Configuration and presentation system of job scheduling simulator

Bachelor Thesis

Gabriela Podolníková

Brno, Spring 2014
Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Advisor: RNDr. Dalibor Klusáček, Ph.D.
Acknowledgement

I would like to thank my supervisor, Dalibor Klusáček for help throughout the work. I am very grateful for his advice and valuable discussions. I would also like to thank my boyfriend, Filip Höfer for his help and encouragement during my study.
Abstract

This thesis focuses on job scheduling simulator – Alea – and the extensions performed. Scheduling simulators are developed for scientific simulations under repeatable and controllable conditions. In order to improve usage and maintainability of the simulator, several modifications were applied.

The main goal of this thesis is to implement a separation of the source code and the configuration of the simulation. A user-friendly interface for parameter setting was designed as a web application. Also, scheduling metrics were separated from the source code and implemented as plugins. Prototype implementations are described in this thesis.

The configuration system persists the items in a generic properties file. The graphical user interface is implemented by a JSP page. The plugin extension is provided by a standard Java tool – the Java reflection.

These extensions increase the modularity of the simulator and provide a more user-friendly software.
Keywords

Grid, scheduling, optimization criteria, Alea, configuration, simulator, web application, plugin, metrics
## Contents

1 Introduction ................................................. 3

2 Grid and Job Scheduling ................................. 4

3 Alea – Job Scheduling Simulator ...................... 7
   3.1 Motivation ........................................... 7
   3.2 Alea Overview ..................................... 7
   3.3 Simulation Outputs – CSV Results ................. 9

4 Motivation for Improvements .......................... 11
   4.1 Analysis of Alea ................................... 11
   4.2 Objectives of This Work ......................... 12

5 Configuration System ................................. 14
   5.1 Backend ........................................... 14
      5.1.1 Backend Implementation ..................... 15
      5.1.2 Configuration Items in Properties File ....... 16
   5.2 Frontend ........................................... 16
      5.2.1 GUI Setup ................................... 17
      5.2.2 Item Configuration in GUI ................... 17
      5.2.3 GUI Implementation ......................... 19
      5.2.4 GUI Request Flow ........................... 20
      5.2.5 Logo ........................................ 21

6 Simulation Output, Optimization Criteria as Plugins . 22
   6.1 Simulation Output .................................. 22
      6.1.1 Technical Solution ............................ 23
   6.2 Scheduling Metrics as Plugins ................. 23
      6.2.1 Java Reflection ............................... 25
      6.2.2 Plugin Configuration ......................... 25
      6.2.3 Implementation of Plugin Configuration ... 27
      6.2.4 Example Plugins ............................. 27

7 Online Source Repository .......................... 29
   7.1 Alea on Github ................................. 29

8 Conclusion ............................................. 31
   8.1 Summary ......................................... 31
   8.2 Suggestions for Future Work .................... 32
Bibliography ........................................... 33
Appendices ........................................... 35
A GitHub Usage ...................................... 36
B Source Code ........................................ 37
Chapter 1

Introduction

In 1998, Ian Foster and Carl Kesselman compared the state of information technologies to the state of electric power grid at the beginning of the twentieth century [10]. The new millennium brought a big progress in the field of computing in parallel environments. With this development, an array of new possibilities came in the area of research. A large research effort is put into finding ways to use the systems more efficiently. In this regard, an important role is played by scheduling algorithms. Testing on real machines is time-consuming, hence software scheduling simulators were introduced.

The main topic of this work is the job scheduling simulator Alea [13], which was developed at the Faculty of Informatics at Masaryk University in Brno. Alea was designed for studying, testing and evaluating different techniques used in job scheduling. Different data sets can be used as input of the application. The performance of various scheduling algorithms on each data set can be compared, thanks to the simulation of the same environment for every experiment. These conditions are impossible while testing on real hardware. The simulator extends an existing set of tools – GridSim [3] – and provides new tools for simulation of scheduling problems in a parallel computing system, so called Grid.

The Grid and job scheduling are briefly explained in Chapter 2. Subsequently, Chapter 3 presents the Alea simulator, including its basic structure, inputs and outputs. Chapter 4 continues with the analysis of this software and proposes improvements.

The contribution of this work consists of several Alea extensions and modifications. The two most important improvements are a configuration subsystem and the modularization of scheduling metrics. These works are described in Chapter 5 and Chapter 6, respectively. This work also aims at broadening Alea audience. In this regard, an online source repository was established, see Chapter 7.
Chapter 2

Grid and Job Scheduling

This chapter provides a basic understanding of the term Grid and the area of job scheduling. The definitions from this chapter will be used further in this work.

A Grid represents a decentralized interconnected parallel system with multiple heterogeneous resources. Dong and Akl [6] describe it as “a shared environment implemented via the deployment of a persistent, standards-based service infrastructure that supports the creation of, and resource sharing within, distributed communities.”

A job is an entity that needs to be executed on some resources. A resource is a means that is necessary to process a job. It is typically CPU, RAM, disc space, or network bandwidth. A job can use one or more instances of a resource type for its execution. For example, if the job uses a single CPU resource, the job is called sequential. If it requires more CPU resources, it is called parallel. Users are the job owners that submit the jobs into the Grid. Each job has its static parameters, such as its runtime, the release date (the earliest time when the job can be executed), or the weight (the importance of this job relatively to other jobs in the system). Job also has dynamic parameters, such as the start time and the completion time. These parameters are not known at the time of submitting the job into the system.

Job scheduling represents the allocations of jobs onto computational resources in time. The Grid scheduling problem consists of a problem of finding a schedule that fulfills certain criteria. These criteria are referred to as optimization criteria that are mathematically expressed using so called objective functions, i.e., given scheduling metrics. The basic goal of optimization is minimizing or maximizing the value of an objective function. With such functions, it is possible to measure the quality of solutions. The optimization criteria can be divided into three categories [6] [12] as follows:

- Resource-centric criteria are used to study the system performance. Popular criteria include electricity consumption or machine usage.
The latter is convenient for maximizing the resource utilization over time. As far as Grid scheduling is concerned, system dynamics and heterogeneity need to be taken into account. Hence, improved variations of the objective functions may be needed, such as weighted machine usage [12]. Alternatively, the measurements can be applied to a limited subset of jobs that are online in a certain timeframe.

- **Job-centric criteria** allow to measure the performance of job execution. The *response time* is the time from the job’s submission until its completion. This metric contains two parts, one being the actual running time and the other being the waiting time. The *wait time* expresses the interval before job’s execution. The *slowdown* criterion is the ratio of the actual response time to the theoretical response time if there was no waiting time. The slowdown may be a better metric than the response time, because it respects the size of the jobs. In turn, it allows to compare the performance of small jobs and large jobs without large jobs getting a penalty for the long execution time [8]. The slowdown’s value must be, by definition, higher or equal to one. On the other hand the extremely small jobs are in disadvantage, so the *bounded slowdown* [9] can be applied. Here, the minimal job runtime makes sure to be greater than some threshold – a predefined time constant. This threshold can eliminate the problem with very short jobs in the standard slowdown criterion.

- **Fairness-related criteria** aim at maintaining a fair share of resource utilization for the consumers. Traditionally, the consumers are the jobs, i.e., one of *n* jobs deserves one *n*-th of resources. Such fairness can be measured via job-centric criteria such as the slowdown. Some authors [12] apply the idea of fairness to the users of the system. In that case, each user of *m* users deserves one *m*-th of the resources. The measurements can be performed by aggregating job-centric criteria per their owners and then measuring, e.g., the standard deviation of the resulting distribution. An example of such a criterion is *average user wait time*. Fairness criteria play an important role in scheduling algorithms since they are closely related to user-perceived behavior of the system.

In order to satisfy the selected criteria, many techniques have been developed. There are multiple ways of categorizing the job scheduling techniques on Grids [4]. One categorization divides scheduling to centralized and decentralized. In the centralized scheduling, a set of clusters is con-
trolled by a single scheduler. This scheduler is aware of all jobs and assigns them onto resources in the system. In the decentralized scheduling, the system has multiple levels that are cooperating. The system is managed by global schedulers that are managing the local schedulers.

Another categorization differentiates scheduling to static and dynamic. The former assumes that all conditions are known in advance. The latter considers system dynamics including machine failures, restarts, and temporal unavailability of resources. Also, user activity plays role as jobs are arriving when submitted by their owners. To sum up, dynamic scheduling considers job and resource availability to be changing in time. The Grid environment is typically dynamic, thus the dynamic scheduling is dominant in this area.

Scheduling algorithms can also be divided to optimal and suboptimal solutions, according to the quality of the generated schedule. The optimization criteria could generate an optimal schedule, nevertheless due to the NP-completeness of the problem, job scheduling becomes challenging with the increasing size of the system. Therefore, real-world solutions are suboptimal. Approximate or heuristic approaches are applied. Approximation algorithms return results that have a bounded distance from the optimum. For each approximation algorithm, there is a factor $k$ defining that the found solution is in the worst case $k$-times worse than the optimal solution. The heuristic approach is not bound by formal definitions. It aims at making the best assumptions with regard to the cost of the generated solution. Since the availability of data for precise computations is limited in the Grid environment, the heuristics are more suitable [6].
Chapter 3

Alea – Job Scheduling Simulator

In this chapter, the job scheduling simulator – Alea – is presented. We will start with the motivation for creating such simulator. Then, the high-level design of the software is described as well as its inputs and outputs. Internal classes that are important for this work are mentioned. This chapter presents the state of Alea before implementing the improvements that comprise the contribution of this work.

3.1 Motivation

Alea [13] focuses on simulations of job scheduling on Grids. A Grid represents an interconnected parallel computing system that consists of multiple nodes and changes in time. Job scheduling in these environments is a difficult task. In order to be more efficient, researchers develop new algorithms that would be appropriate for the given system. The new algorithms must be tested and evaluated before they can be applied to the real systems. There are many reasons why the experimental evaluations cannot be performed directly in the real system, such as the cost of resources, dynamics of the system and the reliability. In order to obtain meaningful evaluation outputs, the environment needs to be controllable and repeatable. This cannot be achieved in a real Grid.

With the aim to obtain a controllable environment, many simulators have been created. These simulators are very useful because they are evaluating existing or proposed solutions by using different setups and different data sets.

3.2 Alea Overview

The Alea simulator has been developed since 2007 and many features have been included. Alea contains scheduling algorithms, various objective functions and the support for configuring machine characteristics. It also of-
fers visual output in bitmap formats as well as workload parsers that read widely used formats, e.g., the Grid Workloads Format (GWF) and the Standard Workload Format (SWF) [13].

The Alea simulator represents a centralized scheduling system allowing to apply and compare various scheduling algorithms. Alea is based on the latest version of the GridSim toolkit, GridSim 5. This toolkit extends simulation package SimJava [11]. The Alea simulator extends the original functionality of the GridSim toolkit. It is based on the Java programming language which allows easy development and portability.

Figure 3.1 presents the original high-level structure of the Alea simulator. Alea provides the Scheduler entity which represents the centralized scheduler. Data sets contain the information about jobs, machines, and queues. The Scheduler class holds simulation data, handles communication and delegates scheduling decisions to the chosen scheduling algorithm. During the simulation, the ResultCollector class collects the results and stores them into CSV files. The structure of the files is explained in Section 3.3. The Visualizer class generates the graphs.

The ExperimentSetup class contains the main method where the parameters of the simulation are set and the simulation is launched. This class
3. Alea – Job Scheduling Simulator

takes care of the lifecycle of a simulation experiment. It instantiates the key classes, such as the Scheduler entity. The data set and algorithm selection is performed and the simulation is started. Finally, the results need to be stored, therefore the methods from ResultCollector class are called.

The optimization criteria (also called objective functions or scheduling metrics) are calculated in several methods in the ResultCollector class. When adding a new objective function, it is necessary to modify these methods in order to obtain the results in the Results file. The names of objective functions appear in the Results file as headers in the appropriate columns.

3.3 Simulation Outputs – CSV Results

Alea generates outputs that capture multiple perspectives and levels of abstraction. The files are named as follows: Results, RGraphs, SGraphs, WGraphs, jobs and users. The contents of these files are explained in the following paragraphs.

The Results file contains the results of the optimization criteria calculated on every data set and scheduling algorithm performed. It is a CSV file. Its structure is shown in Figure 3.2. It consists of several row groups. The number of groups is equal to the number of data sets that entered the experiment. The number of rows in a group is equal to the number of scheduling algorithms that were evaluated during the experiment, plus a header row. The header row is the first row in each group. It contains the names of evaluated scheduling metrics, one column for each metric.

RGraphs, SGraphs and WGraphs contain the entries for cumulative distribution function graphs. These entries concern response time, slowdown, and wait time metrics, respectively. The jobs file provides the list of all jobs in the system and holds the job ID, job’s arrival time, wait time, run time, the number of utilized CPUs and RAM usage. Also there is the user ID – the owner of this job – and the name of the queue. The users file contains the data for all users in the system. It stores the user ID and fairness-related criteria, such as the total user wait time.
### Figure 3.2: Results File

<table>
<thead>
<tr>
<th>data set 1</th>
<th>metric 1</th>
<th>metric 2</th>
<th>...</th>
<th>metric $N_m$</th>
</tr>
</thead>
<tbody>
<tr>
<td>scheduling algorithm 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>scheduling algorithm $N_a$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>data set 2</th>
<th>metric 1</th>
<th>metric 2</th>
<th>...</th>
<th>metric $N_m$</th>
</tr>
</thead>
<tbody>
<tr>
<td>scheduling algorithm 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>scheduling algorithm $N_a$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

...
Chapter 4

Motivation for Improvements

In this chapter, we discuss the motivations to extend Alea. The simulator is analyzed and the points that should be improved are identified. This chapter proposes an updated structure of the simulator and sets the objectives of this work.

4.1 Analysis of Alea

Alea is a Java application built on top of a Grid scheduling simulator framework GridSim [3]. Alea utilizes the framework and adds many features that allow simulation variations via the support of different data inputs, scheduling algorithms, scheduling metrics, and textual as well as graphic outputs. We will focus on the configuration of Alea features and the overall setup of a grid scheduling experiment in this simulator. Also, we will investigate the support of various scheduling algorithms and metrics with respect to its extensibility. This section refers to the structure of Alea described in Chapter 3, i.e., the initial state before performing the extensions that are the objectives of this work.

As of Alea 3, the configuration of the experiment’s parameters was performed directly in the code of ExperimentSetup class. Typically, these parameters are set for each experiment again in order to try different variants of job scheduling. The user had to modify the parameters in the source code and recompile the program. Then it was possible to start the simulation with the new settings. This way of handling the configuration was not comfortable for users. Thus it was deemed necessary to design a system for easy parameter setting with a user-friendly interface, preferably through a web application. There were several reasons for a web-based solution. It is well-received by most users and it could also be reused in a web-based Alea release in the future.

The next problem was that the simulation outputs, the CSV files, were all stored in the same directory, namely in the Alea root folder. Hence,
4. Motivation for Improvements

The need to structure the outputs was identified. The envisioned structure of the file paths should be easy to navigate and provide information on utilized algorithms, data sets, and settings.

With respect to the support of various scheduling techniques and metrics, Alea can host custom scheduling algorithms and objective functions. Each scheduling algorithm is encapsulated into its own class that implements the `SchedulingPolicy` interface. This approach provides a good modularization. However, the code of the simulator needs to be modified and recompiled in order to use a new scheduling algorithm or to apply changes to an existing one.

Unlike scheduling algorithms, objective functions were not encapsulated into individual classes. They were calculated directly in the code of the `ResultCollector` class. If the user needed to add a new objective function, they had to modify several methods in this class. This solution required a good understanding of the source code. Therefore, new modularity was needed in the simulator in order to better facilitate such extensions. Ideally, the objective functions shall be added to the simulator as plugins, one plugin representing one objective function. A new objective function should be added without modifying the existing code.

4.2 Objectives of This Work

The identified shortcomings of Alea brought ideas for improvements. Figure 4.1 shows the newly designed extensions for Alea. These extensions are highlighted as bold in the figure. Three areas for improvement were identified: replacement of hard-coded simulation parameters by a configuration system, re-organization of CSV outputs, and improved modularization through isolating individual scheduling metrics.

The configuration system shall remove the need to recompile the simulator in order to apply configuration changes. The configuration shall be stored in a non-Java text file that should be human-readable. There shall also be a graphical interface in order to provide user-friendly configuration management.

The re-organization of CSV outputs should make output self-descriptive through intelligent naming in the file paths. The output writing mechanism shall also be revised so that it prevents accidental rewriting of results from a previous experiment.

The modularization of scheduling metrics shall facilitate easier adding of custom metrics into the simulator. It shall provide proper encapsulation
and preferably also remove the need to modify and recompile the simulator due to adding a new metric. The metrics shall be added or removed as plugins.

These requirements define the main objectives of this work. Besides the functional requirements, this work also attempts to apply the software engineering principles in order to improve code maintainability. It includes code refactoring and applying common design patterns. The implementations of the tasks are described in the following chapters.

In addition to source code changes, we have also decided to improve the way Alea is presented to the developer community. Alea is an open source project and therefore it shall use modern tools for sharing the source code. There are many web-based source code repository services, typically free of charge. Developers use them for collaboration, versioning, and bug tracking. These sites facilitate the life of communities of programmers collaborating on interesting projects. Alea shall also join these trends.

Figure 4.1: Extensions Overview
Chapter 5

Configuration System

Job scheduling and job scheduling simulations may have many variations. There can be different data sets, scheduling algorithms, and measured metrics. The computations can also be adjusted by various characteristics of simulated systems. Therefore, it is natural to introduce a configuration system.

This chapter discusses the implementation of a new configuration system for Alea. This system has two parts: the backend and the frontend. The backend part provides the definition of configuration data format as well as read, write and check routines for configuration items. The frontend part is a web-based user interface.

5.1 Backend

The requirement to separate the configuration from the source code was handled by creating a configuration file and associated routines. There were multiple implementation options for this problem, therefore several solutions were considered, such as an XML file, JSON (JavaScript Object Notation) or a simple properties file. These technologies are used for data serialization, it means that they convert a (configuration) object into a format that can be stored and loaded. The first two formats support structured data, but they suffer from unnecessary redundancies, especially in case of simple objects. Since Alea generally requires simple configuration objects, we decided to use Java property files. They are expressive enough, easy to handle and the Java Runtime Environment (JRE) provides the basic routines for their handling.

In a properties file, each entry is stored on a single line. The line takes the format of `key=value`. Both the key and the value are free-form strings and are separated by the equals sign or colon sign, white space before and after the separator is ignored. The separator characters can also appear in keys and values, but each such occurrence has to be escaped by a backslash.
JRE provides a support for loading and writing property files since version 1.0. This support is implemented in `java.util.Properties` class. The `Properties` class inherits from class `java.util.Hashtable`, that implements a table, where keys and values are general Java Objects [14]. The Property class uses a String as a key and a String as a value. It also adds routines for loading and storing the data. The data can be read and written using streams such as files. Hence, a property file can be easily handled from a Java program.

### 5.1.1 Backend Implementation

In order to manage Alea configuration, the `AleaConfiguration` class was created. Its static structure is shown in the right-hand part of Figure 5.1. Its code delegates property file loading and writing to the JRE-provided `Properties` class. Besides standard get-and-set functionality, the `AleaConfiguration` class provides several other features, such as type checking and conversions for all configuration object types that are required by Alea. It is possible to handle the following types: String, int, double, boolean, and the array of String, int and boolean. Scalar types are stored in their usual string representation. Array types are persisted as comma-separated lists.

---

**Figure 5.1: AleaConfiguration Class and Configuration GUI**

```text
AleaConfiguration
- fileName : String
- props : Properties
- inputStreamProvider : inputStreamProvider

+AleaConfiguration()
+AleaConfiguration(path : String)
+AleaConfiguration(inputStreamProvider : inputStreamProvider)

+getString(key : String) : String
+getInt(key : String) : int
+getDouble(key : String) : double
+getBoolean(key : String) : boolean
+getIntArray(key : String) : int[]
+getStringArray(key : String) : String[]
+getBooleanArray(key : String) : boolean[]
+getPluginConfiguration(pluginIndex : int) : Map<String, String>
+getPluginConfigurationKey(index : int, pluginKey : String, common : boolean) : String
+getFile() : File
+getFileName() : String
+getKeys() : Enumeration<?>
+getKeyList() : List<String>
+save() : void
+setString(key : String, value : String) : void
+deleteString(key : String)
+typeCheck(type : String, value : String) : boolean
```

---

15
The ExperimentSetup class previously held all configuration in hard-coded fashion. Now it acts as a client of the AleaConfiguration class. In order to retrieve each parameter, the ExperimentSetup class calls a getter method according to the configuration item type and passes the configuration item key. The called method fetches the appropriate entry, performs type conversion, and returns an object or primitive value of expected type.

5.1.2 Configuration Items in Properties File
As a part of this work, a default Alea configuration file was created. It is located in the root folder of Alea as configuration.properties. There are values for all configuration items that are processed by Alea, a total count of 50. A preview of the first few lines of the default configuration is provided in Figure 5.2.

Figure 5.2: Properties File Example

```
visualize=false
reqs=false
failures=false
use_queues=false
use_speeds=false
use_heap=true
```

5.2 Frontend

The simplest way of managing Alea configuration is editing the configuration file by a generic text editor. While this is simple and may be sufficient for advanced users, we have decided to implement a graphical interface that will provide a better comfort of managing the configuration. The solution is based on a Java Server Pages (JSP) web application. As such, it runs in a separate container, as shown in Figure 5.3. The data exchange between the GUI and the simulator is realized via the configuration file. The intended added value of the GUI is in providing these features:

- Textual description for each configuration item;
- Type information for each configuration item;
- Organizing configuration items into categories;
- Automatic type checking before storing the configuration;
5. Configuration System

- "Restore defaults" function;
- User-friendly interface.

Figure 5.3: Container Diagram

5.2.1 GUI Setup

For a comfortable use of the Alea web configuration, it is necessary to set the environment variable \texttt{ALEA\_HOME} prior to launching the web container. It must point to the root directory of the Alea project. If properly set then the web configuration directly works with the configuration file of the given Alea instance. If the \texttt{ALEA\_HOME} environment variable is not set, the Alea web configuration works with a local copy of the configuration. In that case, it is necessary to copy this file into the Alea project in order to run the simulator with the changed parameters.

In order to use the configuration web page, the web application needs to be run in a J2EE web container such as Apache Tomcat or GlassFish. It is also possible to run the AleaWebConfiguration project within the NetBeans IDE that includes a web container. Then the web page is opened in the browser, it typically runs at the following URL:

\texttt{http://localhost:8084/AleaWebConfiguration/}

5.2.2 Item Configuration in GUI

When the configuration page is accessed, the configuration file is loaded and the configuration items can be viewed and edited via an HTML page, see Figure 5.4. The page can be easily filled out, because there are
descriptions for each item. There are four columns, the first one indicates the name of the configuration item, then there is the type that the user should respect, then the value of the parameter and the last one is the additional information. This information provides further knowledge about the configuration item, for example: the mathematical formula that will be used if the value is set to true; or, the IDs of algorithms that can be utilized in the simulation. In order to define simulation parameters, the user fills out the Values column. At the end of the web page, there are three buttons. The OK button saves the changes. The Reset button loads the configuration file again, discarding the current changes. The Defaults button loads the default configuration file.

The items are grouped into sections for easier orientation. As shown in Figure 5.4, the first two sections are called Basic configuration and Runtime estimates. By default, there are a total of 9 sections, including, e.g., Data sets and Algorithms.
5.2.3 GUI Implementation

The solution uses JSP. It is a technology for creating dynamically generated HTML pages based on Java programming language. It inserts Java code into HTML code. Thus on the server side, the generated HTML code is inserted into the place of the original Java code. This technology is similar to other server-side scripting languages such as PHP or ASP.

The implementation of the GUI aims at a high level of flexibility and extensibility. The rendering of the page is based on actual Alea configuration and a configuration metadata file: description file. This description file contains additional information on each configuration item, such as its type and description. Thanks to this generic logic, the GUI can handle new configuration items that were not known at the time of creating the GUI. In order to add a new item, only the metadata file needs to be updated, not the code of the web application. The description file also provides means for grouping the configuration entries into sections. This affects the rendering of the configuration page and provides a better orientation while inspecting and setting parameters.

For storing the metadata for each variable, the format of a property file was selected in order to be aligned with the configuration file itself. In the description file, the keys match the configuration entry keys. The values are structured through separating different information by the pipe symbol |. The structure of the line is defined in Figure 5.5.

Figure 5.5: Description File – Line Structure  
key = item type | item label | additional information.

Figure 5.6: Description File Example

HEADER1=Basic configuration
visualize=boolean|Visualize simulation|May slow down the ...
reqs=boolean|Specific job requirements
failures=boolean|Failure trace|If available.
use_queues=boolean|Use several different queues in the sy...
use_speeds=boolean|Machines’ speeds to adjust job executi...
use_heap=boolean|Use heap to store schedule-data|Should b...
HEADER2=Runtime estimates
estimates=boolean|Job runtime estimates
use_AvgLength=boolean|Average job length
use_LastLength=boolean|Last job runtime
use TSAF2R=boolean|TSAF2R’s estimates|If available in t...
5. Configuration System

Figure 5.7: Message Requests

- **HTTP GET**: Reads the configuration and generates an HTML form.
- **HTTP FORM**: Collects the message with parameters and puts them into **AleaConfiguration**. Saves **AleaConfiguration**. Continues as in the HTTP GET case.
- **HTTP POST**: Receives parameters in the message body. Displays, edits, sends the form.

**5.2.4 GUI Request Flow**

The HTTP request flow for the configuration GUI is shown in Figure 5.7. When the user opens the web application in a browser, the browser sends an HTTP GET request to the server. The server manages the JSP code and loads the configuration file. The configuration file is retrieved by using an instance of the **AleaConfiguration** class. Subsequently, the HTML form is generated. Then the form is displayed at the browser side. The configurable parameters of the simulation can be now edited and can be sent via the buttons on the page. When the user clicks a button, the browser sends an HTTP POST request to the server. The configuration items are sent as HTTP parameters in the body of this message. The server collects these parameters and converts them into an **AleaConfiguration** instance. Depending on the pressed button – **OK**, **Defaults**, or **Reset** – the new configuration settings are saved, replaced by defaults, or left intact. Then the server continues with the same code as for the HTTP GET request. These steps are repeated until the page is closed.

**Information** is an optional field. The web application sorts the parameters according to the order of keys in the description file. The value of each item is taken from the configuration file using the same key. Additionally, there is a reserved prefix for description file entries: **HEADER**. These entries define groups of configuration items. All entries underneath a **HEADER** belong to one group until the next **HEADER** occurrence. The beginning of the default description file is shown in Figure 5.6. This snippet corresponds to the screenshot in Figure 5.4.
5.2.5 Logo

In order to provide a better design of the configuration page and future propagation of the simulator, a logo was created. Two options were designed (see Figure 5.8) and one was picked for the Alea web configuration page. This logo is composed of the name Alea and three lines. The lines could stand for the workloads that are on the input of the simulator or it can mean the output stored in files. Or, the three lines can count for the Alea of version 3. Also, it may resemble job queues. Many meanings can be found in this logo. The bottom logo was inspired by the name of the simulator because alea means dice in Latin. And the number three on the dice stands for the version number of Alea. The violet colours are chosen from a complementary colours repository and the orange one is from the original web page of the Alea simulator.

Figure 5.8: Logo Proposals
Chapter 6

Simulation Output, Optimization Criteria as Plugins

One of the identified shortcomings was that Alea stored all simulation outputs into the Alea root directory. Therefore, we have decided to address the task of structuring the outputs into different files and folders according to the used algorithms and data sets.

Another task was to design a replacement of the original solutions for calculating the optimization criteria. Thus the next part of this chapter describes an extensible system that enables a modular support of various monitored metrics.

6.1 Simulation Output

For easier handling of simulation outputs, a new structure for files and folders was designed, see Figure 6.1. Each run of the simulator will generate files into a new folder named by the time stamp of the simulation start. The time format $yyyy-MM-dd-HH-mm-ss$ was selected. The accuracy of seconds allows to launch the simulator multiple times in one minute\(^1\). In this folder, there are 5 files and several subfolders. Each subfolder holds results for one data set used in the simulation. The names of subfolders follow the names of the data sets and the time stamp is appended. The 5 output files are the following: \texttt{Results}, \texttt{RGraphs}, \texttt{SGraphs}, \texttt{WGraphs} and a copy of the configuration file. The configuration data copy provides a snapshot of simulation parameters and is conveniently stored along with the results for archival purposes. In each subfolder, there are folders for each scheduling algorithm used in the simulation, they are named by the name of the algorithm. In these folders, there are the \texttt{jobs} and the \texttt{users} files. The internal structure and the content of the individual simulation output files are described in Section 3.3.

\(^1\) It is unlikely to launch the simulator multiple times in one minute due to the size of the simulation input data (typically hundreds of thousands of jobs).
6. SIMULATION OUTPUT, OPTIMIZATION CRITERIA AS PLUGINS

Figure 6.1: Output Files

```
<table>
<thead>
<tr>
<th>YYYY-MM-dd-HH-mm-ss</th>
</tr>
</thead>
<tbody>
<tr>
<td>configuration.properties</td>
</tr>
<tr>
<td>Results.csv</td>
</tr>
<tr>
<td>RGraphs.csv</td>
</tr>
<tr>
<td>SGraphs.csv</td>
</tr>
<tr>
<td>WGraphs.csv</td>
</tr>
<tr>
<td>dataset_YYYY-MM-dd-HH-mm-ss</td>
</tr>
<tr>
<td>nameOfAlgorithm</td>
</tr>
<tr>
<td>jobs.csv</td>
</tr>
<tr>
<td>users.csv</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
```

6.1.1 Technical Solution

In order to place the files into the desired directory, the Alea output handling code (namely the Output and ResultCollector classes) was injected with additional logic. This logic keeps track of active algorithms and data sets for the purposes of building the correct paths. Appropriately to the category of the output file, the correct path depth is selected and the path segments are named according to the current data set and algorithm, as needed. The new logic is concentrated to the following methods: ExperimentSetup.getDir(int) and FileUtil.getPath(String). The former returns the directory name at the given hierarchy level (0 to 3, i.e., Alea root, experiment root, data set, or algorithm). The latter takes file name on the input and builds the full path according to the file category, taking advantage of the well-defined file naming – see Section 3.3.

6.2 Scheduling Metrics as Plugins

In the original code of Alea 3, there was no support for modularization of scheduling metrics – refer to Section 4.1. The metrics were all computed in the ResultCollector class and they were mixed with other code. This work took the objective to design proper encapsulation of objective func-
tions. Besides encapsulation, other software engineering principles have also been applied in this process, such as design patterns [7].

The new solution applies two design patterns: strategy pattern and observer pattern. The strategy pattern is reflected in defining a family of algorithms – in our case scheduling metrics – and encapsulating each one. The metrics can vary independently from the simulator itself. The observer pattern was utilized when designing the communication between the simulator (more precisely the ResultCollector class) and the objective functions. Each scheduling metric is notified when a job is finished as well as at the end of the whole simulation. An arbitrary number of metrics can be applied.

In the new solution, objective functions are encapsulated into plugins. Each plugin represents one objective function. A new custom metric can be added without modifying the existing code. As explained below, there is even no need to recompile Alea after adding a new metric. Only the metric itself needs to be compiled.

The encapsulation of objective functions is addressed by the creation of the interface Plugin, that each new plugin needs to implement. The interface contains the declarations of three methods:

- init() – loads the configuration of the plugin. This method is invoked immediately after plugin instantiation and plugin configuration data are passed.
- cumulate() – cumulates data for the metric calculation. This code is invoked when a job is finished.
- calculate() – computes the final result of the metric. This method is invoked at the end of the whole simulation.

We have also introduced the AbstractPlugin class. This class provides a default implementation for the method that deals with plugin configuration: init(). It simply stores the incoming configuration into a field and provides a protected getter. It is recommended that custom plugins inherit from this class, overriding is possible as needed. The cumulate() and the calculate() methods represent the implementation of a specific metric and need to be implemented in every new plugin.

On the input of the cumulate() method, there is a ComplexGridlet instance representing one job. The fields of a ComplexGridlet instance are, e.g.: the release date, its priority, the number of used CPUs, RAM usage, or the user ID.
6. Simulation Output, Optimization Criteria as Plugins

The calculate() method has two inputs: a ResultCollector instance and an instance of SchedulerData. The former is one of the core classes in the Alea simulator – see Chapter 3 – and contains several fields with metric-relevant information, such as the total number of jobs. The latter is a new class that encapsulates values produced by the Scheduler class for the purposes of passing the data to ResultCollector and to objective functions. A side effect of the creation of the SchedulerData class is that the structure of the code was improved. In the headers of relevant methods, this object replaces several parameters by one object.

6.2.1 Java Reflection

The instantiation of plugins is taken care of by Java reflection [15]. Reflection enables to modify the runtime behavior of the application. In our particular case, it allows the application to use code – plugins – that is not known at the time of designing the application. In Effective Java by Joshua Bloch [1], the Java reflection’s performance is considered to be much slower than normal method invocation. Furthermore, while using the reflection, the type and exception checking during the compilation is not possible. Due to these facts, we made sure to only utilize reflection to the necessary extent. Plugins are instantiated at the beginning of the simulation only.

The instantiation of plugins takes advantage of Java reflection and is implemented in a new PluginFactory class. The code is generic in the sense that it can create an instance of any Plugin class given the name of the class as a string. The class needs to be on the classpath, but it does not need to be hard-wired or registered into the factory in any way. The class name can either be simple – in that case, the class is assumed to be in the same package as the factory (plugin subpackage of Alea) – or the class name can be fully-qualified, hence a member of any package. Java reflection is only utilized in this factory.

6.2.2 Plugin Configuration

The Alea configuration system – introduced in Chapter 5 – was also enriched by the support for plugin configuration. Alea can have multiple plugins, each with different configuration needs. In order to satisfy these requirements, the plugin configuration was designed to be very generic and flexible. Plugin configuration essentially has two parts. The simpler one provides means to define the list of active plugins, respectively their class names. The more complex one takes care of configuring each of the plug-
ins. We have identified one common parameter which affects how plugin-computed values appear on the output, see details below. Additionally, each plugin can have an arbitrary number of custom parameters. The configuration system can manage a map of parameters for each plugin.

The configuration of the plugins is in the last section of the Alea web configuration, entitled *Scheduling metrics as plugins*, see Figure 6.2. In this section, the first table configures the plugins to be applied during the simulation. Each plugin is entered by its class name. After entering the names of the plugins, the user clicks the **OK** button and the forms for further configuration of individual plugins will be displayed. For each plugin, there is the **Results header** and a table with custom parameters. The Results header field configures the label of the plugin-generated column in the **Results file**, refer to Section 3.3 and Figure 3.2. This column will contain the values computed by the plugin. Below this common configuration item, there is the table with custom parameters. It allows to configure arbitrary plugin-specific key-value pairs. This table is used when the implemented plugin has custom configurable items. For example: If the accuracy of the computation in a plugin is made configurable, the accuracy will be entered as the key and the number of decimal points as the value. The parameters can be added by clicking the **ADD** button and deleted via the **DEL** button.
6. SIMULATION OUTPUT, OPTIMIZATION CRITERIA AS PLUGINS

There are no limits for such parameters. These parameters are available to plugins at runtime, refer to the init() method in Section 6.2.

6.2.3 Implementation of Plugin Configuration

The plugin configuration is stored into the general Alea configuration file described in Chapter 5. Figure 6.3 shows an example of plugin configuration storage and corresponds to the GUI screenshot in Figure 6.2. The list of plugins is stored as a value belonging to the configuration key plugins. The configuration of a single plugin is stored into the configuration file in this format: plugin.n.subkey=value. The index n refers to the plugin index on the list of plugins. Plugins are indexed from zero. The subkey stands for a plugin-specific configuration item. There is one common subkey, that is configurable for all plugins. It is the Results file header and is stored in the following format: plugin.n.result_header=value.

Figure 6.3: Plugins in the Configuration File

plugins=AverageWaitTimePlugin,MyPlugin
plugin.0.result_header=AvgWaitTimePlugin
plugin.1.result_header=My Header
plugin.1.accuracy=2

6.2.4 Example Plugins

In order to test the plugin extension, two objective functions were chosen to demonstrate the usage.

The results were successfully verified against the original ones and are added by default to Results file. The following metrics demonstrate how to use the Plugin interface:

- **Average wait time** – This plugin calculates the average time from the job submission to the beginning of its execution. The cumulate() method contains the implementation of Formula 6.2. And the body of the calculate() method is a Java equivalent of Formula 6.3. The response time is the period from job arrival into the system until job completion – this is shown in Formula 6.1. The CPU time is the total execution time of the job. The job characteristics that are demanded in the cumulate() method, i.e., finish time, CPU time, and arrival time, are retrieved from the ComplexGridlet instance. The average value
is calculated by dividing the total time by the number of jobs. The total number of jobs is acquired from the ResultCollector instance.

\[
response_j = \max(0.0, finishTime_j - arrivalTime_j) \quad (6.1)
\]

\[
WT_{sum} = \sum_{j \in jobs} \max(0.0, response_j - cpuTime_j) \quad (6.2)
\]

\[
WT_{avg} = \frac{WT_{sum}}{|jobs|} \quad (6.3)
\]

- **Average slowdown** – This plugin uses the bounded slowdown metric with the threshold set to 1. This plugin works in a similar way as the average wait time plugin. In the `cumulate()` method, Formula 6.4 is implemented. The `calculate()` method contains the implementation of Formula 6.5.

\[
BSD_{sum} = \sum_{j \in jobs} \max\left(1.0, \frac{response_j}{\max(1.0, cpuTime_j)}\right) \quad (6.4)
\]

\[
BSD_{avg} = \frac{BSD_{sum}}{|jobs|} \quad (6.5)
\]
Chapter 7

Online Source Repository

In order to share this software, we decided to place it on an online source repository. There are many web-based hosting services, such as GitHub, SourceForge, GoogleCode etc. We were choosing from these three as they are the most popular ones. All of these support versioning systems such as Git or SVN. SourceForge also supports CVS and Mercurial. GoogleCode however has its limits, the service is for OSI-approved open source projects only and the number of projects per person is limited to 25, each with a 5GB total size limit. GitHub has 1GB limit per repository, but it can be negotiated. SourceForge was the first to offer the source code repository for open source projects for free and has been for a long time the best known site. But GitHub overtook SourceForge’s first place and announced 10 million repositories and more than 5 million users in December 2013 [2]. SourceForge has about 300 000 repositories and 3.4 million users [16]. GitHub and SourceForge have similar functionalities but SourceForge is only for open source projects, GitHub has the paid possibility of private repository. GitHub offers social networking functionality such as followers as well as network graphs with the traffic – an overview of visits and commit activity over the year. It provides wikis, code review functionality and bug tracking.

7.1 Alea on Github

GitHub was selected as the online source repository for Alea. It uses Git [5] as the version control system.

Figure 7.1 shows the Alea simulator on GitHub. Alea can be now downloaded from https://github.com/aleasimulator/alea. Alea files can be retrieved by several options. It is possible to simply download Alea project files in the ZIP format. This is accomplished by clicking on Download ZIP on the Alea’s GitHub page. GitHub provides read and write access for collaborators or a read-only clone. These possibilities are explained in Appendix A.
The project on GitHub provides a README file that is displayed on the main page of the project. It is a means to write the major information about the software. The Alea’s README contains the main focus of the simulator and describes the simulator functionalities. Also there is the information about the libraries that the project depends on. Then, there is a link to sample data inputs (workloads). The project does not include the data sets directly, due to their size. Finally, the software license is attached and the authors are mentioned.

Each project on GitHub can have wikis. To facilitate the usage of Alea, various instructions were added into the wikis. The first wiki page is called How to run the simulator, it contains the information on setting up the application and managing its configuration. Then there is How to add a new objective function and How to add a new scheduling algorithm.
Chapter 8

Conclusion

8.1 Summary

In this work, several principles of modern software engineering have been applied to Alea. These modifications enhanced extensibility and modularization. The observer pattern and the strategy pattern [7] are applied within the plugin extension. One plugin represents one optimization criterion. New optimization criteria can be now added easily as new plugins. This extension – unlike the original solution – does not need to be compiled into the Alea code.

The next part of the extensions was the configuration of the simulation parameters. According to software engineering principles, the separation of the configurable items from the source code has a positive impact on the maintainability of the simulator. The separation came together with some elimination of code duplication. Previously, the configuration was done by modifying certain variables and code portions which was a fragile solution. Thus, a code revision was performed. The configuration items of the simulation are now stored in the configuration properties file and the routines for its usage were implemented. The new solution also includes means to configure optimization-criterion plugins including their custom parameters.

For added convenience, a graphical user interface for the configuration subsystem was implemented. This interface takes the form of a dynamic web page, thus the user accesses the configuration via a web browser. In order to provide a better orientation, the configuration items can be split into several categories. A default categorization is provided and can be customized by editing a metadata file. The solution is very flexible. Neither the configuration subsystem nor its web interface need to be modified when adding a new previously unknown configuration item.

Besides these larger changes, other small improvements were implemented. One of these tasks aimed at creating self-documenting outputs. The output files were structured into various levels in the hierarchy of
file paths. The individual simulation results are now easy to find without searching through all output files.

With the implemented changes, Alea was uploaded to an online source repository: GitHub. The GitHub repository is quite easy to work with, is modern and has an increasing popularity for a long time. It also supports the creation of online documentation and hence allows the placement of instructions for simulator usage.

The implemented improvements represent a substantial step in the development of the Alea simulator. The enhancements are based on the demand of Alea users and developers and address the shortcomings that were making daily work with this software difficult. The Alea code was extended and partly also restructured in order to ease future maintenance. These changes took advantage of standard as well as modern constructs of the Java Standard Edition and Enterprise Edition programming language. General software engineering principles and patterns were also applied.

8.2 Suggestions for Future Work

Depending on the practical evaluation of the new reflection-based approach for simple inclusion of new scheduling metrics, we should consider applying the same approach to the scheduling algorithms. This would further enhance the modularity of Alea.

When trying out different data sets and algorithms, the simulation often takes a relatively long time. Currently, Alea uses only a single CPU core for the most of the execution time. In the future, Alea could be better parallelized in order to take advantage of the full power of modern hardware. For instance, the scheduling algorithms or the computation of objective functions could be parallelized. Also, the file reading and writing functionality could be revised towards a higher efficiency, for instance by using non-blocking IO.

Regarding the ease of use, some next steps could be done to make setting Alea up more comfortable. For example, a local J2EE web server can be added to the distribution in order to allow an easy access to the configuration system without having to install any additional software. For a fully comfortable user experience, Windows and Linux installers could be created to facilitate the deployment on the most popular operating systems.
Bibliography


Appendices
Appendix A

GitHub Usage

**GitHub - r/w access for collaborators**
2. Launch GitHub via icon in the Start menu.
3. Sign in / Sign up at the welcome screen (Use <username>@users.noreply.github.com as the email address in order to keep it private).
4. Select aleasimulator underneath GitHub and then create a local clone of the alea repository.
5. Modify files as needed.
6. Open the local repository and use Commit and Sync buttons to upload the modified content.
7. Collaborators can be added via https://github.com web interface (Settings - Collaborators).

**GitHub - read-only clone**
2. Launch Git Shell via icon in the Start menu.
3. Type:
   ```
   git clone https://github.com/aleasimulator/alea.git
   ```
4. Launch GitHub via icon in the Start menu.
5. Open local repositories tab.
6. Locate the root folder of the cloned repository, typically:
   ```
   C:\users\<username>\My Documents\GitHub\<project>
   ```
   Drag and drop it onto the Github window.
Appendix B

Source Code

The contribution of this work comprises of modifications of Alea source code. The full Alea source code with these changes is available at GitHub: https://github.com/aleasimulator/alea – refer to Appendix A. The state described in this thesis reflects the repository on May 15, 2014. Additionally, the appropriate snapshot of the Alea project is also stored in the Archive of Masaryk University Information System, which is available at http://is.muni.cz/thesis/.

The front page of the GitHub web interface as well as the README file in the snapshot contain the instructions for use.