case for the *Executor* and *Listener* components, which are responsible for starting the simulator and reading its output, respectively. The tests on this component were conducted by repeatedly running the framework with different scenarios and checking the output manually. The *Parser* component, on the other hand, was tested automatically, to check if the parser correctly translates the input file into the benchmark model. Listing 4.1 shows the tests made on the parser.

Listing 4.1: Parser tests.

```
1  class ParserTest: FreeSpec({
2    "The parser" - {
3        val parser: Parser = ParserImpl()
4        "should parse a partial input file" {
5            // ARRANGE
6            val expectedBenchmark = simpleBenchmarkBuilder()
7            // ACT
8            val benchmark = parser.parse("src/test/resources/SimpleBenchmark.yml")
```

```
            // ASSERT
            assert(benchmark == expectedBenchmark)
        }
        "should parse a full input file" {
            val expectedBenchmark = fullBenchmarkBuilder()
            val benchmark = parser.parse("src/test/resources/FullBenchmark.yml")
            assert(benchmark == expectedBenchmark)
        }
        "should fail to parse a wrong file" {
            assertThrows<InvalidPropertyValueException> {
                parser.parse("src/test/resources/WrongBenchmark.yml")
            }
        }
    }
})
```

4.1.2 Integration Testing

Integration testing is a software testing method to test the behavior of the various components of the software when integrated. In this work, we used it to test the behavior of the simulators supported by the framework.

Checking if a simulator is running a scenario in the right way is not an easy task to do automatically, as it requires human judgment. Therefore, the integration tests are performed manually.

NetLogo NetLogo was tested by running one of the bundled models, namely the Wolf Sheep Predation model.

The first time, the simulation was launched through the framework in headless mode, without a graphical interface. Several logs were added to monitor the benchmark execution, and everything went as expected: the parser generated the benchmark model, which was used by the controller and executor to launch the simulation. At the end of the simulation, an output file containing the expected information was generated. This CSV file was then transformed into a data structure within the framework containing the results.

The second execution was launched again through the framework, but this time with the graphical interface activated, as Figure 4.1 shows. The same steps as before were observed, with the addition of the simulation being visually displayed.

After comparing the two executions and conducting further ones, no differences were noticed. We can therefore say that NetLogo has been integrated correctly into the framework.

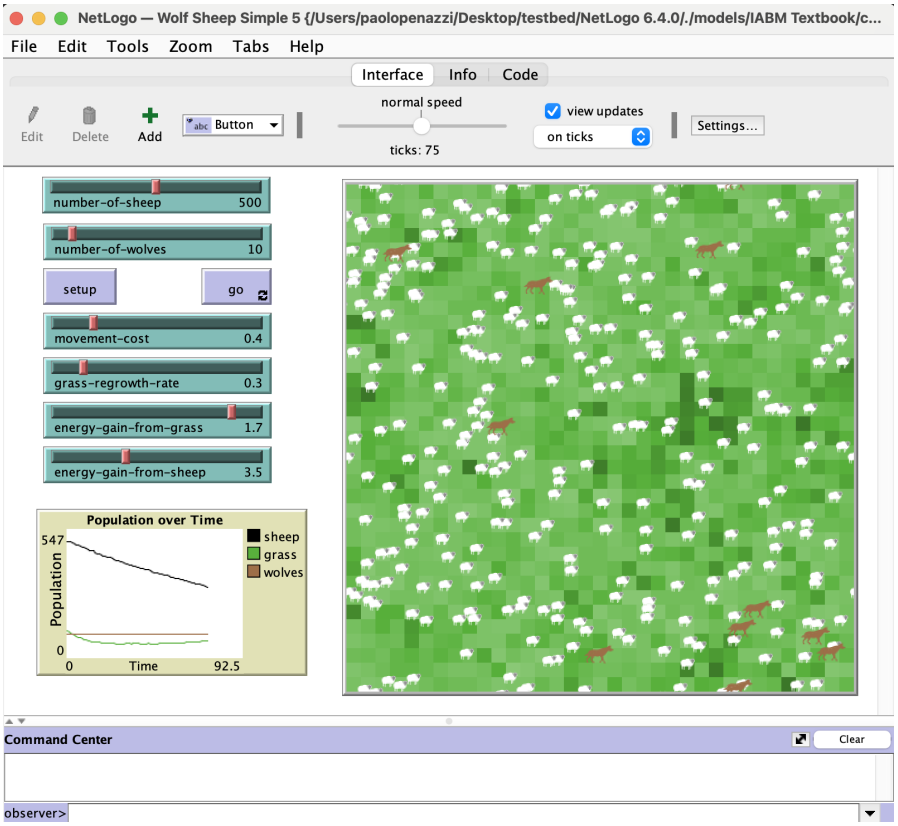


Figure 4.1: NetLogo simulation launched by the framework.

Alchemist The same process was applied to test the integration with the Alchemist simulator.

To test the behavior of Alchemist, a program that computes the gradient, a typical problem known in the literature was chosen. The first execution was done with the graphical interface activated, which allowed verifying the correct execution of the simulation thanks to the graphical effects applied to the nodes. This run is depicted in Figure 4.2. The second execution was done headless and provided the same results as the first. This process was repeated for both incarnations supported by the framework, namely Protelis and Sapere.

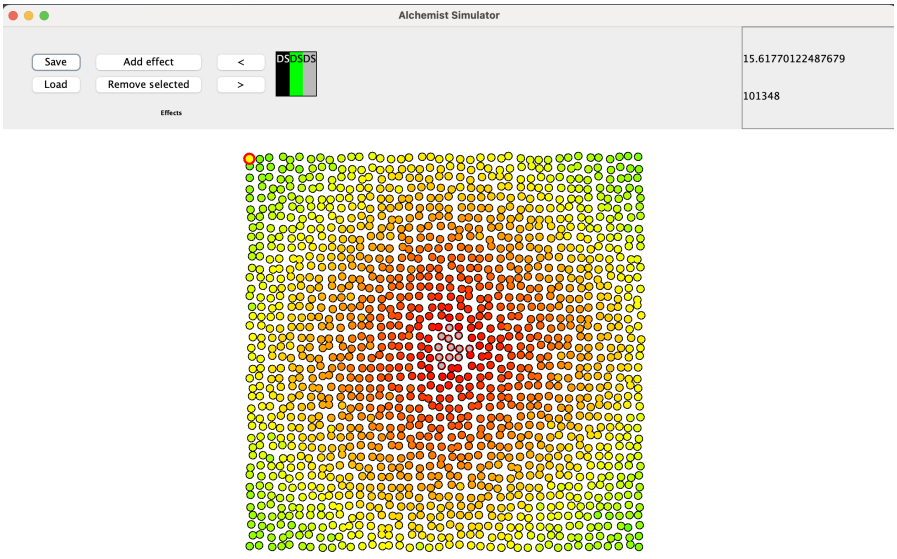


Figure 4.2: Alchemist simulation launched by the framework.

4.2 Evaluation

In this section, we will provide an example of how the framework can be used to benchmark CAS.

Definition and execution of a benchmark We start by defining a problem, which is the computation of the gradient in a grid of nodes. This is a known scenario in the literature and consists of computing the distance of each node in the grid from a source node. The solution to be tested is an implementation of the gradient computation in the Sapere incarnation of Alchemist.

We create a benchmark configuration file, Listing 4.2, which contains all the information about the benchmark to be executed.

Listing 4.2: Case of study: benchmark configuration file.

```
strategy:
 executionOrder:
  - Alchemist-SAPERE-gradient

simulators:
 - name: ALCHEMIST
  simulatorPath: "./"
  scenarios:
   - name: Alchemist-SAPERE-gradient
```

```
      description: Gradient computation.
      input: "sapere-gradient.yml"
      repetitions: 1
```

After the benchmark configuration file, we define the input file for the Alchemist simulation. This file, as Listing 4.3 shows, contains the environment configuration along with the export definition and the termination condition.

Listing 4.3: Case of study: Alchemist input file.

```
incarnation: sapere

launcher:
 type: HeadlessSimulationLauncher

network-model:
 type: ConnectWithinDistance
 parameters: [0.35]

deployments:
 type: Grid
 parameters: [-5, -5, 5, 5, 0.25, 0.25, 0.1, 0.1]
 contents:
  in:
   type: Rectangle
   parameters: [-0.5, -0.5, 1, 1]
  molecule: source
 programs: *grad

export:
 - type: CSVExporter
  parameters: {fileNameRoot: export, interval: 5, exportPath: ./}
  data:
   - time
   - molecule: gradient, value
   property: value
   aggregators: [mean]
   value-filter: onlyfinite

terminate:
 - type: afterTime
  parameters: 100
```

We proceed to implement a first version of the *gradient* program as Listing 4.4 shows.

Listing 4.4: Case of study: gradient program A.

```
_send: &grad
 - {time-distribution: 0.1, program: '{source} --> {source} {gradient, 0}'}
 - {time-distribution: 1, program: '{gradient, N} --> {gradient, N} * {gradient, N
    +#D}'}
 - program: >
   {gradient, N}{gradient, def: N2>=N} --> {gradient, N}
 - time-distribution: 0.1
   program: >
   {gradient, N} --> {gradient, N + 1}
 - program: >
   {gradient, def: N > 30} -->

```

Now we define the processing function, as Listing 4.5 depicts, which will be in charge of aggregating the data and returning the result. The function is implemented without the use of any external library, using pure Kotlin. It is not recommended to do so, as Kotlin is not a data analysis language and the code itself is not very readable.

Listing 4.5: Case of study: output processing function.

```
  val timeValues = benchmarkOutput["Alchemist-SAPERE-gradient-1"]!!["time"]
  val gradientValues = benchmarkOutput["Alchemist-SAPERE-gradient-1"]!!["value[
      mean]"]
  val timeToStabilize = gradientValues.reversed().foldRight(Triple(0.00, 0, 0)) {
      value, acc ->
      if (acc.second == 0) {
          if (value == acc.first && value != 0.00) Triple(
              acc.first,
              gradientValuesDouble.indexOf(acc.first),
              acc.third - 1
          ) else Triple(value, 0, acc.third + 1)
      } else {
          acc
      }
  }
  return listOf(
      ScenarioResult(
          "Time to stabilize: ",
          listOf(timeValuesDouble[timeToStabilize.third]),
          VisualisationType.SINGLE_VALUE
      )
  )
```

The function looks at the gradient value and returns the time at which the gradient stabilizes. A stable gradient is defined as two consecutive values that are the same and different from zero. To make the comparison values of type *Double*

are used, with a precision of two decimal places.

In Figure 4.3 we can observe the results of the benchmark execution, along with the output data of the simulator.

```
Time values: [0.0, 0.0, 5.0, 10.0, 15.0, 20.0, 25.0, 30.0, 35.0, 40.0, 45.0, 50.0]
Gradient values: [0.0, 0.0, 2.18, 4.13, 3.91, 3.82, 3.77, 3.76, 3.75, 3.75, 3.75, 3.74]

[TESTBED] Results:
Time to stabilize: 35.0
```

Figure 4.3: Case of study: Benchmark result A.

Compare a new solution to the reference We want to create a new solution for the gradient computation problem and compare it to the reference solution. To do so, we can use the benchmark configuration file and the processing function used for the reference solution, develop our own solution and run the benchmark to get the result.

To define the solution we take the Alchemist input file and change the time distribution of the reactions, as shown in Listing 4.6.

Listing 4.6: Case of study: gradient program B.

```
1  _send: &id001
2  - {time-distribution: 1, program: '{source} --> {source} {gradient, 0}'}
3  - {time-distribution: 2, program: '{gradient, N} --> {gradient, N} * {gradient, N
       +#D}'}
4  - program: >
5    {gradient, N}{gradient, def: N2>=N} --> {gradient, N}
6  - time-distribution: 1
7   program: >
8    {gradient, N} --> {gradient, N + 1}
9  - program: >
10   {gradient, def: N > 30} -->
```

Figure 4.4 shows the results of the benchmark execution. It is now possible to compare them against the reference solution.

```
Time values: [0.0, 0.0, 5.0, 10.0, 15.0, 20.0, 25.0, 30.0, 35.0, 40.0, 45.0, 50.0]
Gradient values: [0.0, 0.0, 4.33, 4.02, 4.01, 3.99, 4.0, 3.97, 3.97, 3.99, 3.99, 3.99]

[TESTBED] Results:
Time to stabilize: 30.0
```

Figure 4.4: Case of study: Benchmark result B.

The objective of this case study was not to assess particular aspects or the performance of a CAS. Instead, a basic metric was employed to estimate the system's stabilization speed. This experiment illustrated the feasibility of defining a benchmark, executing it, and deriving a meaningful outcome for the user. It serves as a reference for testing and evaluating new solutions without the need to redefine the benchmark; modifications can be made solely by altering the solution. As a result, the framework requirements outlined in the analysis phase are deemed fulfilled.

Chapter 5

Conclusion and Future Work

The work presented in this thesis has resulted in the development of a testbed prototype designed for benchmarking CAS. The objective of the framework was to provide a tool that integrates the most widely used simulators in the field of CAS. This allows users to assess a solution's behavior in different scenarios, utilizing different simulators, and facilitating the comparison of the obtained results.

To achieve this goal, the framework was designed to be flexible and extensible. Users can express benchmark configurations in a straightforward and intuitive way, and integrate new simulators without breaking the existing structure.

A pivotal aspect of the framework lies in its abstract design. Each system component has a general behavior that is independent of the simulator, yet remains incomplete. This architectural choice enables users to extend the framework by incorporating specific logic for new simulators. For the users who are not interested in introducing support for new simulators, the testbed still serves as a useful tool for evaluating solution behavior in specific scenarios. Furthermore, the framework promotes community collaboration, enabling individuals to define benchmarks and share results. This approach allows users to incorporate their solutions into pre-configured benchmarks, supporting result comparisons.

In our opinion, the testbed constitutes a promising foundation for the creation of a comprehensive tool for testing CAS. It addresses the key challenges in the development of an open benchmarking platform, providing a solid base for future works.

Future Works Despite the work that has been done, the testbed is still in its early stages and some features are missing for it to become a complete tool for testing CAS. The scientific community requires a high standard of quality and reliability, and the framework must be able to provide it. This section will provide a list of possible future works that could be done.

- **Increase the number of supported simulators** The framework currently supports two simulators, Alchemist and NetLogo. In the field of CAS, there are other simulators used to test different aspects of the systems. Supporting a wide range of simulators is necessary for the framework to be useful and relevant within the scientific community. Moreover, this would enable the user to test the behavior of the system in its entirety.

- **Multi-platform support** At the time of writing, the testbed has been developed for JVM platforms. Switching to Kotlin-Multiplatform would allow the framework to be compiled for different target platforms, such as JavaScript, iOS, and native. The ability to provide support for different platforms will help the framework to be more widely adopted.

- **Graphical User Interface** Currently, it is possible to interact with the framework only through a Command Line Interface (CLI). The user experience is known to be one of the most important aspects of software, as it can make the difference between a successful and an unsuccessful product. It is fundamental to provide the user with a graphical interface to interact with the application, both for the benchmark configuration and the visualization of the results. Benchmarking often involves comparing data from different solutions and a graphical interface would make this process easier and more intuitive.

- **Provide a way to download simulators** At present, the framework requires the user to download the simulators manually. Providing the user with a way to download a simulator, using a simple command, would make the framework more user-friendly. Moreover, this would ease the user from the task of manually downloading the simulators.

- **Maintaining benchmarks history** The comparison of new and existing solutions to well-known problems is a fundamental aspect when developing innovative solutions. Keeping a history of benchmark results in an online repository would facilitate such comparisons. Currently, users lack access to a database containing known benchmarks, various solutions, and their respective results. Providing them with this database would streamline the comparison process.

Bibliography

[1] Dhaminda B. Abeywickrama, Nicola Bicocchi, Marco Mamei, and Franco Zambonelli. The SOTA approach to engineering collective adaptive systems. *Int. J. Softw. Tools Technol. Transf.*, 22(4):399–415, 2020.

[2] Alessandro Aldini. Design and verification of trusted collective adaptive systems. *ACM Trans. Model. Comput. Simul.*, 28(2):9:1–9:27, 2018.

[3] Raquel Almeida, Henrique Madeira, and Marco Vieira. Benchmarking the resilience of self-adaptive systems: A new research challenge. In *29th IEEE Symposium on Reliable Distributed Systems (SRDS 2010), New Delhi, Punjab, India, October 31 - November 3, 2010*, pages 348–352. IEEE Computer Society, 2010.

[4] Aykut Argun, Agnese Callegari, and Giovanni Volpe. *Simulation of Complex Systems*. IOP Publishing, 2021.

[5] Rajive Bagrodia, Richard Meyer, Mineo Takai, Yu-an Chen, Xiang Zeng, Jay Martin, and Ha Yoon Song. Parsec: A parallel simulation environment for complex systems. *Computer*, 31(10):77–85, 1998.

[6] Jacob Beal, Danilo Pianini, and Mirko Viroli. Aggregate programming for the internet of things. *Computer*, 48(9):22–30, 2015.

[7] Jacob Beal and Mirko Viroli. Aggregate programming: From foundations to applications. In Marco Bernardo, Rocco De Nicola, and Jane Hillston, editors, *Formal Methods for the Quantitative Evaluation of Collective Adaptive Systems - 16th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2016, Bertinoro,*

Italy, June 20-24, 2016, Advanced Lectures, volume 9700 of *Lecture Notes in Computer Science*, pages 233–260. Springer, 2016.

[8] Aaron B. Brown, Joseph L. Hellerstein, Matt Hogstrom, Tony Lau, Sam Lightstone, Peter Shum, and Mary Peterson Yost. Benchmarking autonomic capabilities: Promises and pitfalls. In *1st International Conference on Autonomic Computing (ICAC 2004), 17-19 May 2004, New York, NY, USA*, pages 266–267. IEEE Computer Society, 2004.

[9] Antonio Bucchiarone and Marina Mongiello. Ten years of self-adaptive systems: From dynamic ensembles to collective adaptive systems. In Maurice H. ter Beek, Alessandro Fantechi, and Laura Semini, editors, *From Software Engineering to Formal Methods and Tools, and Back - Essays Dedicated to Stefania Gnesi on the Occasion of Her 65th Birthday*, volume 11865 of *Lecture Notes in Computer Science*, pages 19–39. Springer, 2019.

[10] Roberto Casadei and Mirko Viroli. Towards aggregate programming in scala. In *First Workshop on Programming Models and Languages for Distributed Computing, PMLDC@ECOOP 2016, Rome, Italy, July 17, 2016*, page 5. ACM, 2016.

[11] Roberto Casadei, Mirko Viroli, Gianluca Aguzzi, and Danilo Pianini. Scafi: A scala DSL and toolkit for aggregate programming. *SoftwareX*, 20:101248, 2022.

[12] Gabriella Castelli, Marco Mamei, Alberto Rosi, and Franco Zambonelli. Pervasive middleware goes social: The SAPERE approach. In *Fifth IEEE Conference on Self-Adaptive and Self-Organizing Systems, SASOW 2011, Ann Arbor, MI, USA, October 3-7, 2011, Workshops Proceedings*, pages 9–14. IEEE Computer Society, 2011.

[13] Christian S. Collberg and Todd A. Proebsting. Repeatability in computer systems research. *Commun. ACM*, 59(3):62–69, 2016.

[14] Jack Collins, Mark Robson, Jun Yamada, Mohan Sridharan, Karol Janik, and Ingmar Posner. RAMP: A benchmark for evaluating robotic assembly manipulation and planning. *IEEE Robotics Autom. Lett.*, 9(1):9–16, 2024.

[15] Dairo de Ruck, Victor Goeman, Michiel Willocx, Jorn Lapon, and Vincent Naessens. Linux-based iot benchmark generator for firmware security analysis tools. In *Proceedings of the 18th International Conference on Availability, Reliability and Security, ARES 2023, Benevento, Italy, 29 August 2023- 1 September 2023*, pages 19:1–19:10. ACM, 2023.

[16] Jozo J. Dujmovic. Universal benchmark suites. In *MASCOTS 1999, Proceedings of the 7th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 24-28 October, 1999, College Park, Maryland, USA*, pages 197–205. IEEE Computer Society, 1999.

[17] Hamza Es-Samaali, Aissam Outchakoucht, Siham Benhadou, Oussama Mounnan, and Anas Abou El Kalam. Anomaly detection for big data security: A benchmark. In *BDET 2021: The 3rd International Conference on Big Data Engineering and Technology, Singapore, June 25-27, 2021*, pages 35–39. ACM, 2021.

[18] Xavier Etchevers, Thierry Coupaye, and Guy Vachet. Experiences in benchmarking of autonomic systems. In Athanasios V. Vasilakos, Roberto Beraldi, Roy Friedman, and Marco Mamei, editors, *Autonomic Computing and Communications Systems, Third International ICST Conference, Autonomics 2009, Limassol, Cyprus, September 9-11, 2009, Revised Selected Papers*, volume 23 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 48–63. Springer, 2009.

[19] Odd Erik Gundersen and Sigbjørn Kjensmo. State of the art: Reproducibility in artificial intelligence. In Sheila A. McIlraith and Kilian Q. Weinberger, editors, *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 1644–1651. AAAI Press, 2018.

[20] Matthias M. Hölzl, Axel Rauschmayer, and Martin Wirsing. Engineering of software-intensive systems: State of the art and research challenges. In Mar-

tin Wirsing, Jean-Pierre Banâtre, Matthias M. Hölzl, and Axel Rauschmayer, editors, *Software-Intensive Systems and New Computing Paradigms - Challenges and Visions*, volume 5380 of *Lecture Notes in Computer Science*, pages 1–44. Springer, 2008.

[21] Serge Kernbach, Thomas Schmickl, and Jon Timmis. Collective adaptive systems: Challenges beyond evolvability. *CoRR*, abs/1108.5643, 2011.

[22] Charles M. Macal and Michael J. North. Tutorial on agent-based modelling and simulation. *J. Simulation*, 4(3):151–162, 2010.

[23] Marco Mamei, Matteo Vasirani, and Franco Zambonelli. Self-organizing spatial shapes in mobile particles: The TOTA approach. In Sven Brueckner, Giovanna Di Marzo Serugendo, Anthony Karageorgos, and Radhika Nagpal, editors, *Engineering Self-Organising Systems, Methodologies and Applications [revised versions of papers presented at the Engineering Selforganising Applications (ESOA 2004) workshop, held during the Autonomous Agents and Multi-agent Systems conference (AAMAS 2004) in New York in July 2004, and selected invited papers]*, volume 3464 of *Lecture Notes in Computer Science*, pages 138–153. Springer, 2004.

[24] Rocco De Nicola, Stefan Jähnichen, and Martin Wirsing. Rigorous engineering of collective adaptive systems: special section. *Int. J. Softw. Tools Technol. Transf.*, 22(4):389–397, 2020.

[25] Rocco De Nicola, Michele Loreti, Rosario Pugliese, and Francesco Tiezzi. A formal approach to autonomic systems programming: The SCEL language. *ACM Trans. Auton. Adapt. Syst.*, 9(2):7:1–7:29, 2014.

[26] Michael J North, Nicholson T Collier, Jonathan Ozik, Eric R Tatara, Charles M Macal, Mark Bragen, and Pam Sydelko. Complex adaptive systems modeling with repast simphony. *Complex Adaptive Systems Modeling*, 1(1), March 2013.

[27] D Pianini, S Montagna, and M Viroli. Chemical-oriented simulation of computational systems with ALCHEMIST. *Journal of Simulation*, 7(3):202–215, aug 2013.

[28] Danilo Pianini, Mirko Viroli, and Jacob Beal. Protelis: practical aggregate programming. In Roger L. Wainwright, Juan Manuel Corchado, Alessio Bechini, and Jiman Hong, editors, *Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015*, pages 1846–1853. ACM, 2015.

[29] Stuart Russell and Peter Norvig. A modern, agent-oriented approach to introductory artificial intelligence. *SIGART Bull.*, 6(2):24–26, 1995.

[30] Andreas Spillner, Tilo Linz, and Hans Schaefer. *Software Testing Foundations: A Study Guide for the Certified Tester Exam (3rd ed.)*. Rocky Nook, 2011.

[31] Casper Thule, Kenneth Lausdahl, Cláudio Gomes, Gerd Meisl, and Peter Gorm Larsen. Maestro: The INTO-CPS co-simulation framework. *Simul. Model. Pract. Theory*, 92:45–61, 2019.

[32] Quan Tu, Shilong Fan, Zihang Tian, and Rui Yan. Charactereval: A chinese benchmark for role-playing conversational agent evaluation. *CoRR*, abs/2401.01275, 2024.

[33] Ante Vilenica and Winfried Lamersdorf. Benchmarking and evaluation support for self-adaptive distributed systems. In Leonard Barolli, Fatos Xhafa, Salvatore Vitabile, and Minoru Uehara, editors, *Sixth International Conference on Complex, Intelligent, and Software Intensive Systems, CISIS 2012, Palermo, Italy, July 4-6, 2012*, pages 20–27. IEEE Computer Society, 2012.

[34] Mirko Viroli, Giorgio Audrito, Ferruccio Damiani, Danilo Pianini, and Jacob Beal. A higher-order calculus of computational fields. *CoRR*, abs/1610.08116, 2016.

[35] Yihan Zhang, Lyon Zhang, Hanlin Wang, Fabián E. Bustamante, and Michael Rubenstein. Swarmtalk - towards benchmark software suites for swarm robotics platforms. In Amal El Fallah Seghrouchni, Gita Sukthankar, Bo An, and Neil Yorke-Smith, editors, *Proceedings of the 19th International Conference on Autonomous Agents and Multiagent Systems, AAMAS '20, Auckland, New Zealand, May 9-13, 2020*, pages 1638–1646. International Foundation for Autonomous Agents and Multiagent Systems, 2020.

Acknowledgements

First and foremost, I would like to express my gratitude to my supervisors, Prof. Danilo Pianini and Prof. Lukas Esterle, who have supported and guided me throughout this work. Thank you for consistently providing feedback and being available for discussions.

I also want to thank my family, who has always supported me and never ceased to offer their encouragement. A special thank goes to my girlfriend Rita, who has been a constant source of support and love, always motivating me to be the best version of myself. I want to express my gratitude to my friends, Dani, Magna, Ste, Raffo, Enea and Luca, my university colleagues, and all the people who have been part of my life during this journey.