UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

TAIS LOUREIRO BELLINI

Extensible Simulator for Replay of Trace Files in the Pajé format

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Science

Advisor: Prof. Dr. Lucas Mello Schnorr

Porto Alegre
May 2016

ABSTRACT

Observation of program behavior through trace files is particularly important in High Performance Computing (HPC) since it enables an accurate performance analysis. In this context, the Pajé framework is commonly employed to simulate trace files in the Pajé file format. The simulator works by recreating the program behavior in an offline fashion, allowing the performance analyst to better understand performance issues that might have occurred during execution. As of today, there are at least three issues with the Pajé simulator: little extensibility, lack of partial outcomes and impermanent results. Although Pajé has been built in an extensible manner, it is necessary to write a full component for Pajé to actually achieve extensibility. This is a complex task since it implies in understanding the internal class hierarchy of the framework. The second issue is that it is impossible to get a partial view of simulated data prior to the end of the input trace. And finally, the third is related to the ephemerality of results: all simulated data is discarded once Pajé exits. To address these issues, an extensible plugin-based trace file simulator called Aiyra has been designed and implemented. Aiyra's objective is to allow extreme extensibility by letting the performance analyst write plugins that deal with simulated data. The plugins can be attached to the simulator in specific and important points where the trace events are combined. As a proof of concept, we have implemented plugins to dump partial data that has just been simulated and to make the results permanent, by inserting simulated data into a database. A performance analysis has been conducted to compare Aiyra against the existing Pajé trace simulator. Our simulator presented better performance results with larger files, that being possibly attributed to the fact that our solution keeps the memory footprint low throughout execution. We have also evaluated the database plugin in a number of different scenarios through the use of a rigorous experimental design.

Keywords: Trace simulation. Replay. Pajé. JavaCC. MySQL. Performance.

Simulador extensível para arquivos de rastro no formato Pajé

RESUMO

Observação do comportamento de programa através de arquivos de rastro é particularmente importante em Computação de Alta Performance, uma vez que permite que uma análise de desempenho precisa. Neste contexto, a ferramenta Pajé é bastante utilizada para simular arquivos de rastro no formato de arquivo Pajé. O simulador funciona, recriando o comportamento do programa em modo *offline*, permitindo que o analista de desempenho entenda melhor problemas de desempenho que possam ter ocorrido durante a execução. Atualmente, há pelo menos três problemas com o simulador Pajé: pouca extensibilidade, falta de resultados parciais e resultados temporários. Embora o Pajé tenha sido construído para ser extensível, é necessário escrever um componente inteiro para conseguir realmente extensibilidade. Esta é uma tarefa complexa, pois implica em compreender a hierarquia de classes interna do programa. A segunda questão é que é impossível obter uma visão parcial do dados simulados antes do final do arquivo de entrada. E, finalmente, o terceiro problema está relacionado com a efemeridade dos resultados: todos os dados simulados são descartados quando o Pajé termina de executar. Para resolver estas questões, um simulador extensível baseado em *plugins* chamado Aiyra foi concebido e implementado. O objetivo da Aiyra é permitir extrema extensibilidade deixando o analista de desempenho construir *plugins* que lidam com os dados simulados. Os *plugins* podem ser conectados ao simulador em pontos específicos onde os eventos de rastreamento importantes são combinados. Como prova de conceito, implementamos plugins para mostrar na saída padrão dados parciais que acabaram de ser simulados e para tornar os resultados permanente, inserindo os dados simulados em uma base de dados. Uma análise de desempenho foi conduzido para comparar Aiyra com o simulador Pajé existente. Nosso simulador apresentou melhores resultados de desempenho com arquivos maiores, o que foi atribuído ao fato de que o nosso solução mantém o consumo de memória baixo ao longo da execução. Nós também avaliamos o *plugin* de inserção em uma base de dados para diferentes cenários através de um rigoroso *design* experimental.

Palavras-chave: Pajé.Rastro.Simulador..

LIST OF FIGURES

Figure 2.1 JavaCC's file generation flow 13
Figure 2.2 Architecture of JDBC. [Inspired in (POINT, 2016b)] 14
Figure 2.3 Example of ER Model 15

Figure 3.1 Example of Entities Hierarchy 21
Figure 3.2 PajeNG Architecture [inspired in (KERGOMMEAUX; STEIN; BERNARD, 2000)] 22
Figure 3.3 Events class hierarchy 23
Figure 3.4 Events class hierarchy 24
Figure 3.5 Entities class hierarchy 25

Figure 4.1 Aiyra Architecture 29
Figure 4.2 Aiyra's Core Architecture 31
Figure 4.3 Aiyra's Plugin Package 33

Figure 5.1 Aiyra's Dump Plugin 35
Figure 5.2 Aiyra's Insert DB Plugin 36
Figure 5.3 ER Model for the Pajé format 38

Figure 6.1 Network topology 42
Figure 6.2 Results of comparison between Aiyra and PajeNG 44
Figure 6.3 Results of comparison between Aiyra and PajeNG for the small input 46
Figure 6.4 Results of batch sizes variability for big input 48
Figure 6.5 Results of batch sizes variability for medium input 49
Figure 6.6 Results for remote and local executions for the big input 49
Figure 6.7 Results for remote and local executions for the medium input 50
Figure 6.8 Results of batch sizes variability for the small input 50
Figure 6.9 Results of batch sizes variability for the big input 51
Figure 6.10 Results of batch sizes variability for the medium input 51
Figure 6.11 Results of batch sizes variability for the small input 52
Figure 6.12 Insertion time for the big input 53
Figure 6.13 Insertion time for the medium input 53
Figure 6.14 Insertion time for the small input 54
Figure 6.15 Remote and local insertion times for the big input 55
Figure 6.16 Remote and local insertion times for the medium input 55
Figure 6.17 Results of batch sizes variability for the medium input 56
Figure 6.18 Results of batch sizes variability for the medium input 56
Figure 6.19 Results of batch sizes variability for the small input 57

LIST OF TABLES

Table 6.1 Experimental Units description 41
Table 6.2 JVM heap sizes in bytes 41
Table 6.3 Average Execution Time (s) for each configuration 45

LIST OF ABBREVIATIONS AND ACRONYMS

GC	Garbage Collector
CSV	Comma Separated Values
JVM	Java Virtual Machine
DBMS	Database Management System
JDBC	Java Database Connectivity

CONTENTS

1 INTRODUCTION **9**
2 BASIC CONCEPTS **12**
2.1 The Java Compiler Compiler (JavaCC) tool **12**
2.2 JDBC and MySQL **13**
2.3 Entity-Relationship and Relational Models **14**
2.4 Experimental Design **16**
2.5 The R language **17**
3 THE PAJENG FRAMEWORK **19**
3.1 The Pajé Trace File Format **19**
3.1.1 Header section: events definition 20
3.1.2 Body section: timestamped events 20
3.2 PajeNG Tools and Simulation Library **21**
3.2.1 Class hierarchy for Paje events 23
3.2.2 Class hierarchy for the Paje types 24
3.2.3 Class hierarchy for the Paje entities 24
3.2.4 The core simulator 25
3.3 Current Issues Regarding PajeNG **26**
4 AIYRA - A JAVA-BASED SIMULATOR FOR PAJE TRACE FILES **28**
4.1 The controller: option handling and JavaCC **28**
4.2 Aiyra's core simulator **29**
4.3 The plugin package **32**
5 AIYRA'S STANDARD PLUGINS **34**
5.1 Paje Null Plugin **34**
5.2 Paje Dump Plugin **34**
5.3 Paje Insert Database Plugin **35**
5.3.1 Entity-Relationship Model 37
5.3.2 Relational Model 38
6 PERFORMANCE EVALUATION **40**
6.1 Methodology **40**
6.1.1 Aiyra and PajeNG Comparison Methodology 41
6.1.2 PajeInsertDBPlugin evaluation 42
6.2 Results and Graphics **43**
6.2.1 Comparison between Aiyra and PajeNG 43
6.2.2 PajeInsertDBPlugin Evaluation 46
7 CONCLUSION **58**
REFERENCES **60**
APPENDIX A — JAVACC TUTORIAL **62**
A.0.1 Usage with Java 65

1 INTRODUCTION

Observation of program behavior is particularly important in High Performance Computing since it enables an accurate performance analysis. A very common method of evaluation is registering program events in trace files that work as a log, so, whenever a relevant event happens, it will be logged in the trace file with the proper information. Then, these traces are replayed in simulation combining information that is spread across multiple events deriving new and richer entities. This event processing is usually done once and discarded, which can be very inconvenient considering that some trace files are very large. Therefore, it would be interesting to save these entities for further analysis.

In this context of performance analysis, a common tool used is the Pajé Trace Simulator (KERGOMMEAUX; STEIN; BERNARD, 2000), an open source project that reads trace files in a specific format, the Pajé Trace File Format (SCHNORR, 2016a), and processes five types of entities: Containers, States, Events, Variables and Links. These concepts represent basic structures of computer programs executed in parallel or distributed systems, such as processes, threads, network links, hardware counters and so on. As a base for the implementation of this proposal, the new generation of Pajé, PajeNG (SCHNORR, 2016b), was used. Details of PajeNG, the types of entities and the Pajé File Format will be presented in Chapter 3. Although it has a visualization functionality, we will focus on the simulation part of the Pajé implementation in this project. The most commonly used tool of PajeNG is the Pajé Dump[1] (`pj_dump`), that dumps to the standard output all information regarding entities created throughout the simulation.

There are at least three problems with the current implementation of Pajé: little extensibility, lack of partial outcomes and impermanent results. The original Pajé and, consequently, its next generation were idealized and built in an extensible way, that could be easily expanded based on the user's needs. Although the implementation is very modular, it is necessary to write a full component for Pajé, which is a complex thing because it implies in understanding the internal objects, class hierarchy, protocol, and so on. Until now, very few people have actually extended this tool. Throughout the simulation, the Pajé tool creates entities according to the events listed in the trace file, saving each one of them in memory

[1] http://github.com/schnorr/pajeng/

to dump everything at once in the end. Since some trace files can be very large (over 1 Gigabyte), it may take a while for the results to be printed out. Besides not being able to have a partial view of already simulated entities, the user won't have records of the results between executions for different files unless he specifies a destination for it himself. To address these issues, an extensible trace files simulator, Aiyra[2], was developed in Java. The full code is publicly available on Github.

The objective of this proposal is to allow the performance analyst to change the simulator behavior when a new entity is detected. Thus, the partial results can be immediately presented to the user, or saved in a database, or even discarded if not relevant. This extensibility is implemented through the concept of plugins that are attached to the simulator in specific and important points where the trace events are combined. This main objective solves the lack of extensibility problem. Once the simulator allows the immediate manipulation of the entities, the other two issues can be easily addressed with extensions. Hence, the secondary objectives of the project are the creation of plugins to dump partial data that has just been simulated and to make the results permanent.

For the validation of the extensible trace files simulator, two plugins were implemented: Paje Dump Plugin and Paje Insert Database Plugin. The first one plays the same roll as the original Paje Dump tool in the Paje new generation, with the difference that the entities are dumped at the moment they are completed. The second one inserts all the data in a relational database. A specific schema for the Pajé Format was designed and will be presented in Chapter 5. A performance analysis was developed to compare Aiyra against the previous one. It is worth highlighting that the new simulator had better performance results with bigger files (over 120 Megabytes), that being possibly attributed to the fact that it discards from memory entities that will no longer be used. Additionally, the Pajé Insert Database plugin was evaluated comparing its different possibilities of usage. In this investigation, we varied the frequency of the insertions in the database by grouping queries in memory until it had a specific size to insert. The objective of this test was to understand the impact of an access to a database in the performance of the program. Likewise, the usage of the memory was also examined to determine the best balance between execution time and memory

[2]http://github.com/taisbellini/aiyra/

management. As we will see in Chapter 6, the memory usage had more impact in the performance than the accesses to the database themselves, probably due to the Garbage Collector mechanism used by Java.

This document is organized as follows. Chapter 2 provides the basic concepts of the technologies used throughout the project. Chapter 3 gives an overview of the existing simulator for the Pajé format and the problems with this current implementation. Chapter 4 describes the details of the extensible simulator developed in Java and Chapter 5 characterize the plugins developed. Chapter 6 presents a performance analysis of the Pajé Insert Database Plugin and a comparison between Aiyra and PajeNG. The conclusion and final considerations are expressed in Chapter 7.

2 BASIC CONCEPTS

This chapter describes the basic notions of the concepts and technologies used to develop this project. They contribute to the understanding of our work. It is structured in five topics: the Java Compiler Compiler (JavaCC), the Java Database Connectivity (JDBC) API in the MySQL context, a brief description about the construction of conceptual and logical database schemas, an overview about experimental design, and the R language in the experimental design context.

The Java Compiler Compiler (JavaCC) tool

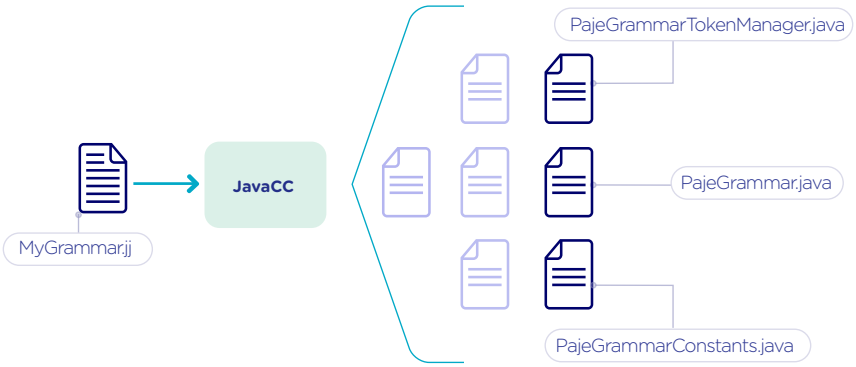
The Java Compiler Compiler (JavaCC) (JAVACC, 2016) is a scanner and parser generator configured with a set of regular expressions describing the tokens of a language and a grammar using these tokens. As output, it generates a lexical and syntax parser in the Java language (HAMILTON, 2016). The lexical code separates the input file into tokens; the parser code is responsible for the syntax analysis.

What differentiates JavaCC from other parser generators that exist for the Java language is that it creates source code in Java. This facilitates the understanding and eliminates the need of having dependencies in the code. JavaCC has also shown itself to have a much better performance than other tools such as Another Tool For Language Recognition (ANTLR), that requires a runtime library (CRAWLEY, 2015). ANTLR was our first choice of parser generator, but it was soon discarded due to its very low performance.

JavaCC can be downloaded, unzipped and added to the PATH. It also has a plugin for Eclipse. Figure 2.1 exemplifies the flow in JavaCC. Once installed, JavaCC processes your grammar defined in a file with extension `.jj` using the command `javacc`. The whole grammar is kept in this file and it is the only file that needs to be modified. It is also possible to add Java code that has to be executed during the parsing. A detailed tutorial about the JavaCC is in Appendix A.

In Figure 2.1 we can see an example of the processing of a file named **MyGrammar.jj**, which results in seven files: the parser in **MyGrammar.java**; the lex-

Figure 2.1: JavaCC's file generation flow



ical analyser in **MyGrammarTokenManager.java** and some useful constants in **MyGrammarConstants.java**. The other four files generated: **Token.java**, **TokenMgrError.java**, **SimpleCharStream.java** and **ParseException.java** are boilerplate files that can be reused within parsers and are not affected by the grammar itself. The corresponding Java source code for the scanner and parser can be compiled as usual with `javac`.

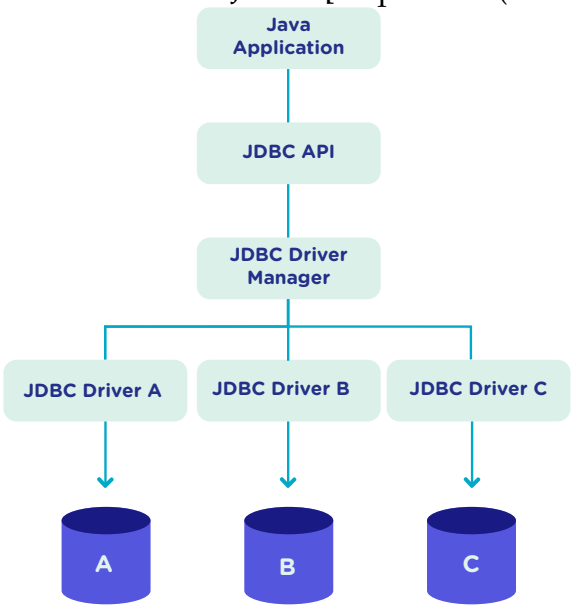
JDBC and MySQL

The Java Database Connectivity (JDBC) API is a standard for connectivity between Java and a range of relational databases (ORACLE, 2016). It comprises methods to query and update data, enabling the Java language to interact with several Database Management Systems (DBMS) in a standard manner. This API facilitates the migration from one database tool to another and unbounds your application from a DBMS.

The JDBC architecture, depicted in Figure 2.2, consists in two layers: JDBC API and JDBC Driver API. JDBC can support multiple heterogeneous databeses (POINT, 2016b) by using drivers connected to them. In the example of Figure 2.2, we have an application communicating with three different databases: A, B and C. The JDBC Driver API manages the corresponding drivers to ensure that the correct one is being used. The JDBC API layer, in turn, administrates the communication between the application and the driver manager. The JDBC API consists in classes and interfaces, such as **DriverManager**, which makes a connection between requests from the application and the proper database driver; **Connection**, containing all the methods necessary to contact the database; **State-**

ment, that creates objects that will be submitted to the database; and **ResultSet**, where the retrieved objects are placed.

Figure 2.2: Architecture of JDBC. [Inspired in (POINT, 2016b)]



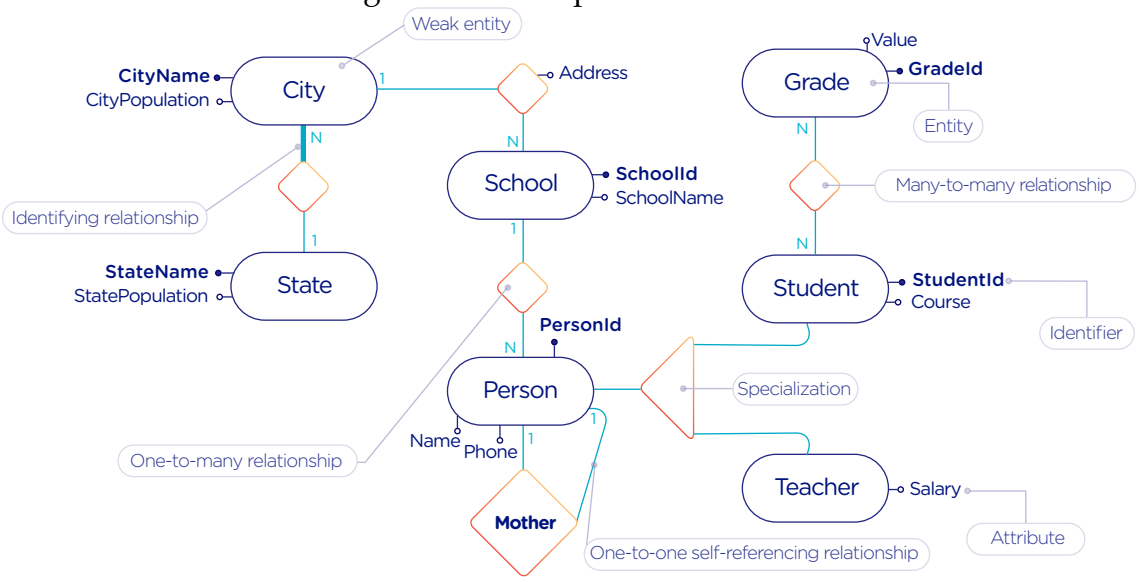
Among the possible DBMS that can be used in a Java application using JDBC is the MySQL system, one of the most important open-source DBMS in the market. It has been developed by Oracle and uses SQL language as interface. To enable the use of MySQL with Java, Oracle provides a driver for JDBC, as well as a native C library to allow developers to embed SQL commands directly in the application's code.

Entity-Relationship and Relational Models

An Entity-Relationship (ER) model defines a database in a conceptual view (HEUSER, 2008). This model can be represented by an ER Diagram (ERD) and can be denoted by **schema**. It is based in the notion of **entities**, which can be real-world objects that are easily identifiable (POINT, 2016a), and the **relationship** between them. Figure 2.3 exemplifies a schema of a school system where the people and places involved are represented. The entities have a set of attributes, where one or more are defined as the **identifier**, which uniquely identify an object of that entity. It is also possible for a relationship to have attributes, like the **address** attribute in Figure 2.3. Besides, an entity can derive other more specialized entities, which is called **specialization**.

A relationship between entities **A** and **B** can have one of the following patterns: **one-to-one**, where an object of the entity A can be associated to only one of type B and vice versa; **one-to-many**, which means that an instance of the entity A can be associated to more than one entities of type B, but B entities can only relate to at most one of type A; **many-to-many**, where one object from the A entity can be associated to more than one entities of type B and vice versa. These characteristics also apply to self-referencing relations, where there is a relationship of an entity with itself. Besides, a connection can be an **identifying relationship**, which means that the relationship identifies an object. In Figure 2.3 we can see the example of the relationship between a **City** and a **State**, where we define that a state can have more than one city, while a city belongs to only one state. Also, the state identifies a city along with its name, since there can be other cities with the same name but in different states. In these cases the entity is called a **weak entity**.

Figure 2.3: Example of ER Model



The ER Model is an overview of the structure of a database. To evolve to the implementation, a translation to a Relational Model must be performed. The ER model is a conceptual description of the database, while the relational model is a logical representation (HEUSER, 2008). Relational databases are based in the concept of **tables** (BARRY, 2013), thus, the terminology used in this step involves **tables**, **rows** (or **tuples**) and **columns**. The identifier is called **primary key**. There is a set of rules to make this translation, although sometimes it is necessary to

adapt the schema based on the user's needs.

The **entities** of the ER model become **tables** in the relational model, while its attributes become the **columns** of the tables. Each instance of an entity is a row and its identifier can be one or more columns that compose the **primary key**. When there is an identifying relationship, the weak entity attaches the identifier of the other entity to its own primary key. In Figure 2.3, the **City** table has two columns as primary key: **stateName** and **cityName**. A **one-to-one** relationship generates a merge of both tables involved. **One-to-many** links adds an attribute in one of the tables. The entity that can only be related to one of the instances of the other receives the attribute, which is called **foreign key**. It also absorbs the attributes of the relationship. In the example of figure 2.3, the table referencing the **School** entity would have the **stateName** and **cityName** as foreign keys and **address** as attribute. The foreign key is what preserves the link between two entities. **Many-to-many** relationships have to be represented by a separate table with the primary keys of both entities involved, which also work as foreign keys. In our illustration, the relationship between **Grade** and **Student** would become a table, with **studentId** and **gradeId** as primary and foreign keys.

The translation of specializations can be made in two ways: one single table for all of the hierarchy, where the primary key would be the identifier of the most generic entity and there would be optional columns; and one table for each specialized entity, where all of them would have the identifier of the most generic entity composing the primary key. In Figure 2.3, the first case would generate a single **Person** table, with **personId** as primary key and **name**, **phone**, **studentId**, **course** and **salary** as attributes. In the second option, **Student** and **Teacher** become tables, with **personId** composing their primary key.

Experimental Design

Experimental design, in the context of performance analysis, aims to define a minimum number of experiments that collects the maximum information necessary (JAIN, 2015). It also targets random variations that could affect the results, guaranteeing that the number of tests executed and the error margin calculated is sufficient to avoid misleading conclusions.

There is a specific terminology used in experimental design. The term **Re-**

sponse Variable is the outcome of an experiment; **Factors** are all of the variables that can have several different values affecting the response variable, and **Levels** are the possible values that a factor can assume. Also, the **Primary Factors** are the factors that need to be quantified, **Secondary Factors** are the factors whose impacts in the performance are irrelevant for the analysis, **Replication** is the number of repetition of all or some experiments and **Design** is the specification of total number of experiments based on factor level combination and number of replications for each experiment. The **Experimental Unit** is the entity used for the experiment, which could be a computer, for example, and **Interaction** is when the levels of a fator affect the results of other factor.

There are several types of experimental design modeling. One of them is the full factorial design, which consists in evaluating every possible combination at all levels of all factors. With this type of design, it is possible to measure factors with multiple numbers of levels. The advantage of this model is that every possible combination is measured, generating richer results. However, depending on the number of factors, levels and replications, it may generate too many experiments, which can cost a lot of time. Therefore, when using this technique, it is important to weight the relevance of each factor and level to generate an appropriate and accurate design. To calculate the total size of the sample you multiply the numbers of levels of the factors and the number of replications (SAS, 2016). For example, a design with a three-level factor, a two-level factor and 20 replications would have 120 experiments (the result of $3 * 2 * 20$).

When there are too many factors and levels, it may not be possible to use the full factorial design. In these cases, one can use a fractional factorial design, which covers just a fraction of the full factorial design. In this type of experiment, a carefully chosen subset of factors and levels is taken into consideration, based on the most important features the analyser wants to test. Although it saves time and expenses, the results provide less information.

The R language

R is a language for statistical computing and graphics generation. It can be very easily extended, by creating and using packages. With R, it is possible to create full factorial or fractional designs using the **DoE.base** package. It con-

tains the class **design** with several accessor functions to create different types of design. One particular important function is the **fac.design**, which creates full factorial designs with an arbitrary numbers of levels. The function receives several arguments, including number of factors, levels and replication. The usage of the function is the following:

```
require(DoE.base);
fac.design(
  nfactors=2,
  replications=30,
  repeat.only=FALSE,
  blocks=1,
  randomize=TRUE,
  seed=10373,
  nlevels=(3,6),
  factor.names= list(
                input=c("small", "medium", "big"),
                batch=c("A", "B", "C", "D", "E", "F")));
```

where `nfactors` represents the number of factors, **replications** is the number of replications, **repeat.only** tells if the replications of each run are grouped together, **blocks** is a prime-number telling in how many blocks the experiment is subdivided, **randomize** informs the design is randomized, **seed** is the optional seed for the randomization, **nlevels** is a vector with the number of levels for each factor and **factor.names**: a list of vectors with factor levels. This example is one of the designs used for the performance evaluation in Chapter 6.

3 THE PAJENG FRAMEWORK

The Pajé Visualization Tool is an implementation to display the execution behavior of parallel and distributed programs. It reads information from trace files that describe the important events during the execution of a program and replays them through simulation. It has been developed to simulate trace files in the Pajé Trace File Format, thus, it is important to understand how the Pajé trace files are composed. Section 3.1 describes this format and all entity types it contains. The next section describes the new generation of the Pajé Visualization Tool, the PajeNG, focusing on the **libpaje** module, which is where the core simulation is performed.

The Pajé Trace File Format

The Pajé Trace File Format (SCHNORR, 2016a) is a textual and generic pattern that describes the behaviour of parallel and distributed programs. The Pajé format describes five types of entities: containers, states, events, variables and links. Each entity is always associated to a container, even the containers themselves. A **container** can be any hardware or software entity, such as a processor, a thread, a network link, etc. It is the only Pajé object that holds other objects, including containers, which makes it the main component to define a type hierarchy. A **state** is used to describe periods of time where a container stays at the same state, like a thread that is blocked, for example. It always has a beginning and an ending timestamps. An **event** has only one timestamp, and can be anything noteworthy to be uniquely identified. A **variable** entity represents the progression of a variable's value along time. It is represented by an object with a value and two timestamps, beginning and end, indicating how long the variable had that specific value. A **link** represents a relationship between two containers, such as a communication between processes. It contains two timestamps specifying the beginning and the end of the communication. A Pajé trace file is divided in two segments: event definition and timestamped events. A brief description of these sections is provided below.

Header section: events definition

The first part of a trace file describes all of the possible events of the trace. This part is composed by a block where the first line contains the name of the event, like **PajeDefineContainerType**, for example, followed by a unique identifier. The identifier is an integer and will be used later by the events to determine the type of event in question. After, there is a set of fields, one in each line. Each field comprises a name, and a type. The type of a filed can be a string, double, int, date, hex or color.

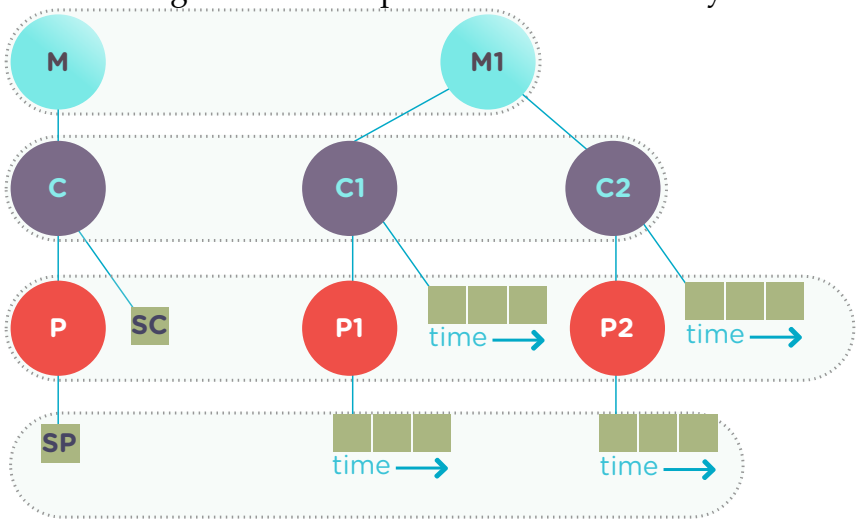
Body section: timestamped events

After the events definition, the events themselves are described, one in each line. Every event starts with its identifying number, which was defined previously, followed by the fields separated by space or tab. Before the entities, such as states or links, can be created, a hierarchy of types and containers must be defined and containers need to be instantiated, since every entity belongs to a container. There are sets of events associated to each kind of entity described above, besides the events that define entity types.

The Pajé objects are organized in two separated hierarchies: types and entities. These hierarchies are specific for each trace file, although it can be repeated in traces with the same scenario. In the structure of the trace file, the type hierarchy comes just after the event definition. There, each type of the program is defined and one of the fields is always the parent type. Each entity is always associated to a type and they must follow the same precedence as the types definition. In Figure 3.1, we have a type hierarchy on the left, and a corresponding entities hierarchy on the right. The only kind of entity that holds other entities is the container, thus, the rounded entities are container types, while the squared ones represent states. On the right, the **M** type stands for machine, **C** is core, **P** is process and **Sp** and **Sc** are states. On the left, we have one machine, **M1**, with two cores: **C1**, running a process **P1**; and **C2**, running a process **P2**. The processes and the cores have a stack of states organized by the timestamp. Notice that the entities tree respect the precedences set on the left. The difference between both hierarchies relies on the number of nodes: while the type hierarchy has only a

few, the entities hierarchy may have millions depending on the number of containers in the trace.

Figure 3.1: Example of Entities Hierarchy



Type definition events do not have a timestamp field and can occur at anytime in a trace file, as long as the type is not used before its definition. It is more common to have all the types defined in the beginning. The events associated to the containers are timestamped and can create or destroy instances during the trace file. A container cannot be referenced after it was destroyed. Variables can be set at a specific timestamp and have its value changed throughout the simulation by addition and subtraction events. The value of a variable is a double precision floating-point number, which is different from the values of the other entities. A variable must be set before changes to its value can be made.

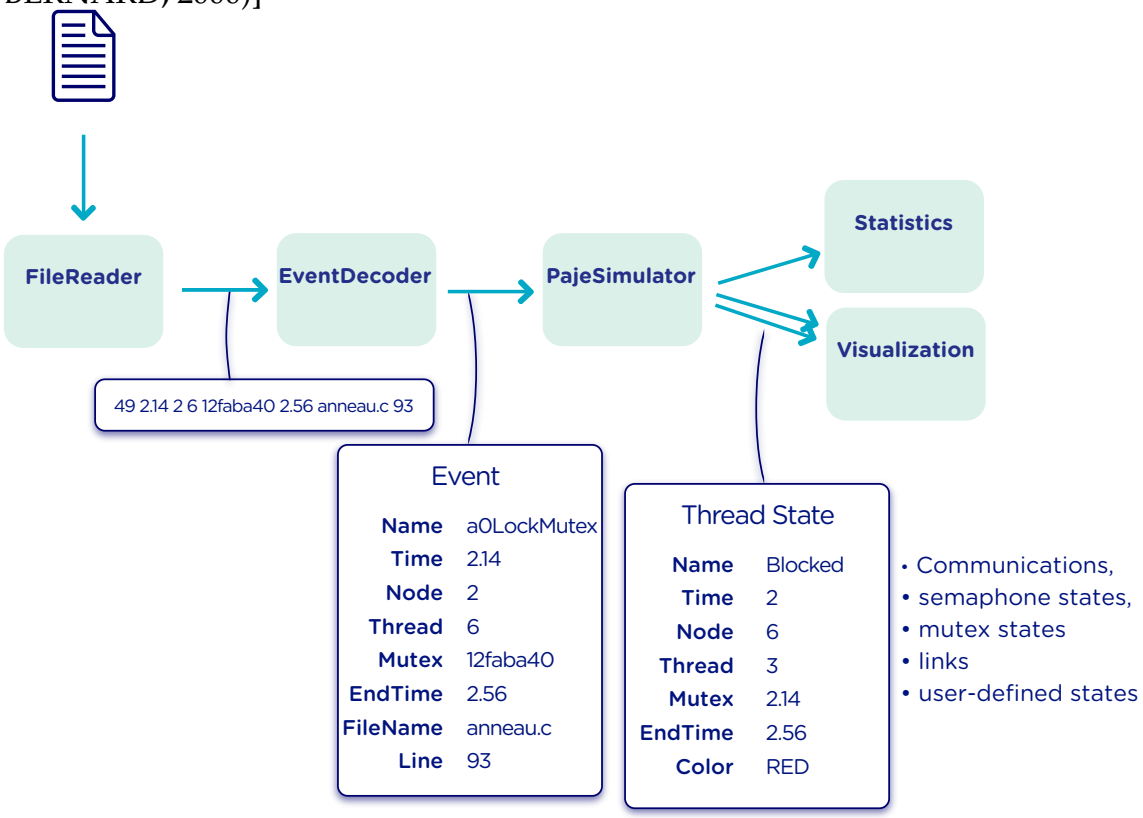
PajeNG Tools and Simulation Library

The PajeNG implementation is the new generation of the Pajé Visualization Tool (SCHNORR, 2016b). It was developed in C++ and follows the same architecture as the original Pajé, written in Objective-C. It comprises a library containing the core of the simulation (**libpaje**), a space-time visualization tool and some auxiliary tools to manage the trace files. The base for the implementation of this project was the **libpaje** library.

The library, represented in Figure 3.2 has three main components forming a pipeline that results in complete simulated entities. These components are: **FileReader**, **EventDecoder** and **PajeSimulator**. First, the **FileReader** reads a chunk

of data from the trace file and puts it in memory. Then, the **EventDecoder** breaks it into events identifying, line by line, the event's fields and creating an object with all the necessary information. Last, the **PajeSimulator** receives this event object and addresses to the proper simulation.

Figure 3.2: PajeNG Architecture [inspired in (KERGOMMEAUX; STEIN; BERNARD, 2000)]

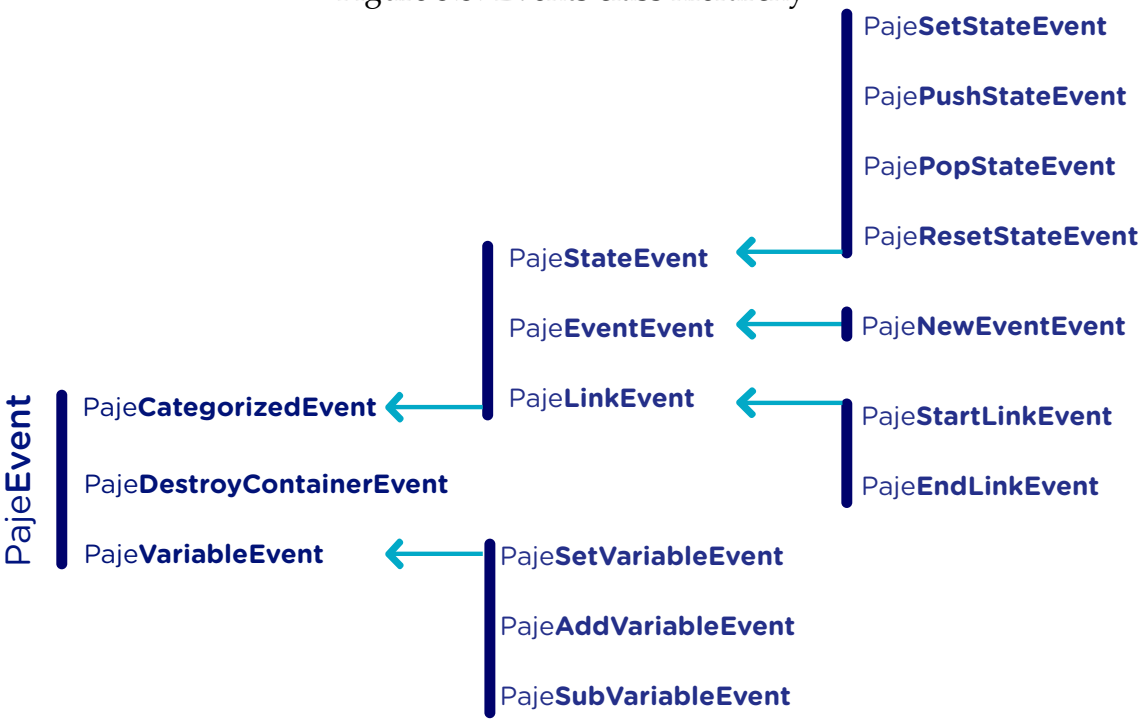


Pajé was idealized to be extensible, specially in terms of creating new types of events. Actually, the Pajé format itself is very expandable, which makes it necessary to build a simulator accordingly. This flexibility is implemented by a class hierarchy, going from the most general, containing the basic fields common to every type and entity, to the most specific. Besides, the PajeNG tool supports extra fields in the events, which allows the simulation of extended entities. There are three main class hierarchies that are particularly important in this objective: one for events, one for types and one for entities. With this modular implementation, it is relatively easy to add a new type of event or entity and integrate it with the rest of the code.

Class hierarchy for Paje events

An event object is what is passed as an argument to the simulator so that it can be processed. Therefore, it must contain all of the necessary information for the simulation. The first object created when a trace file is being parsed is of type **PajeTraceEvent**, which is a class containing all the fields read by the **EventDecoder**. As depicted in Figure 3.3, the event hierarchy starts with a simple **PajeEvent** class. This class has a trace event object, a container, a type and a timestamp. The immediate childs of PajeEvent are: **PajeCategorizedEvent**, **PajeVariableEvent** and **PajeDestroyContainerEvent**. The variable event is the parent of the specific events for variables, which are set, add and subtract. A categorized event is characterized by having a **PajeValue** associated to it, thus, **PajeStateEvent**, **PajeEventEvent**, **PajeLinkEvent**, and their respective childs inherit from it.

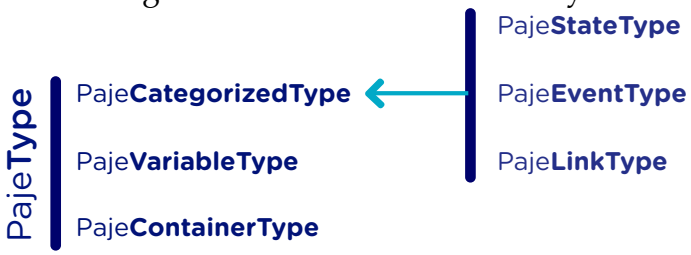
Figure 3.3: Events class hierarchy



Class hierarchy for the Paje types

Figure 3.4 portrays the type hierarchy, where the first element is the **PajeType**. It has a name, an alias and a parent type, which is also a PajeType. These fields are the ones common to all the type definition events described in section 3.1. The immediate childs of this class are: **PajeCategorizedType**, **PajeVariableType** and **PajeContainerType**. As the events, the categorized types are associated to a value, hence, the PajeCategorizedType has a PajeValue field and methods to manipulate it. Its childs are the **PajeStateType**, **PajeEventType** and **PajeLinkType**.

Figure 3.4: Events class hierarchy



Class hierarchy for the Paje entities

As demonstrated in Figure 3.5, the **PajeEntity** is the first node of the entities tree. It originates a **PajeSingleTimedEntity** class, that describes entities with one single timestamp. The **PajeUserEvent** is the only entity with this characteristic, but it is possible to add, in the future, more entities with just one timestamp. The **PajeDoubleTimedEntity** inherits from this class and represents entities with start and end timestamps. Like the other hierarchies, the valued entities are grouped together so a **PajeValuedEntity** is a child of the double timed entity, having **PajeUserState** and **PajeUserLink** as descendents. The double timed entity also has **PajeUserVariable** and **PajeNamedEntity** as childs. A **PajeContainer** inherits from the named entity.

Figure 3.5: Entities class hierarchy

The core simulator

All the simulation is performed in two classes: **PajeSimulator** and **PajeContainer**. A PajeSimulator object is instantiated in the beginning of the program and incorporates all the event processing of the simulation. The type definitions, container creations and entity value declarations are completed and stored in the PajeSimulator object. Every time there is an event of type **PajeCreateContainer**, a PajeContainer object is instantiated. All other events are always associated to a container, thus, they will be simulated in the appropriate container instance. The PajeContainer object will keep the entities until the program finishes. Since all the data from the simulation is kept in memory, the end timestamp is used to signal that an entity no longer can be referred.

The PajeSimulator class lists every type declared and container created throughout the simulation by using map structures (`typeMap` and `contMap`) with the name or alias as key. There is always a pointer to the root type and another to the root container initialized in the beginning of the program. The simulator contains one method for each type of event, which perform all the validations, besides the processing itself. Whenever there is an event that defines a type the entity generated is added to the `typeMap`. `contMap` and the proper method of the container object is called.

The PajeContainer class also uses map structures to store all the entities that are related to it including other containers. Besides one general structure that lists all of the objects related to the container (`entities`), there are auxiliar structures for some specific types, such as states (`stackStates`) and links (`pendingLinks`). There is some redundancy between `entities` and the other constructions but, since the objects are pointers, the changes made in one structure are reflected in the other ones.

Every event that pushes a state will add a state entity to the end of the

`stackStates` stack, while every pop state event will "remove" the last state in the vector by setting its end time. The simulation keeps track of the pending communication links and fails if a container is destroyed, or the simulation ends, before all the links are completed. The PajeContainer class contains a method for each event that is associated to a container, adding and removing entities of these structures listed above.

Current Issues Regarding PajeNG

The focus of the Pajé implementation is to allow the user to extend the Pajé format and adapt the simulator to it. Its support for extra fields allows the inclusion of different descriptions for the events and its modularity facilitates the integration of new classes. Altering or adding simulation behavior can be done by modifying only the `PajeSimulator` and `PajeContainer` classes.

Although complying with its goal of extensibility in terms of expanding the Pajé format, we identified three main issues in the current implementation of PajeNG: little flexibility in the manipulation of data, lack of partial outcomes, and ephemeral results. When the entities are already simulated, a deeper understanding of the code structure is necessary if one wants to define another way of handling the results. Also, the user needs to manage a full set of entities, since there is no flexibility of discarding data that is not relevant. The second issue relies on the fact that the **PajeSimulator** instance maintains all of the simulated objects in memory. If a user wants to see the resulted entities during the simulation, he would need to get into the **PajeSimulator** code to make the necessary changes. Technically, since all the results are stored in memory, it would be simple to add a new functionality, but it is limited to the manipulation of the whole set of results, not each entity separately. Last, the results kept in memory during simulation are discarded at the end, which implies in executing all the simulation again if a trace file needs to be revisited.

Considering the presented issues, an extensible simulator written in Java was developed. The intention of this proposal is to make the simulation core more transparent for the performance analyst providing the created entities in a way that he can manipulate them without looking to the rest of the implementation. The program uses the concept of plugins that attached to every type of

event. The simulator itself addresses the first issue presented, while the creation of new plugins provide a possible solution to the other two. The details of this novel approach, developed in our work, are detailed in the next chapter.

4 AIYRA - A JAVA-BASED SIMULATOR FOR PAJE TRACE FILES

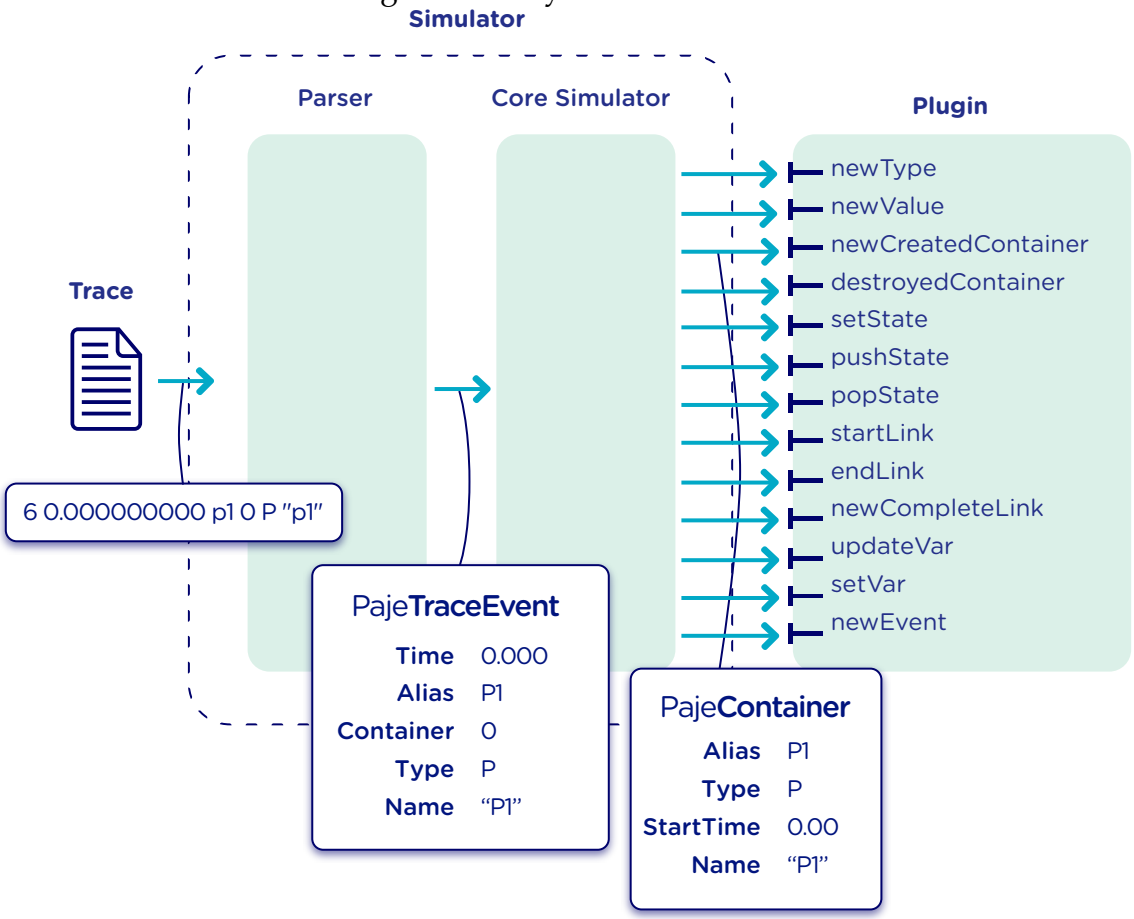
Aiyra is an extensible simulator written in Java that reads trace files in the Pajé format and, instead of storing the results in memory, forwards every created entity to a common place where it can be manipulated freely. The architecture of the implementation, characterized in Figure 4.1, contains three packages: **controller**, **simulator core** and **plugin**. Every event of a trace file always goes through all packages. First, the trace file in the input is read by the parser, where a trace event object is created. This instance contains the type of event in question and the field values. In the example of Figure 4.1, the event read is the creation of a container of type **P** with alias **P1** and parent **0**, which is root. Then, the simulator receives this object and executes the simulation based on the event type. The simulation always generates an entity, even if incomplete. In Figure 4.1, a **PajeContainer** is created without an ending timestamp. Finally, this new entity is sent to the plugin, which contains specific entry points for every different kind of entity.

The program receives arguments from the user in its execution. The **filename** option (`-f`) is the only mandatory one, which indicates what is the trace file to be simulated. There are other two general options: **comment** (`-m`), a comment about the file; and **plugin** (`-p`), which indicates which plugin will be used in the simulation. The details about the already implemented plugins are presented in chapter 5. The following subsections detail each one of the packages.

The controller: option handling and JavaCC

The controller package is the entry point of the program, thus, it also handles the arguments passed by the user. For this processing, an external library (LAUX, 2004) was used. The arguments handling is centralized in one single class, **OptionsHandler**, to facilitate the inclusion of new ones. The Paje file format (see Section 3.1) is parsed by a grammar written using the JavaCC syntax. The file **PajeGrammar.jj** containing all the grammar rules of the format is processed by the Java Compiler Compiler (JavaCC) to generate the parser. Each event definition is stored in an array, while the events are simulated as soon as they are obtained from the trace.

Figure 4.1: Aiyra Architecture



The controller package is composed by all of the JavaCC files described in Section 2.1 and the OptionsHandler class. The generated class **PajeGrammar.java** contains, besides the parsing component, all the necessary Java code for the program to run, such as the initialization of the simulator object, where all of the simulation takes place. Every time an event is identified, the simulator instance, which is the entry point of the simulator core package, is called to simulate that event. The next section describes the simulator core package.

Aiyra's core simulator

Aiyra's core simulator, depicted in Figure 4.2, follows the exact same structure of the **PajeNG** implementation described in section 3.2. Every event read by the parser and sent to the core by the controller goes through the **PajeSimulator** component, which is then forwarded to a **PajeContainer** if necessary. In the example of Figure 4.2, the simulator receives a **PajePopStateEvent**, that is validated

in the **PajeSimulator**, forwarded to the **C1** container, and then dispatched to the proper instrumentation point. The class hierarchy follows the same organization as the PajeNG, thus, it is equally expandable in terms of creating new types of events or entities. However, it does not support extra fields in the events since the focus on the implementation was extending the output of the simulator. This makes our solution more limited for changes in the Pajé Trace file, which happens not very often. Despite that, it would be simple to adjust it since changes do not affect the implementation of the plugins.

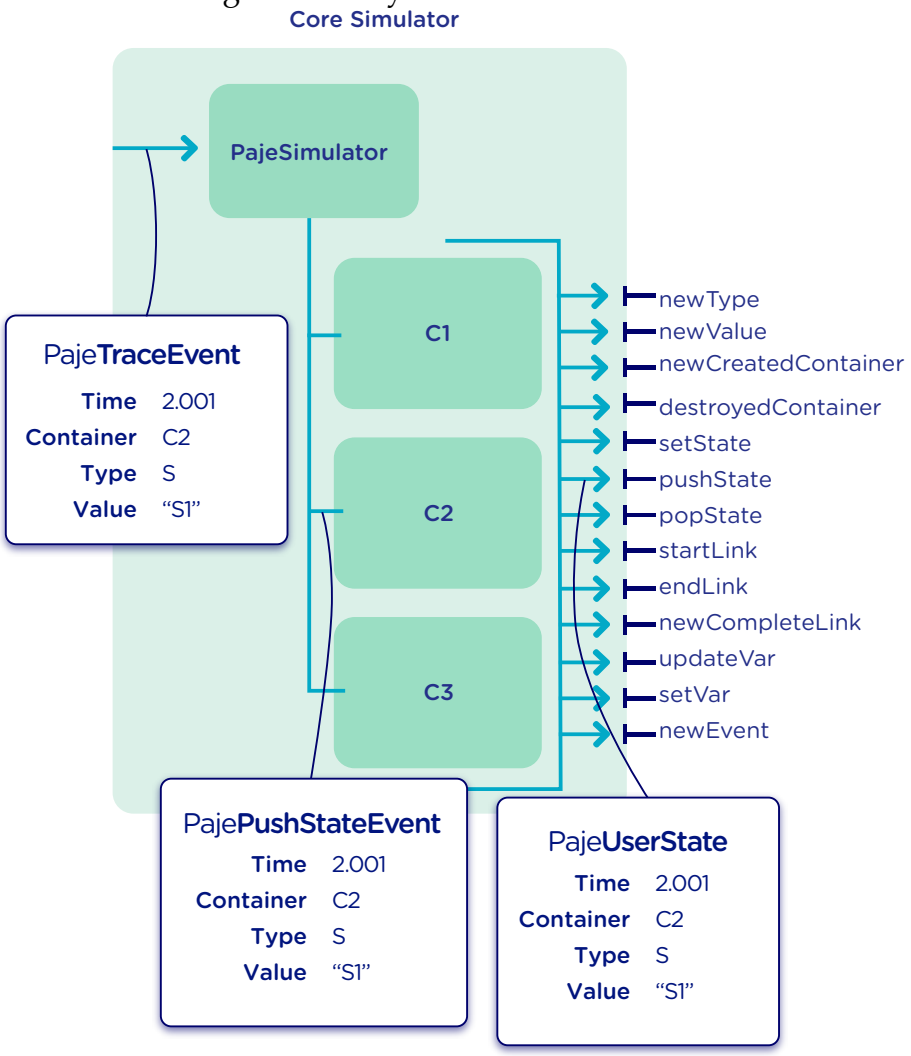
Every entity generated is represented by an object with attributes representing its fields. The class hierarchy of the entities is the same as the one presented in Figure 3.5. All of the types derive from the **PajeType** class, which contains **alias**, **name**, **depth**, and **parent**, a **PajeType** as well, as attributes. It also provides the `getNature()` method, to identify which entity this type describes. The nature is an enumeration and can assume **ContainerType**, **StateType**, **EventType**, **LinkType** or **VariableType**. The **PajeVariableType** adds a **PajeColor** to its attributes, which is an object with the values for red (`r`), green (`g`), blue(`b`) and alpha(`a`). The **PajeLinkType**, in turn, includes **startType** and **endType**, which stand for the type of the start and end containers of the communication. A value is represented by a **PajeValue** class, with **name**, **alias**, **type** and **color** as attributes.

A container object (**PajeContainer**) has an **alias**, **name**, **type** and **parent** (another **PajeContainer**), besides the structures to store the entities related to it, as described in section 3.2. Since the container class is a child of the **PajeDoubleTimedEntity**, it also has a **startTime** and an **endTime**. All other entities are associated to a container and a type, thus, they have a **container** and a **type** fields. The event entity (**PajeUserEvent**) is the only one that derives from **PajeSingleTimedEntity**, hence, it has a unique timestamp named **time**. Also, it has a **value** attribute, which is a **PajeValue**. The other valued entities, **PajeUserState** and **PajeUserLink**, inherit the **PajeValue** attribute from the **PajeValuedEntity** class. A **PajeUserVariable** object also has a **value** attribute but, unlike events, states and links, it is a double number. The **PajeUserLink** has a string that defines the **key** and start and end containers identified by **startContainer** and **endContainer**.

Every trace event simulation has an instrumentation point, which dispatches the entity objects generated to the plugin package. These points are either in the **PajeSimulator** or in the **PajeContainer**, as illustrated in figure 4.2. In the **PajeS-**

imulator are the outputs regarding the definition of types and values and the creation of containers. Although in this point the containers are not complete objects, since they do not have ending timestamp or the related entities, they are forwarded anyway with the alias and type information. In Figure 4.2 we represent the processing of a **PajePushState**, which, after being validated by the simulator, is forwarded to the appropriate container. The **PajeContainer** is in charge of dispatching to the plugin the instances related to it, which involve the states, events, links and variables. It also may send unfinished objects. In our example, the container sends the **PajeUserState** to the **pushState** plugin without an ending timestamp. When there is a **PajeDestroyContainerEvent**, the container object is sent again, now complete with an ending timestamp.

Figure 4.2: Aiyra's Core Architecture



The choice of creating an instrumentation point for each trace event is due to the intention of covering all of the different needs of the user. One may need

the container name before it can process the entities related to it, for example, which cannot be achieved by receiving the container only when it is completed. Or else, may be a situation where the push state events need to be measured, instead of the pop state events, where the entities are finished. Since we cannot predict all of the use cases, it is desirable to have a broad approach. A full list of the plugin entrances and the information received in each one is presented in the next section.

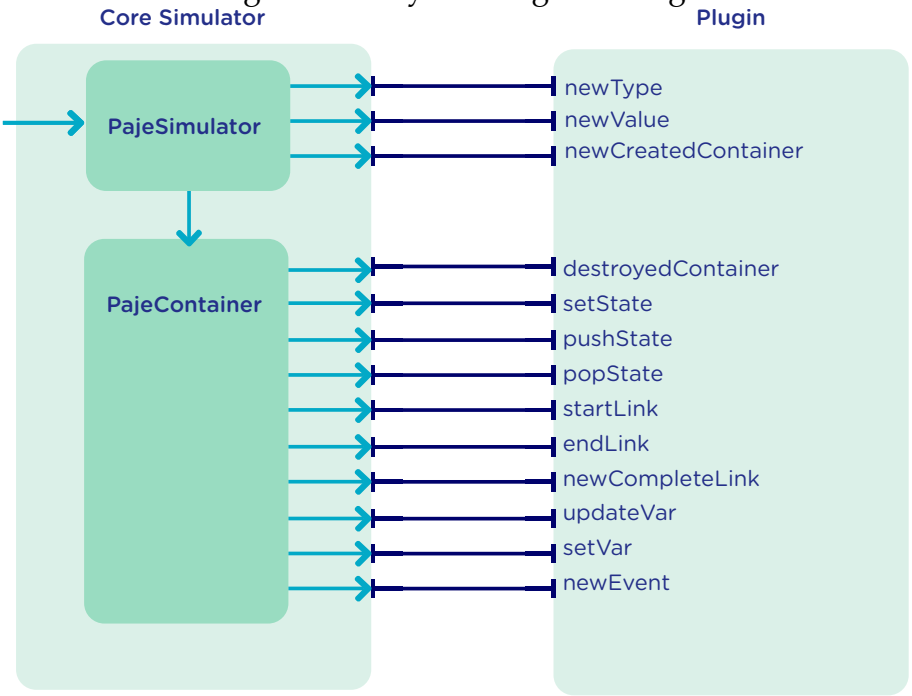
The plugin package

The plugin package, as depicted in Figure 4.3 is a common place where all entities created throughout simulation are sent. It has sets of entry points specific for each type of entity and event. The entrances consist in: **newType**, **newValue**, **startLink**, **endLink**, **newCompleteLink**, **newCreatedContainer**, **destroyedContainer**, **setState**, **pushState**, **popState**, **resetState**, **setVar**, **updateVar**, and **newEvent**. The details of each point are presented later in this section. The plugin package is composed by an abstract class, the **PajePlugin**, with one method for each instrumentation point. It also contains a method called **finish** where the user can perform some concluding actions after the simulation is completed. To create a new plugin, the user just needs to extend the **PajePlugin** class and override its methods. It is also possible to extend another existing plugin, if the differences are small and not worth of a new class implementation.

The entrances of the plugins comprise the definition of types and values, the creation of containers, and the formation and completion of new entities. The **newType** entry point is a unique entrance for when a type of any kind is defined, having the **PajeType** object as argument. The `getNature()` method can be used to identify the exact type. The **newValue** method receives every **PajeValue** created.

When a container is created in the simulation, the instance is forwarded to the **newCreatedContainer** entry point, with the end timestamp set to -1. Whenever a method receives an entity that is not completed yet, the end timestamp will be -1. The **destroyedContainer** method takes in a complete container that has just been destroyed. Most of the entities are removed during simulation, but the destroyed container may have some remaining ones that could not be excluded,

Figure 4.3: Aiyra's Plugin Package



such as variables.

The link entry points receive **PajeUserLink** objects. In the **startLink**, the end time and end container of the communication link are unknown, while in the **endLink**, instance has the end point of the link but not the start. The **newCompleteLink** method takes in a link entity with beginning and end. Anytime a variable is set or updated, there are three **PajeUserVariable** entities sent to the plugin: the **first**, which contains the first value of the variable; the **last**, which is the one immediately before the variable in question; and the new variable which is not completed yet (**newVar**). The aditions and subtractions are sent to the same point (**updateVar**). Since the previous variable objects are necessary to generate the new value, they are not removed from memory during simulation. The set, push and pop state instrumentation points all receive a **PajeUserState**. The only one with an entity with beginning and end timestamps is the **popState**. The **PajeUserEvent** objects are sent to the **newEvent** function.

To validate the concept of the plugins and its entry points, three plugins were created: **PajeNullPlugin**, **PajeDumpPlugin** and **PajeInsertDBPlugin**. Their implementation is described in the next chapter.

5 AIYRA'S STANDARD PLUGINS

To address traditional uses of Paje trace files, we have implemented three plugins for the Aiyra framework: the **PajeDumpPlugin**, the **PajeInsertDBPlugin**, and the **PajeNullPlugin**. The first one is used to match the behavior of the existing `pj_dump` tool but without the issues we have mentioned in Section 3.3; the second one can be used to insert the trace file in a relational database, allowing the user to use SQL commands to inspect simulated traces; and finally, the third can be used to evaluate the Aiyra's performance for any kind of input. We detail each of them in the following, from the one that presents the lowest to the highest complexity.

Paje Null Plugin

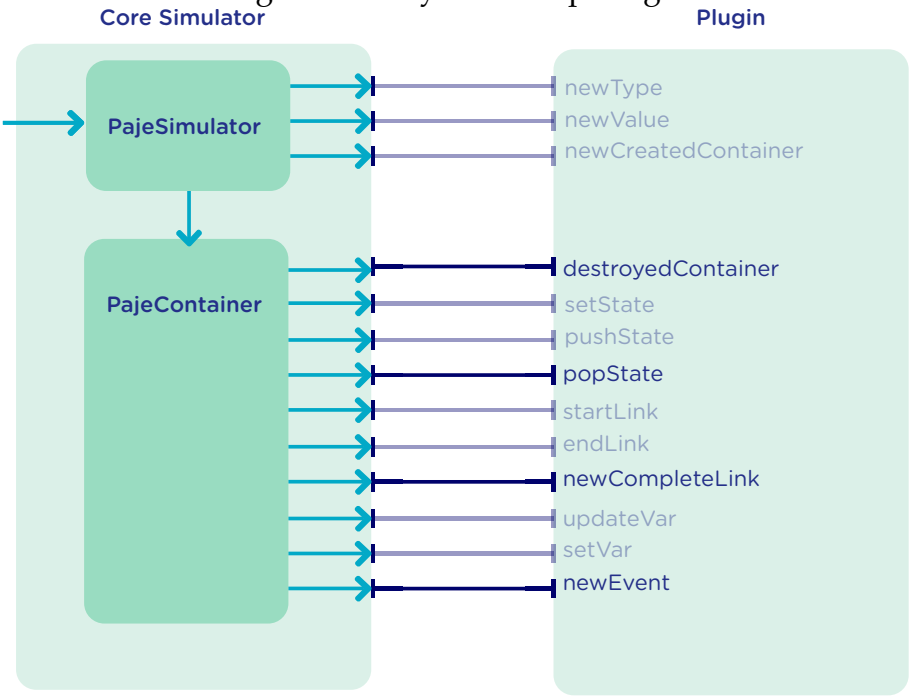
The **PajeNullPlugin** is the default plugin option. It does not make any treatment to the data so the objects are simply discarded. The utility of this plugin relies on the need to verify the performance of the simulation itself, without the interference of data manipulation. Since the main goal of this proposal is to detach the core simulation from the data handling, it is desirable to be able to execute the core alone.

Paje Dump Plugin

The Dump plugin performs the same action as the **pj_dump** tool, which dumps to the standard output the entities generated by the simulator. The implementation consists in inserting a `print` function in each instrumentation point that receives a complete entity. These points are: **destroyContainer**, **popState**, **newCompletedLink**, and **newEvent**. Figure 5.1 demonstrates the usage of the available entry points. When it is a destroyed container, it is necessary to iterate over the entities left in the container. The variables printed in the destruction of the container, since they are not removed during simulation.

The difference between the **PajeDumpPlugin** and the **pj_dump** tool is that the first one outputs the information as soon as the entity is completed. The

Figure 5.1: Aiyra's Dump Plugin



pj_dump, in turn, keeps everything in memory before dumping it all at once. With this approach, it is possible to solve the issue regarding the need to wait for the program to end to have the results.

This plugin can be called with the argument `PajeDump` in the `-p` option and adds a new argument (`-l`) that can group together a certain number of entities before dumping it. The option receives an integer as parameter defining the number of lines it should reach before dumping the entities. This provides a little more flexibility for the user and may improve the performance, since the printing function of Java costs time. For it to be possible, a `StringBuilder` is used as a buffer keeping all of the output until it reaches the number of lines desired.

Paje Insert Database Plugin

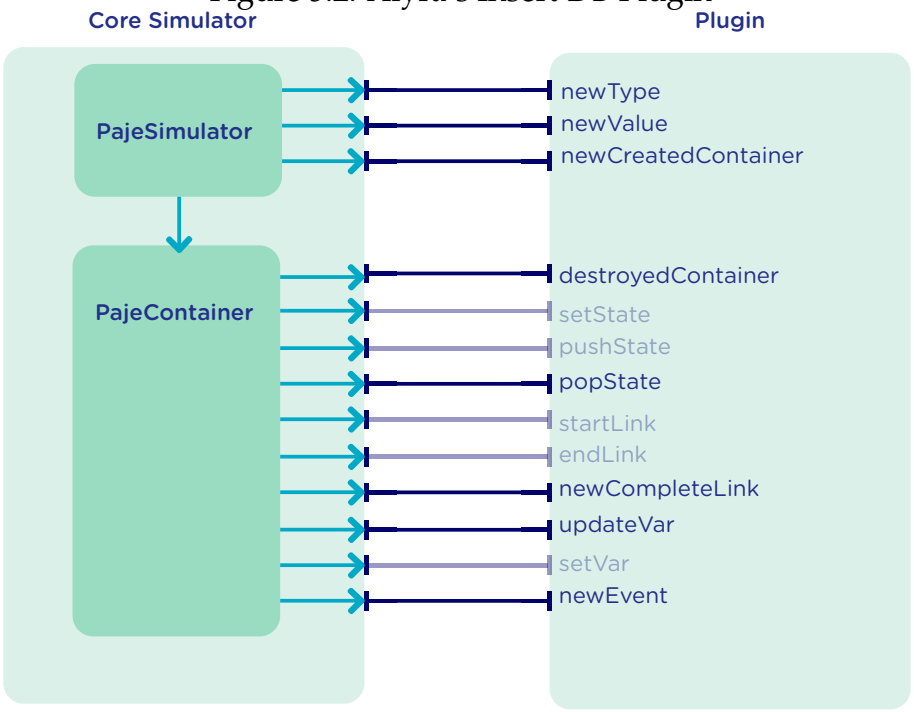
The **PajeInsertDBPlugin** saves in a relational database all the results of the simulation. For the implementation, the JDBC API was used to make a connection with the MySQL database. The schema used was specially designed for the Pajé format and is presented in the next subsections. This plugin allows the user to save data from multiple files in the same database.

The plugin can be used by specifying `mysql` as argument for the `-p` op-

tion. It is necessary to have a MySQL connection and a database with the correct schema. To specify the server of the connection, there is the option `-s`. It is also possible to inform a username (`-u`) and a password (`-pwd`). The default for these options is: **localhost**, **root** and **root**, respectively.

The following entry points were used in the **PajeInsertDBPlugin**: **newType**, **newValue**, **newCreatedContainer**, **destroyedContainer**, **popState**, **newCompleteLink**, **updateVar** and **newEvent**. Figure 5.2 demonstrates these points. Types, values and containers are inserted in the database as soon as they are created due to the dependency of other entities on these ones. When a container is destroyed, its **endTime** is updated in the database.

Figure 5.2: Aiyra's Insert DB Plugin



The first approach of this implementation consisted in inserting the entities in the database at the time they were created. Database accesses cost time and, by executing some preliminary tests, we observed a very bad performance, that took an unacceptable amount of time (over 24 hours for 1 Gigabyte trace files). To solve this problem, we used the mechanism of **batches** provided by JDBC, which sends a block of queries all at once, reducing the communication overhead. This functionality is optional and can be included by adding the (`-batch`) option with an integer as argument. This number will define how many queries it will store before inserting a batch in the database. This is only applicable to states, events, links and variables, since types, values and containers are immediately inserted.

performance analysis for different sizes of batch is presented in the next chapter.

To create a relational database for the Pajé format, first, we created an entity-relationship model that is described in the subsection below.

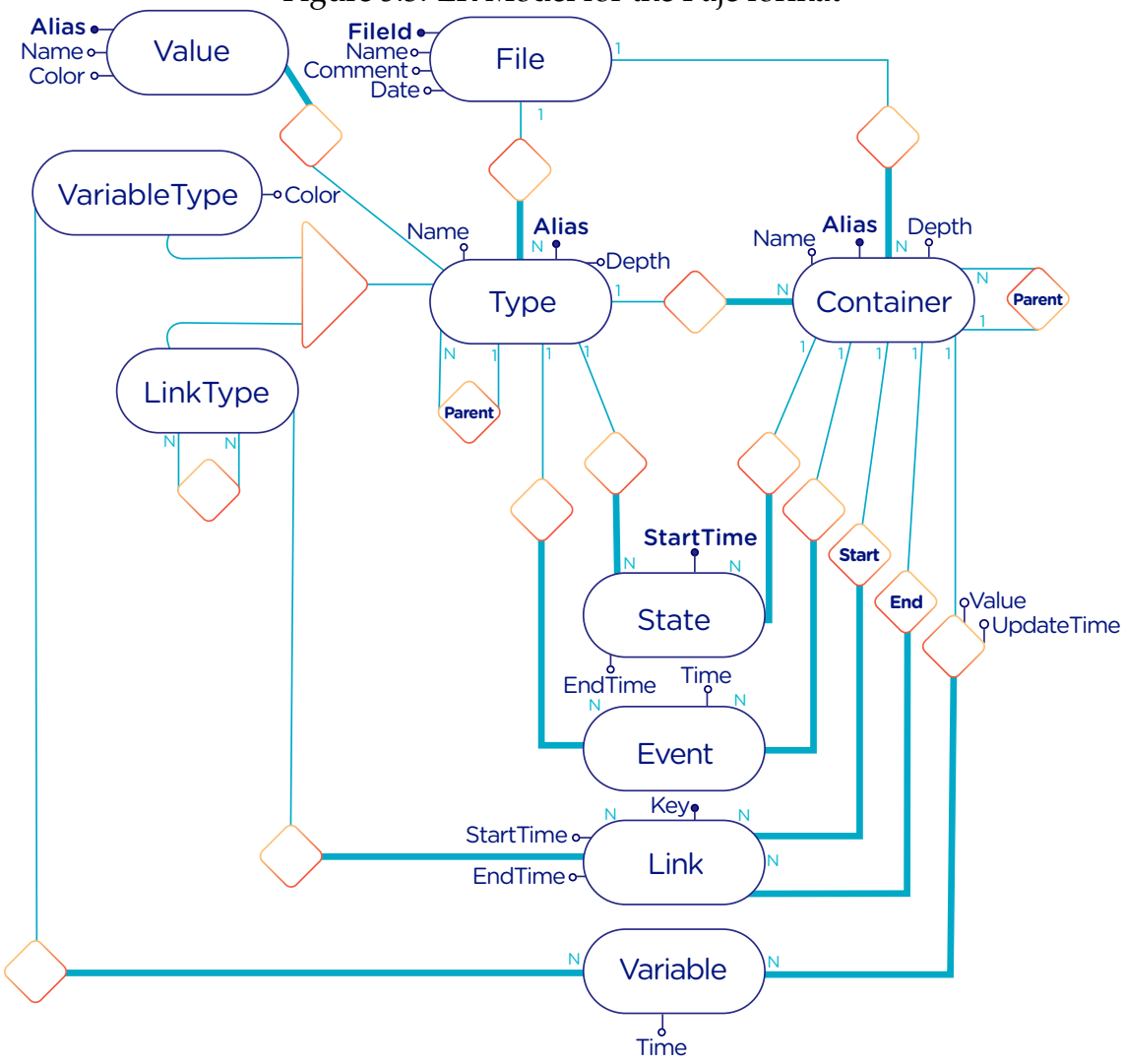
Entity-Relationship Model

The entity-relationship (ER) model, illustrated in Figure 5.3, contains one entity for each type of Pajé object. Also, to support multiple files, there is a **file** entity, which has the **name**, a **comment** and the **date** as attributes, as well as a **file_id**. The **Type** and **Container** entities have an identifying relationship with **file**, which means that the file id is part of their identifier. The relationship is one-to-many, since a file can have multiple types and containers.

The **Type** entity has **alias**, composing the identifier, **name** and **depth** as attributes. It also contains a self-referencing one-to-many relationship to indicate the **parent** type, as a type can have multiple children. It is associated to a **Value** entity, which describes the **PajeValue** class, with **alias**, **name**, **type** (identifying relation) and **color**. Link and variable types have exclusive attributes that are not common to all types, thus, both are specializations of **Type**. **LinkType** adds a relationship with itself to represent a the start and end. This is a many-to-many relationship because the types can be the start and end to various communications. The **VariableType** has a color attribute.

The **Container** entity has an identifying one-to-many relationship with **Type**, as well as every other entity, since all of them are classified by a type. Containers have the same attributes as types, including the **parent** one-to-many relation. All of the entities that are related to a container, have an identifying one-to-many relation with **Container** entity. **State** has **startTime** and **endTime** attributes, where the first is identifier. **Link** has two one-to-many relationships with **Container**, one for **startContainer** and one for **endContainer**. A **Variable** entity contains the **Time** attribute, as well as an **updateTime** in the relation with **Container**. Also, this relation has a **value** attribute. The **Event** entity has a **time** field.

Figure 5.3: ER Model for the Pajé format



Relational Model

A translation to a logical model was made after the creation of the conceptual model. In this conversion, besides applying the universally known rules presented in Chapter 2, we considered the usability of the schema, analysing the common requests made in the Pajé data. This reflection is a usual part of the process, where the needs of the client are contemplated.

The entities defined in the ER Model all became tables. For the **Type** specialization, we used the first option presented in Section 2.3, combining everything in a single table with the following fields: **file_id**, **alias**, **name**, **depth**, **parent_type_alias**, **start_link_type**, **end_link_type** and **color**. **File_id**, inherited from the identifying relation with **File**, and **alias** compose the primary key. The

self-referencing relationships are described as foreign keys in their tables. The entities associated to the container all have at least three foreign keys that are also identifiers: **type_alias**, **container_alias** and **file_id**. Since the **Link** entity has a unique key, its two foreign keys from **Container** don't belong to the identifier.

In our ER Model, the value is only associated to the **Type**, thus, if one wants to know the value of a state, for example, it needs to first get its type, then, go to **Type** table to retrieve the value. Since it is desirable to easily get an entity's value, we added a relationship between the valued entities (**State**, **Link** and **Variable**) with **Value**. **value_alias** is an identifying foreign key for all, except **Link**, where the identifier consists only in the **key**, **type** and **file_id**. With the conceptual model of the **Variable** entity, it is required to retrieve two rows if one needs to know the beginning and ending timestamps of one entity. Since this information is very important, we changed the **Variable** table for the tuples to explicitly have **startTime** and **endTime**.

6 PERFORMANCE EVALUATION

An evaluation of Aiyra's performance was made to have concrete conclusions about the outcome of this proposal. Two main tests were executed: a comparison between Aiyra and PajeNG and an analysis of the impact of different batch sizes in the **PajeInsertDBPlugin**. Since Aiyra is strongly based in the PajeNG implementation, it is valid to examine if the modifications and language transition have brought significant performance impact on the simulation. The plugin that inserts the data in a MySQL database is the only one that brings an extremely different functionality to the program, hence, it was chosen to be studied. As it involves the connection with an external tool, the analysis of its performance and the study of the most efficient use of it is very important.

The experiment execution order is dictated by full factorial experimental designs created in the R language with the **DoE.base** package. The package generates a Comma-Separated Values (**CSV**) file with one column for each factor. The rows represent each possible combination of the different levels and multiplies it by the number of replications. We created **bash** scripts to execute the experiments of the design generating another **CSV** sheet including the response variables defined for the experiments. The details about the factors and levels for each test are described in the next section. The remainder of this chapter comprises the analysis results.

Methodology

The experiments are performed in three different machines: **luiza**, with a Mac OSX environment, **guarani**, and **orion1**, both running Linux. The details about the experimental platforms are described in Table 6.1. We have created three input trace files with different sizes identified by **small**, **medium**, and **big**. The sizes for each of these cases are 128 Kilobytes, 128 Megabytes, and 1 Gigabytes, respectively.

Java programs run in the Java Virtual Machine (JVM), an abstract computing machine where the specifications about memory size are placed. For the experiments, the heap size is the relevant information. We used the default configuration, which is presented in Table 6.2.

Table 6.1: Experimental Units description

	Luiza	**Orion1**	**Guarani**
Processor	Intel Core i7	Xeon E5-2630	Intel Core i5-2400
CPU(s)	1	2	1
Cores per CPU	4	6	4
Max. Freq.	2.7 GHz	2.30GHz	3.10GHz
L1d/L1i Cache	32/32KBytes	32/32KBytes	32/32KBytes
L2 Cache	256KBytes	256KBytes	256KBytes
L3 Cache	6MBytes	15MBytes	6MBytes
Memory	16GBytes	32GBytes	20GBytes
OS	OSX 10.10.5	Ubuntu 12.04.5	Debian 4.3.5-1

Table 6.2: JVM heap sizes in bytes

	Luiza	**Orion1**	**Guarani**
Inital heap size	268435456	526368896	327213824
Maximum heap size	4294967296	8422162432	5236588544

Aiyra and PajeNG Comparison Methodology

This experiment evaluates the performance of Aiyra using the **PajeNullPlugin** against two versions of the **pj_dump** tool (**pj** and **pjflex)**, both part of PajeNG. The difference between the `pj_dump` versions is in the reading of the trace file: while the first (**pj**) uses a hand-tailored parsing, the second (**pjflex**) uses a standard scanner and parser generator (based on the **flex** and **bison** from GNU). Since we only need the execution time to carry out the comparison, the **pj_dump** executions received `--quiet` as a parameter to avoid the actual dumping of the information in the standard output. It is important to highlight that Aiyra does not perform any action in the resulted entities and discards all of them.

Concerning the experimental design, two factors are chosen: **input** and **version**. The first assumes the values **small**, **medium**, and **big**. The second, **aiyra**, **pj** and **pjflex**. Each experimental combination is executed 30 times so we can understand measurement variability. This value was selected after a few preliminary experiments where we have noticed very little experimental variability. As the executions take a significant amount time, 30 repetitions is enough to have reliable results. Since we have two factors, each with three levels, we have a total of 270 experiments (the product of $3 * 3 * 30$). The outcome of this experiment is the **execution** time for each combination of input and version. We execute the design in each one of the platforms, thus, we had three files each with 270 rows.

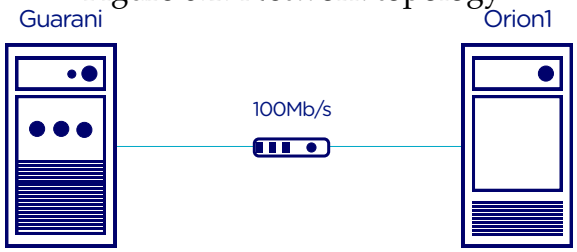
The total number of experiments considering the three platforms is therefore 810.

PajeInsertDBPlugin evaluation

The **PajeInsertDBPlugin** provides an option for the user to define a batch size for the insertion in the database. The size defines the number of entities to be inserted at once. This means that the queries are stored in a buffer until a counter reaches the specified value. Although this approach reduces the execution time, compared to the first attempt described in Chapter 5, it occupies a significant amount of memory. We want to define what is the best choice of batch size for different scenarios.

For this experiment we have used the same experimental platforms as the first one, but we added a fourth experiment which consists in the remote access between **guarani** and **orion1**. In the experiment, we had the simulator running in **guarani** inserting data in the database hosted by **orion1**. Figure 6.1 represents the network topology connecting the two machines. Both are placed at the Informatics Institute of the Federal University of Rio Grande do Sul. Albeit the platforms are equipped with gigabit network cards, the theoretical bandwidth is only 100Mb/s. We believe this is due the existence of a 100Mb/s switch between both endpoints limiting the network speed. The average round-trip time (RTT) is 0.5 seconds and was measured by the **ping** bash command.

Figure 6.1: Network topology



The factors for this evaluation consist in the **input**, the same as the ones described previously, and **batch**, which assumes six different values. The **batch** factor consists in the size of the batch to be inserted in the database. The batch size numbers are not fixed and varies depending on the different input sizes. The levels are classified from **A** to **F**, where **A** represents the highest number possible for a batch, meaning one single insertion with all the contents of the trace. The other five levels consist in dividing the previous one in half. The **A**

value for each input size was previously calculated and the rest was generated by the dividing the first one. In this design, we have a six-level factor, a three-level factor and 30 replications, which results in 540 experiments to run in each of the four experimental units (total of 2160 experiments).

To complement this analysis, we also generate traces for the batch executions. The trace contains the **start time** and **end time** of every database insertion. This data is useful to obtain richer information about the impacts of the batch mechanism in the performance. The next section demonstrates the results and our interpretation of the experiments.

Results and Graphics

We have used the R language to merge the data from the different files generated and to plot results. For the experiments we have used the average execution time among the replications as a measurement. For **PajeInsertDBPlugin**, there is also the average insertion time. The time unit is seconds. We considered the standard error to be three standard deviations divided by the square root of the number of experiments, which cover 99.7% of the cases assuming a normal distribution (NARASIMHAN, 1996). We describe below our expectation about each experimental setup and the interpretation of the measurements.

Comparison between Aiyra and PajeNG

Expectation

It is universally known that C++ is a language with better performance than Java. We suppose that Aiyra will be slower than PajeNG, but with an acceptable execution time. It is also expected that the version **pj_flex** will be slower than **pj**, an information provided by the PajeNG developers.

Observed Results

We can see in Figure 6.2 that, for the **medium** and **big** inputs, Aiyra was actually faster than both versions of PajeNG. Table 6.3 portrays the average execution time in seconds for these input sizes separated by platform. For all platforms, **aiyra** is at least two times faster than either **pj** and **pjflex**. To understand

this results it is crucial to recognize the difference between the implementation and configuration of both programs. Aiyra is designed to get rid of the entities as soon as they are finished. Thus, with the **PajeNullPlugin**, very little is kept in memory. One of Java's biggest overhead is the memory handling, specially the Garbage Collector(GC). Since we have an implementation that stores as few objects as possible, and the less alive objects in the program, the faster is GC (KOPP, 2011), Java may have a chance in this case. Another important difference among Aiyra and PajeNG is the process of reading the trace file, parsing it and then sending to the simulator. As seen in Chapter 3, the PajeNG file reader first reads from the file a chunk of data, then the decoder, which has predominant execution time, breaks it into events and sends them to the simulator. The next segment of data is only read after the first is completely decoded. On the other hand, Aiyra's controller sends each event read immediately to the simulator. The controller package performs the reading and parsing simultaneously and is probably faster than the decoding process of PajeNG. Since the memory allocation is not usually a problem in C++ programs, it is more likely that this result relies on the implementation difference. As expected, the **pj_flex** version was slower than **pj**, thus, slower than **aiyra**.

Figure 6.2: Results of comparison between Aiyra and PajeNG

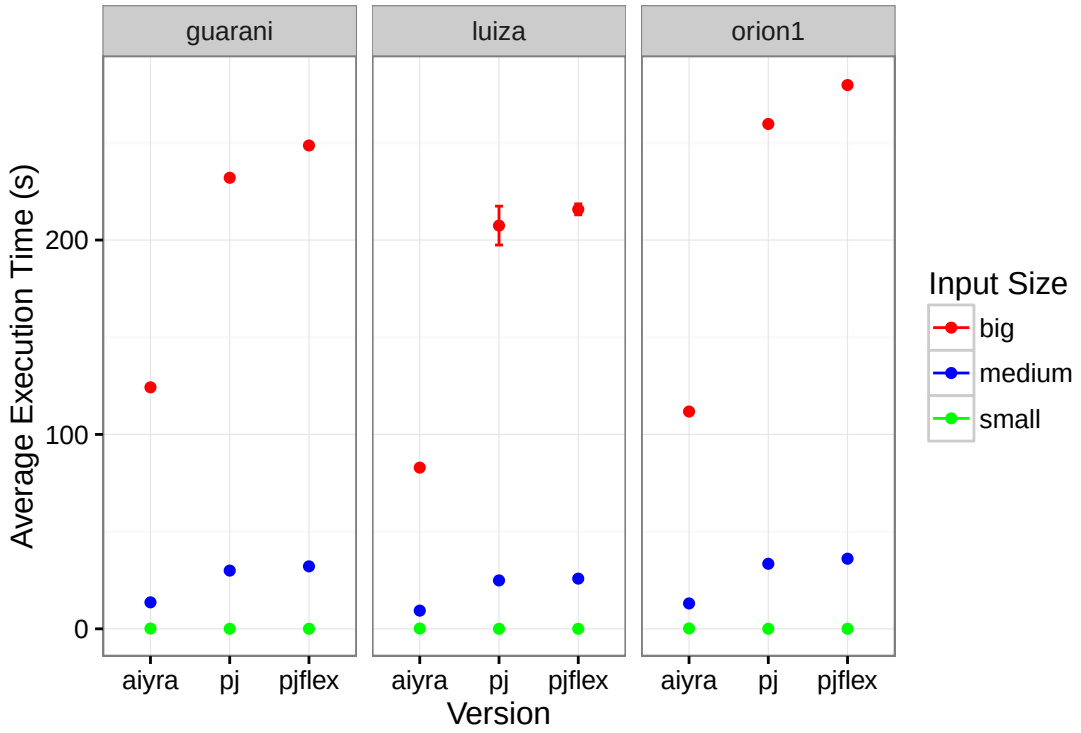


Table 6.3: Average Execution Time (s) for each configuration

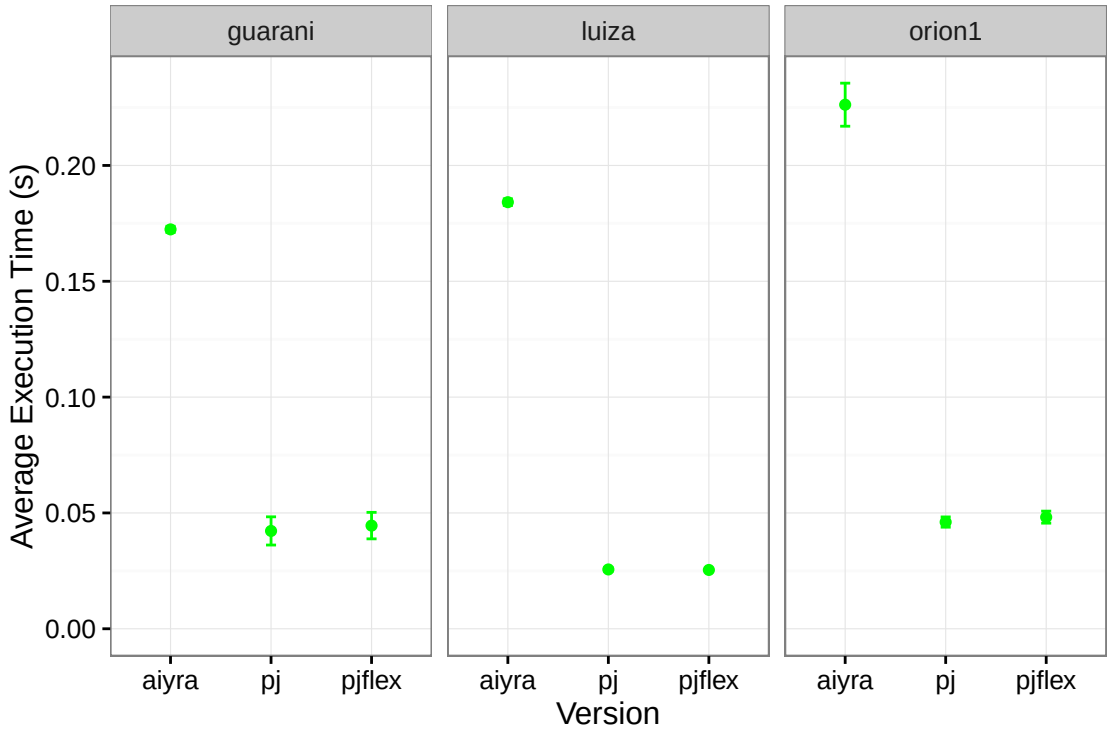
Platform	Tool	Big	Medium
Luiza	Aiyra	82.94644007	9.38572857
	Pj	207.434215571	24.92862427
	PjFlex	215.79006593	25.85007707
Guarani	Aiyra	124.24501307	13.62979877
	Pj	232.04630737	29.97373843
	PjFlex	248.65192840	32.14986230
Orion1	Aiyra	111.78373253	13.08560363
	Pj	259.72360747	33.47163073
	PjFlex	279.74462510	36.08434247

Notice that the experiment in **luiza** had a bigger variability with the **pj** version and the **big** input. The average execution time for the big file in the **aiyra** version was around 80 seconds, but there was an outlier experiment that executed in 2888 seconds. This particular measurement has been manually removed from the data set. Even with this particular case deleted, an instability can still be perceived. The experiments are randomized in the design so, if there was any disturbance in the environment during the experiment, it should have affected other configurations as well, but we can notice the variability only in this particular configuration. There is no confirmed explanation for this situation, it may be because **luiza** is running a different operating system and did not handle well the C++ parsing with the biggest file. Considering that it does not affect the overall analysis, the experiment was considered valid.

So far, we have analysed the execution time of the **medium** and **big** inputs. Figure 6.3 shows only the measurement considering the **small** input in the different platforms, to have a closer look in its time scale. The behavior for the **small** input is different from the others. In this case, we observe exactly the expected result (see previous subsection). For the bigger files, we concluded that the process of decoding the data in blocks was what made **pj** and **pj_flex** slower. In this case, we have a very small file, so we will have less blocks and the decoder is called fewer times.

These results are particularly important to assure the relevance of this proposal. By detaching the core simulation from the data handling, we perceive that there is room for performance improvements in the replay of large Pajé trace files. It is possible to observe that the fewer data we retain in memory, the better our

Figure 6.3: Results of comparison between Aiyra and PajeNG for the small input



performance will be. The implementation of Aiyra gives the user more flexibility to manage the memory usage of his program and space to develop high performance implementations. It is also possible to notice that changes in the design of the program, specially in the trace file parsing and detected events handling, may have significant impacts in the performance of the program.

PajeInsertDBPlugin Evaluation

The objective of this experiment is to measure what is the best balance between number of database accesses and memory usage. We have already discarded inserting each entity at a time, so it is necessary to keep entities temporarily in memory. We tested the simulation time and memory usage for different combinations of input and batch size. The response variables were: **execution time**, which is the total duration of the execution; **insertion time**, the time it took to insert the data in the database; and **maximum memory**, the maximum usage of the memory during the simulation. The memory value used was the maximum observed in the replications. As this experiment consists in storing data of 540 executions in memory, we used a `-test` flag, which drops the database and

recreates it after each simulation.

Since we had a very large number of experiments, it took more than 48 hours to execute the design in each platform. **Guarani** and **orion1** are machines shared by the professors and students in the Informatics Institute of the Federal University of Rio Grande do Sul, thus, it was not desirable to disable other users from using the machine for longer than two days. For these reasons, the experiment in **orion1** and **guarani** had only 10 replications. Even so, we have observed very little variability among the executions in the exploratory tests. The experiments for these platforms do not have the **A** batch size.

Expected

According to the observation of bad performance when executing excessive accesses to the database, we expect that the bigger the batch, the better the performance, since it will make fewer requests to the MySQL server. Naturally, this would also cause a higher memory usage, so the maximum memory utilization will be larger for bigger batches. We have three experiments making an access to a local database, and one performing remote request in another machine in the same network. It is logical to expect that the remote case takes longer than the local experiment.

Observed Results

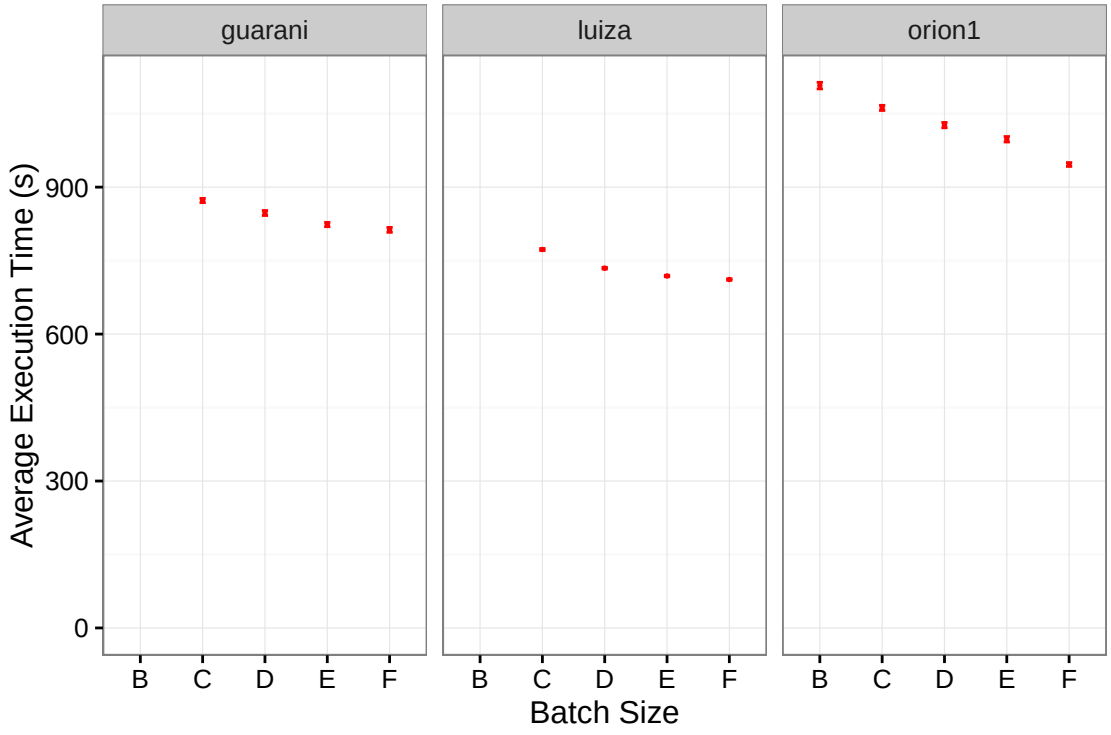
As the values for A, B, C, D, E and F vary among the different input sizes, it is consistent to analyze each input separately. The results are divided in subsections according to the metrics analysed: **execution time**, **insertion time**, and **maximum memory usage**. There is also a subsection to analyse metrics obtained during the execution of the batches.

Execution Time

The **execution time** measures the duration of the entire simulation from beginning to end. Figure 6.4 illustrates the behavior of the execution using the **big** input. We have results for all batch sizes except A. The reason for this is that Aiyra exceeded the heap space to store data in memory, mainly because we have adopted default values for heap allocation in the JVM, as presented in Table 6.2. The **guarani** platform was incapable of handling A and B sizes. Our analysis is unaffected by these missing cases since it is possible to observe a pattern, which is the opposite of what we have expected. We can see that the smaller the batch size,

the better the performance. In fact, this is easily explained since Java's memory management may be very slow, specially if the heap size is big. By these results, we see that the memory usage overhead is higher than the database access cost. With the **medium** input, presented in Figure 6.5, we observe the same behavior as the big one for **luiza**, **guarani** and **orion1** platforms. The **orion1** results are inconclusive for the **medium** input size, probably due to external interferences. Although we have locked the machine for the experiment, it was out of our hands to stop eventual programs that could be running in other students' users and affecting the performance.

Figure 6.4: Results of batch sizes variability for big input



Figures 6.6 and 6.7 portray the comparison between the local and remote execution times in **guarani**. As expected, the remote access increases the execution time. It is not a big difference because both machines are in the same network with only one hop of distance.

For smaller inputs, however, we see (in Figure 6.8) that the behavior is what we have predicted. It is explained by the very little memory usage, that is too irrelevant to impact the performance. It is worth mentioning that 128 Kilobytes, which is the size of the **small** input, is very uncommon and that the usual traces are at least in the Megabytes order of magnitude.

Figure 6.5: Results of batch sizes variability for medium input

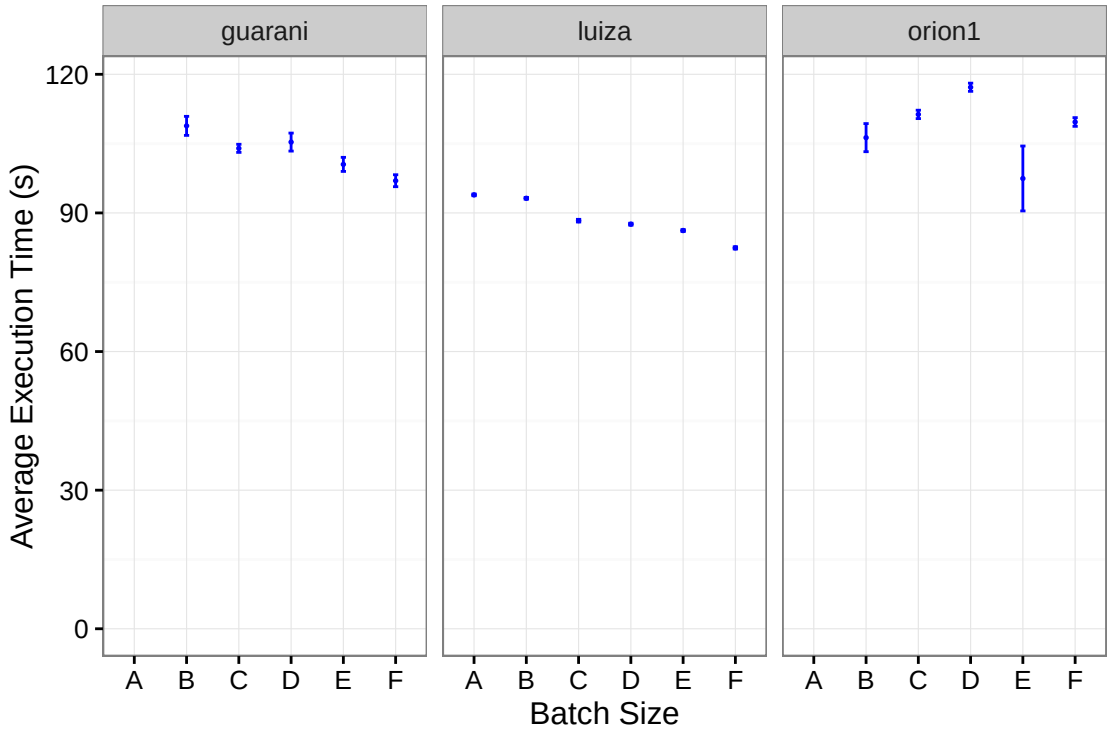
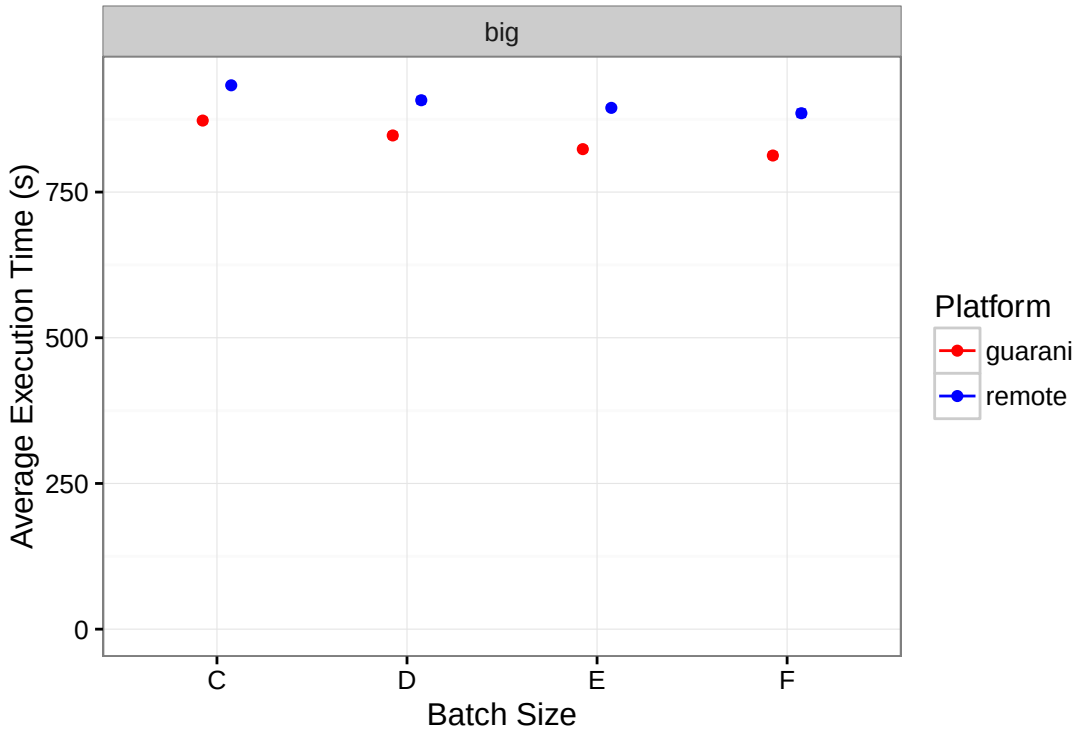


Figure 6.6: Results for remote and local executions for the big input



Memory Usage

Going further on the analysis, we see in Figures 6.9 and 6.10 the memory usage

Figure 6.7: Results for remote and local executions for the medium input

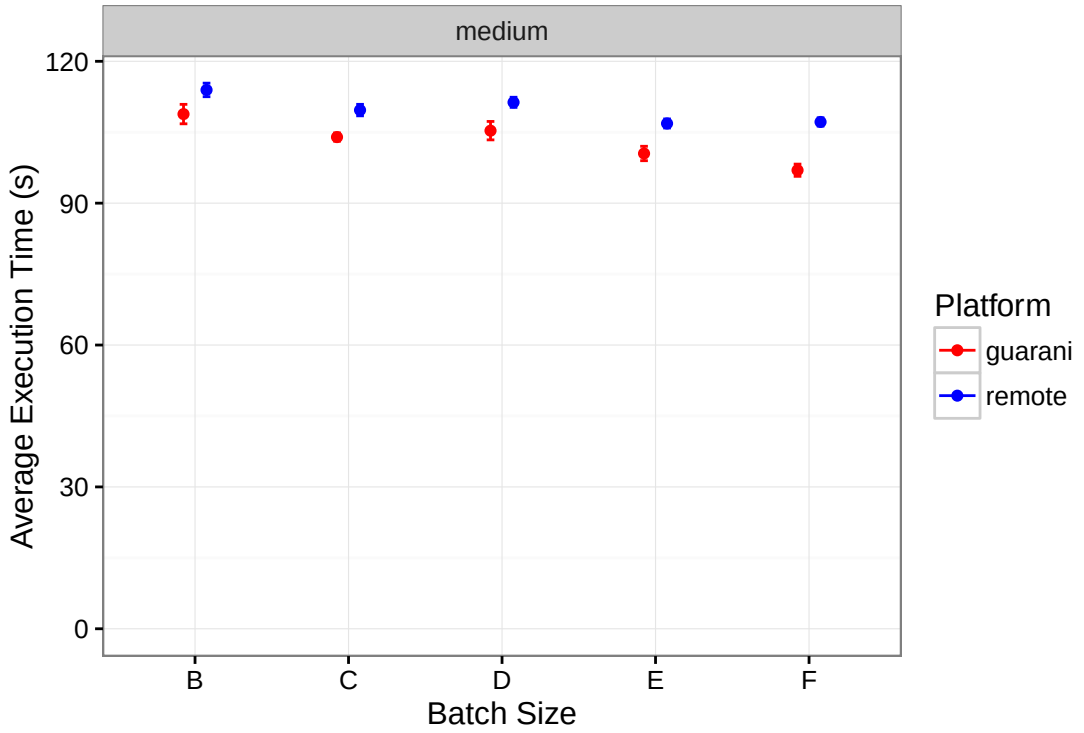
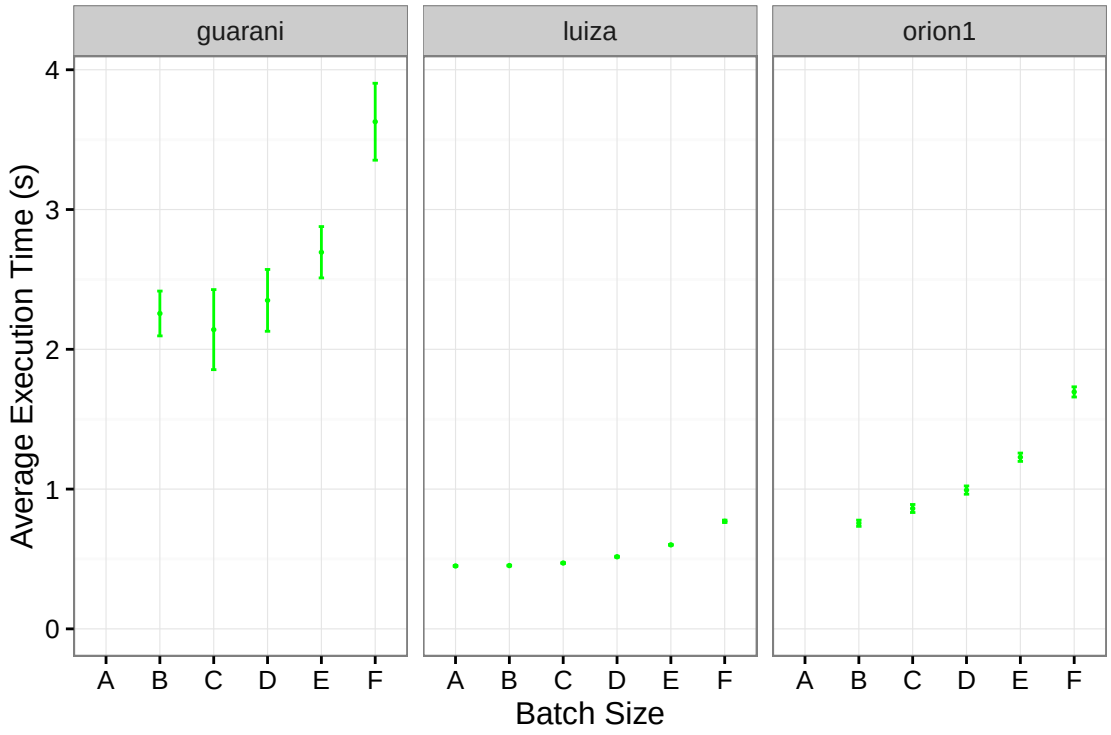


Figure 6.8: Results of batch sizes variability for the small input



for the scenarios with **big** and **medium** inputs. Naturally, the memory usage peak is higher for bigger batches.

Figure 6.9: Results of batch sizes variability for the big input

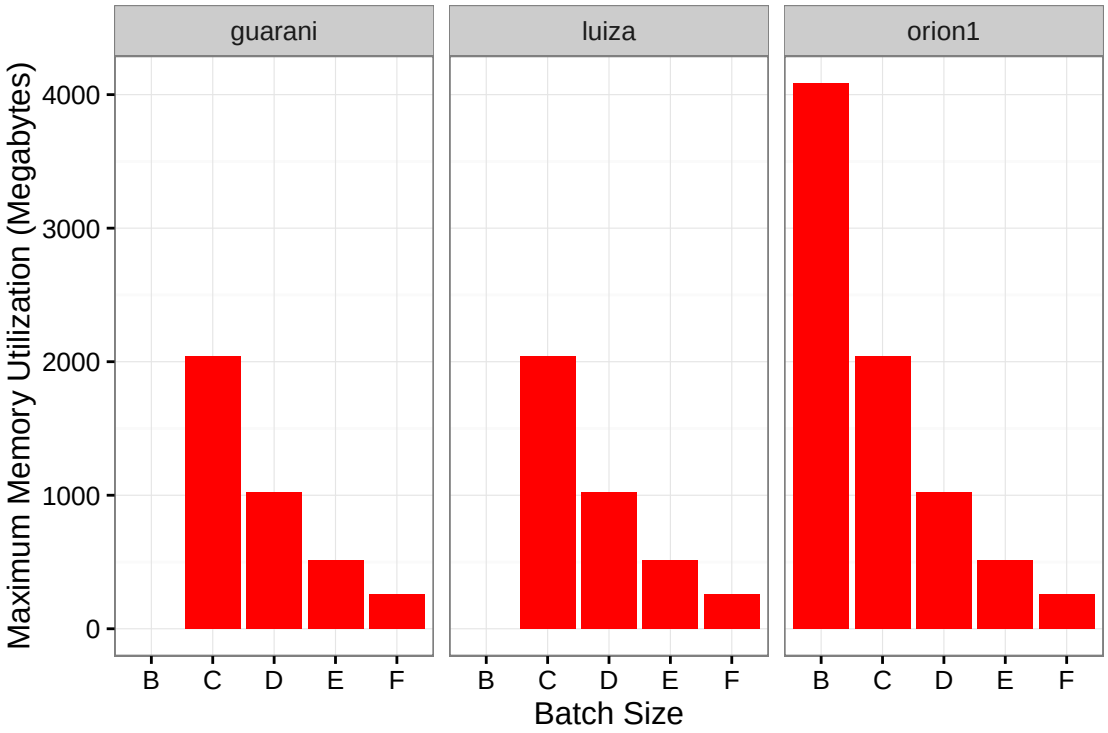
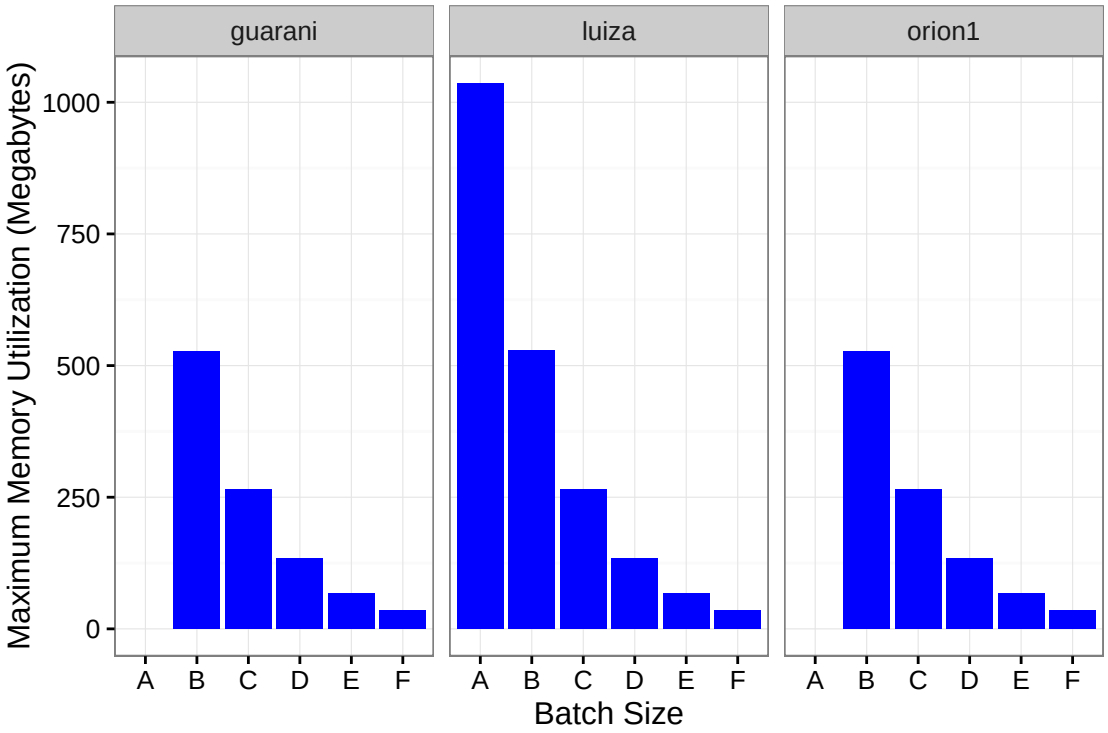


Figure 6.10: Results of batch sizes variability for the medium input



The memory usage with the **small** file, portrayed in Figure 6.11, had the same behavior as the previous ones, except for the **guarani** platform. Since it is

not a logical that the batch size **E**, which is the second smallest of the levels, occupies more memory than **A**, **B**, **C**, **D**, and we can see in Figure 6.8 that there was a big variability in this platform, we consider that it was an external interference. Even though, the memory usage of this **input** is less than 5 Megabytes, while for the others it reached over 1Gigabyte for the **medium** input and 4Gigabytes with the **big**.

Figure 6.11: Results of batch sizes variability for the small input

Insertion Time

The insertion time measures only the moments of the execution that the program is making an insertion in the database. By analyzing the results, we observed that the number of accesses to the database did not impact on the performance of **big** and **medium** traces. Figures 6.12 and 6.13 support our conclusion that the number of accesses did not affect the insertion time and what increases the execution time is the memory usage. We see in both graphics that the insertion time is almost the same for the different batch sizes. The medium input in **guarani** had actually a lower insertion time for smaller batches.

For the **small** input, the average insertion time had a small difference between batches, as depicted in Figure 6.14. It presented the expected behavior described previously.

Figure 6.12: Insertion time for the big input

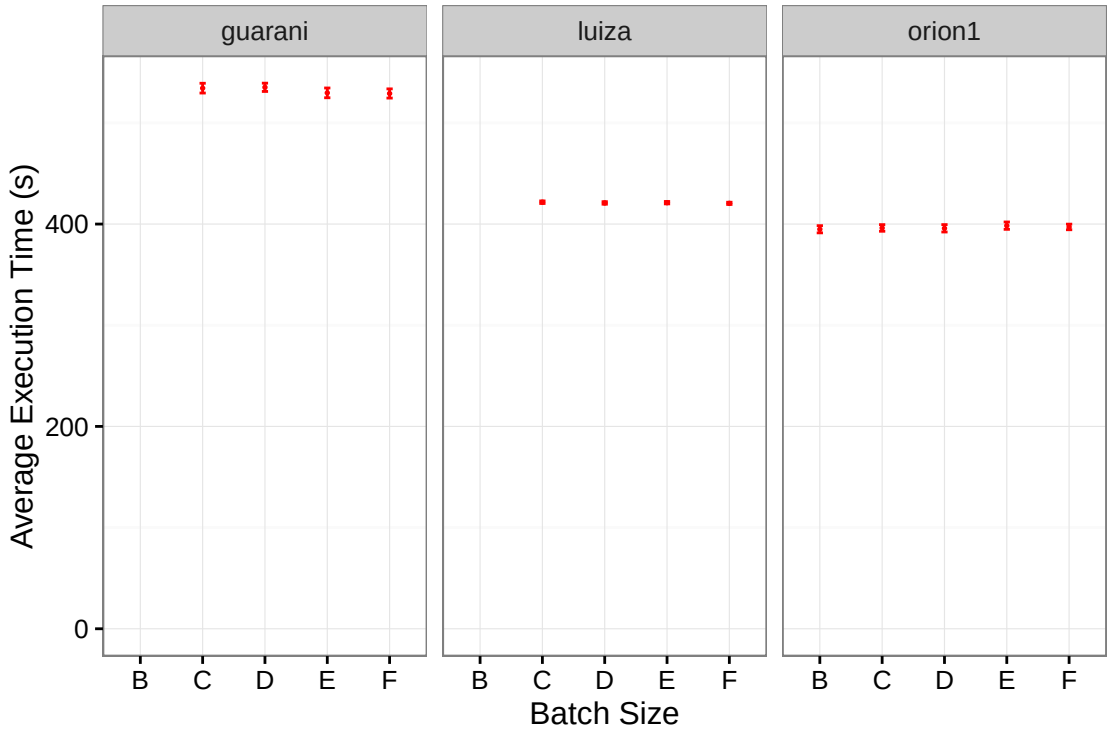
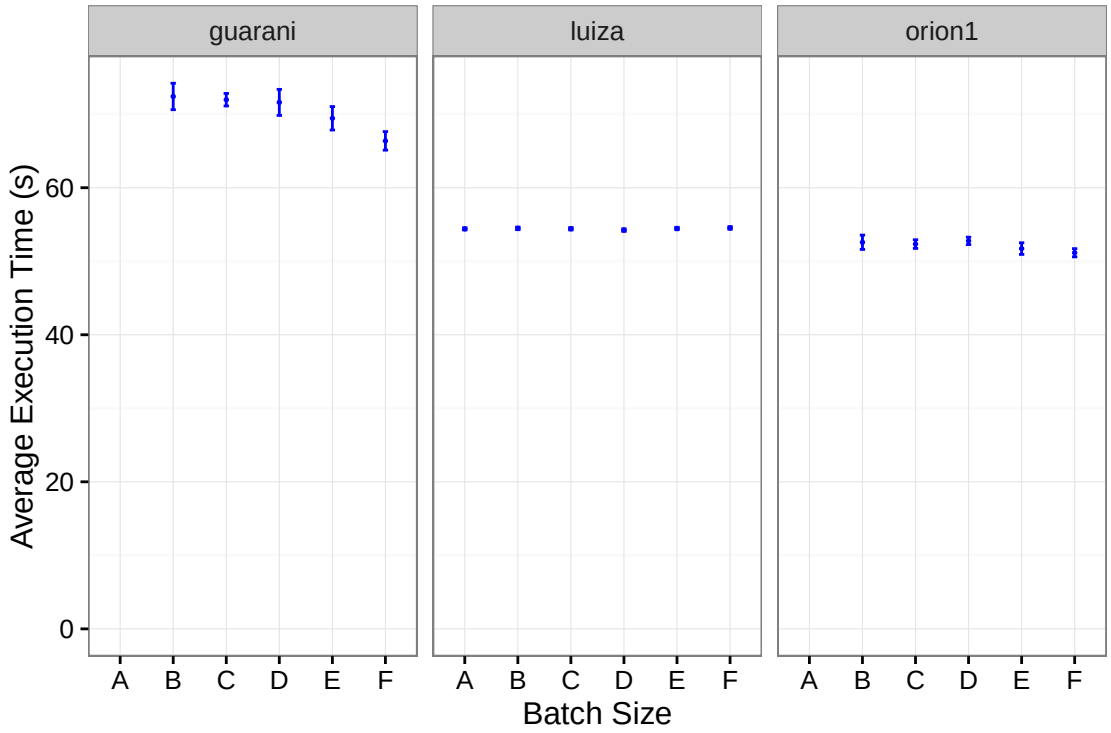
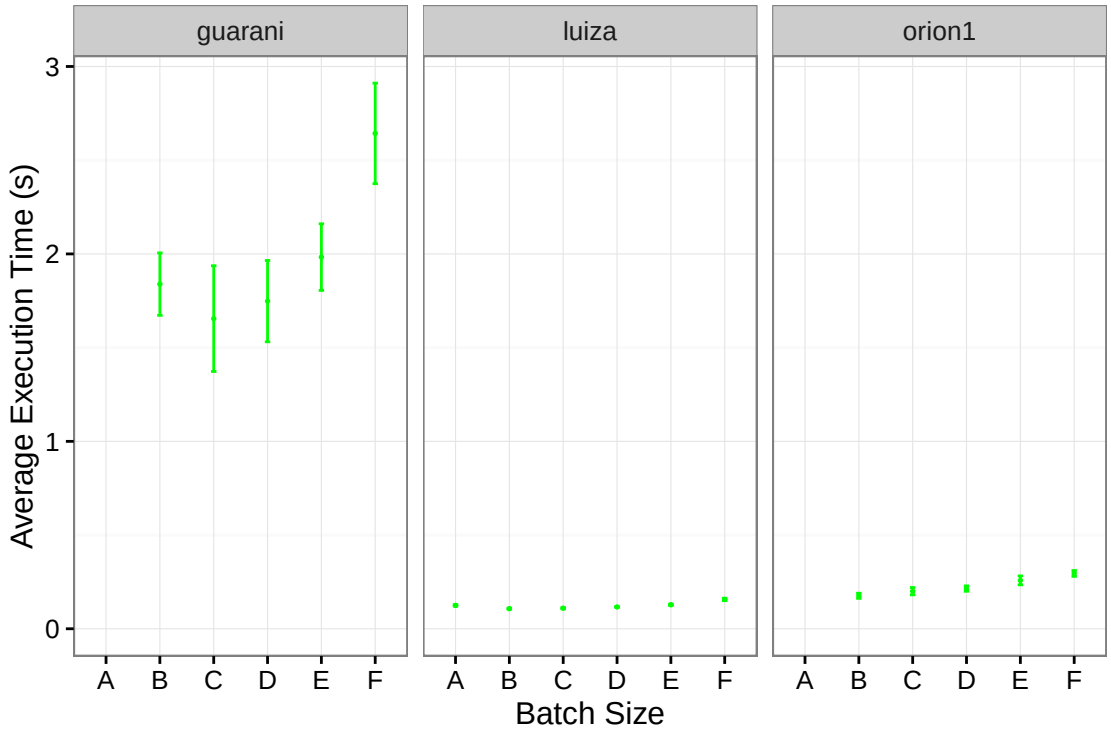


Figure 6.13: Insertion time for the medium input



We can also observe by analysing Figures 6.15 and 6.16 that the difference of performance among the remote and local experiments rely exclusively on

Figure 6.14: Insertion time for the small input



the insertion time. That is probably the case since insertion requests have to be communicated through the network topology. Since the network is a contention point, in situations with high data volume, this affects directly the performance, as our results have shown.

Batch Insertion Traces

To have a more detailed sight of the insertion time behavior, we generated traces for the batch executions. These traces logged the duration of every batch insertion for each experiment. It is depicted, in Figures 6.17 and 6.18, 10 replications of the experiments for each platform and batch size. The graphics present a timeline in the horizontal axis of all the execution. There is one line for each replication. The colored segments portray the moments where a batch is being executed. For batch size **C**, for example, there are 4 insertions, thus, there are four colored segments for each replication. It is possible to see through these graphics that bigger batches take longer to execute, a detail that we had not considered in the hipotesis. In this way, we see that the insertion time does not significantly change between configurations for the same input size, as the number of queries is always the same, while the simulation itself is what impacts the run time. As we have seen before, the memory usage is what penalizes the performance.

Figure 6.15: Remote and local insertion times for the big input

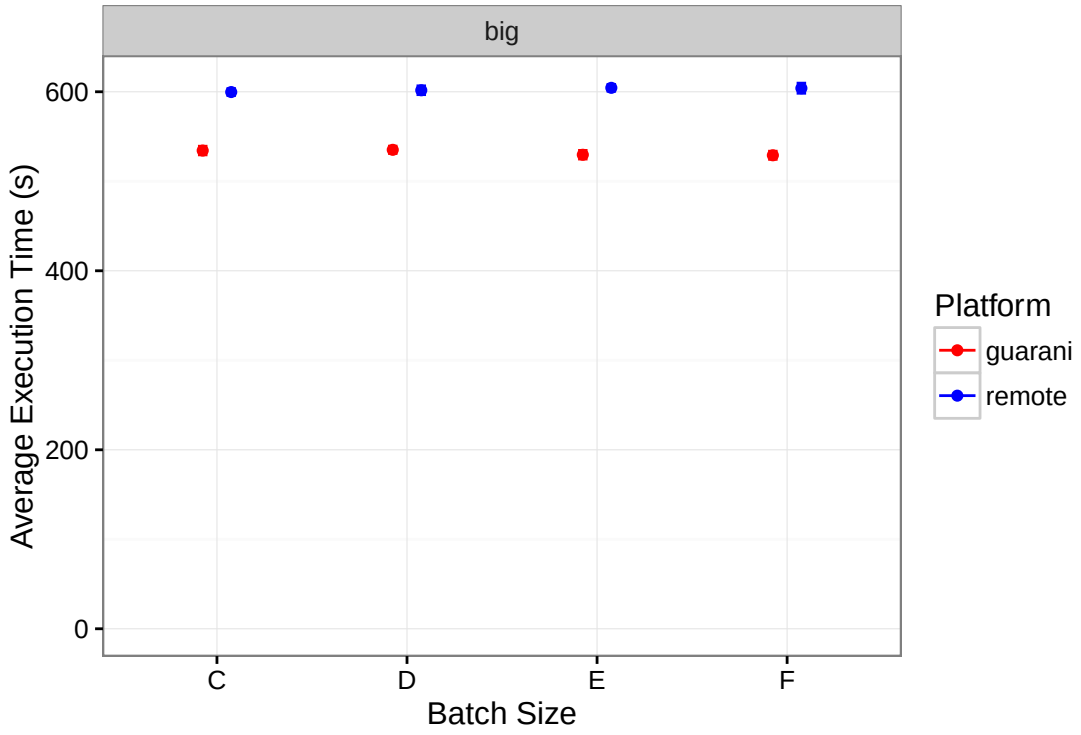
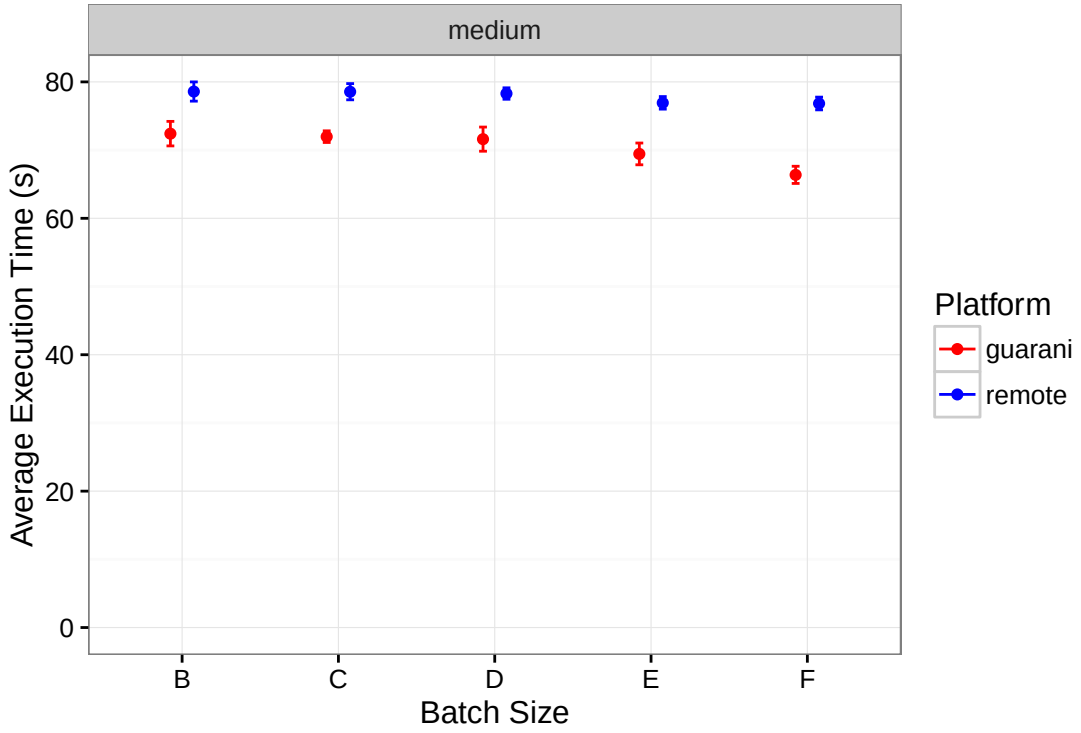


Figure 6.16: Remote and local insertion times for the medium input



For the small input, we can observe, in Figure 6.19, the opposite behavior. In these cases, it is better to insert all the data at once.

Figure 6.17: Results of batch sizes variability for the medium input

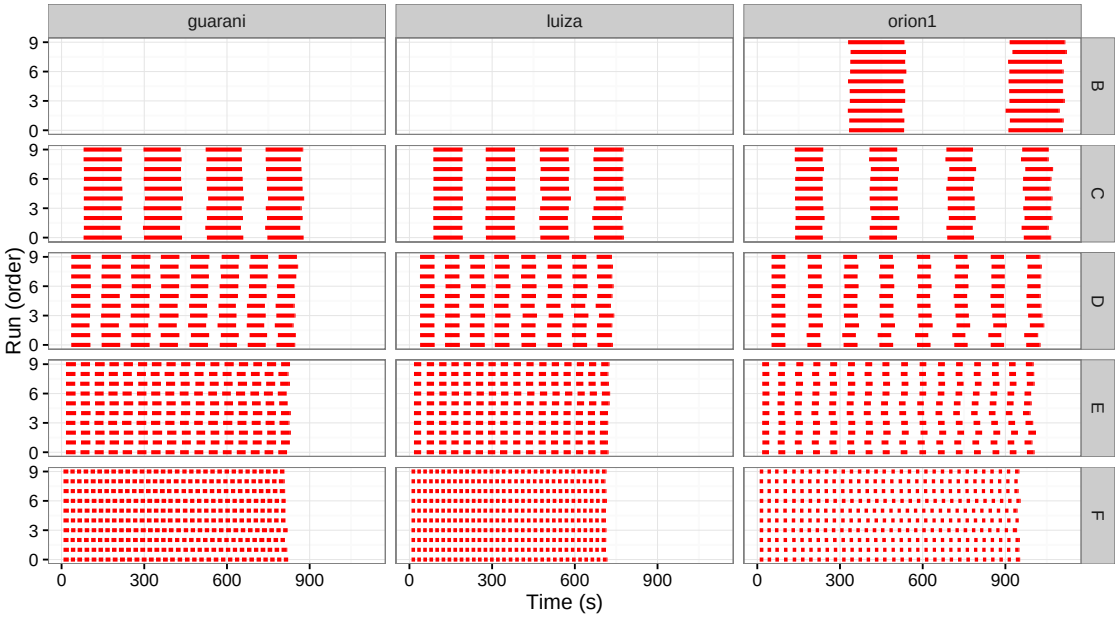


Figure 6.18: Results of batch sizes variability for the medium input

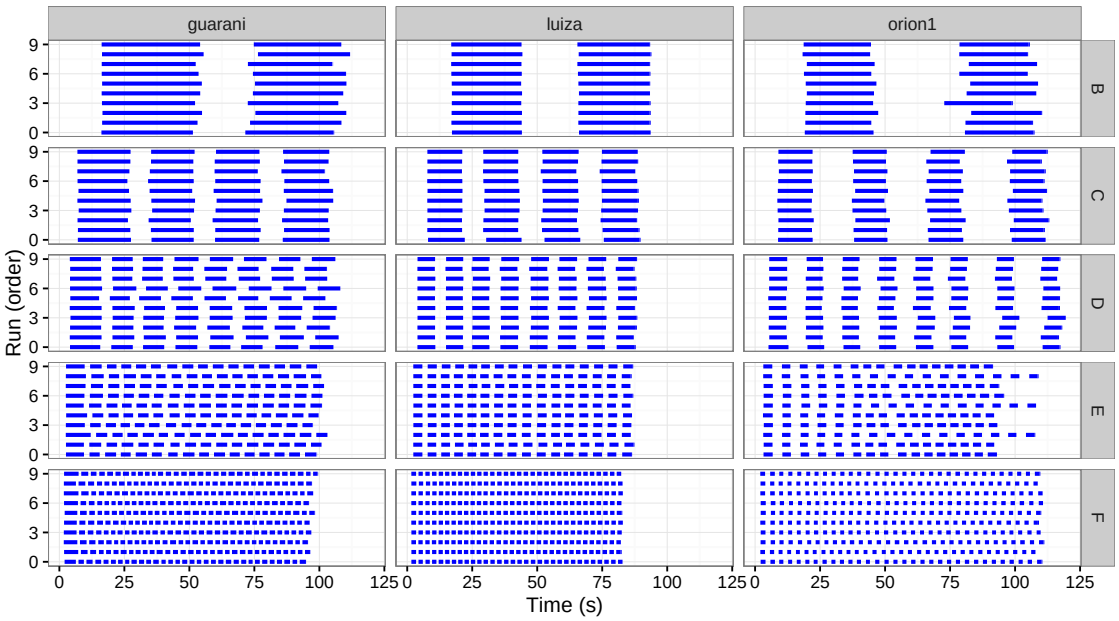
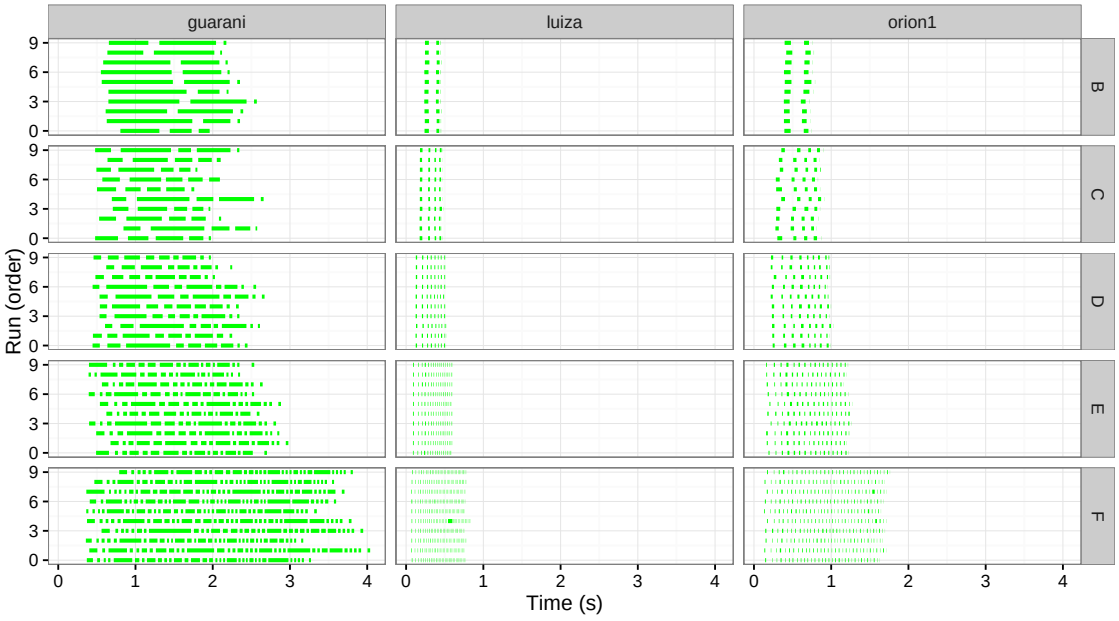


Figure 6.19: Results of batch sizes variability for the small input



7 CONCLUSION

In this proposal, we have elaborated a different approach to an existing simulation tool for Pajé Trace files (PajeNG). We have observed issues in the PajeNG implementation, mainly regarding the lack of flexibility to handle the resulting data of the simulation and the ephemerality of these results. To improve the extensibility of the tool, we introduced the concept of plugins, where every generated entity is sent to. This permits the manipulation of the results totally deattached from the core simulation. We also validated this design by implementing plugins.

The separation of the core simulation from the data handling provided very good results. We were able to perform a simulation two times faster than the original tool. This is very relevant, since it opens several possibilities of creating high performance solutions. Building plugins for the solution was extremely important to validate and improve the proposal. The first approach consisted in having instrumentation points only for the events that generated complete entities, like **PajePopState**, which was soon discarded during the implementation of the **PajeInsertDBPlugin**. We saw that, for example, to insert a state entity, with foreign keys referring a container and a type, the object referenced must already be in the database. Thus, we noticed that, for the extensibility to apply to the maximum number of potential scenarios, it needed to give access to every type of event possible in the Pajé format. Implementing this plugin took a lot of time, and changes to it were made very frequently. It was interesting to observe that the modifications were only performed in the **PajeInsertDBPlugin** class, demonstrating clearly that the handling of the results was totally deattached from the core simulation.

The **PajeInsertDBPlugin** analysis presented some issues that the Java language may bring regarding memory usage. First, it can become very slow depending on the number of objects it is storing. Second, there must be a carefull analysis on the configuration of the JVM to assure that it will have a capacity consistent to the platform in which it is running. The **PajeDumpPlugin**, besides solving an issue, brings another concern which is the order in which the entities are presented, for example, states will be dumped before their containers. However, Aiyra is designed to facilitate the creation of new solutions to address these

issues and any other that might appear in the future.

For future work, it would be interesting to provide an interface for the plugin package to make it support multiple languages. There is also room to improve the organization of the entry points so that they can be even more generic. It would also be interesting to shift the options handling regarding each plugin to its proper class, instead of centralizing all in the controller. The way it is currently implemented, the user needs to modify the code in the controller package to add an argument to the plugin. In terms of analysis, the **PajeInsertDBPlugin** can be deeper explored by evaluating the performance of queries to the relational database created.

REFERENCES

BARRY, D. K. **Relational Model Concepts**. 2013. <http://www.service-architecture.com/articles/database/relational_model_concepts.html>. Accessed: 2016-05-09.

CRAWLEY, B. **Parser Generators: ANTLR vs JavaCC**. 2015. <https://dzone.com/articles/antlr-and-javacc-parser-generators>. Accessed: 2016-04-23.

HAMILTON, D. G. **JavaCC**. 2016. <http://www.computing.dcu.ie/~hamilton/teaching/CA448/notes/JavaCClex2.pdf>. Accessed: 2016-04-23.

HEUSER, C. A. **Projeto de Banco de Dados**. 6th. ed. [S.l.]: Ed. Sagra-Luzzato, 2008.

JAIN, R. **Art of Computer Systems Performance Analysis: Techniques for Experimental Design Measurements Simulation and Modeling**. 2nd. ed. [S.l.]: John Wiley & Sons, 2015.

JAVACC. **Java Compiler Compiler tm (JavaCC tm) - The Java Parser Generator**. 2016. <https://javacc.java.net/>. Accessed: 2016-04-23.

KERGOMMEAUX, J. C. de; STEIN, B. de O.; BERNARD, P. E. Pajé, an interactive visualization tool for tuning multi-threaded parallel applications. **Parallel Computing**, v. 26, n. 10, p. 1253–1274, 2000.

KOPP, M. **The impact of Garbage Collection on Java performance**. 2011. <http://apmblog.dynatrace.com/2011/03/24/the-impact-of-garbage-collection-on-java-performance/>. Accessed: 2016-05-17.

LAUX, D. M. **Processing command line arguments in Java: Case closed**: Facilitate command line argument processing for java tools with a simple helper class. 2004. <http://www.javaworld.com/article/2074849/core-java/processing-command-line-arguments-in-java--case-closed.html>. Accessed: 2016-02-28.

NARASIMHAN, B. **The Normal Distribution**. 1996. <http://statweb.stanford.edu/~naras/jsm/NormalDensity/NormalDensity.html>. Accessed: 2016-05-16.

ORACLE. **Java SE Technologies - Database**. 2016. <http://www.oracle.com/technetwork/java/javase/jdbc/index.html>. Accessed: 2016-05-09.

POINT, T. **ER Model - Basic Concepts**. 2016. <http://www.tutorialspoint.com/dbms/er_model_basic_concepts.htm>. Accessed: 2016-05-09.

POINT, T. **JDBC Tutorial**. 2016. <http://www.tutorialspoint.com/jdbc/jdbc-introduction.htm>. Accessed: 2016-05-14.

SAS. **Full Factorial Designs**. 2016. <http://www.jmp.com/support/help/Full_Factorial_Designs.shtml>. Accessed: 2016-04-24.

SCHNORR, L. M. **Pajé trace file format**. Porto Alegre, Brazil, 2016. <https://github.com/schnorr/pajeng/tree/master/doc/lang-paje>.

SCHNORR, L. M. **PajeNG - Trace Visualization Tool**. 2016. <https://github.com/schnorr/pajeng>. Accessed: 2016-05-09.

APPENDIX A — JAVACC TUTORIAL

To build a grammar that will be compiled by `JavaCC` you only need to create one file whith '.jj' extension. The structure of this file is the following:

```
options{
}
```

A set of optional flags. An example, is the flag `STATIC`, which means that there is only one parser for the JVM when set to true.

```
PARSER_BEGIN(MyGrammar)
public class MyGrammar {
}
PARSER_END(MyGrammar)
```

In this part, the Java code will be placed and it's the main class of the program. Notice that the class must have the same name as the generated parser.

```
TOKEN_MGR_DECLS:
{
}
```

The declarations used by the lexical analyser are placed in the **TOKEN_MGR_DECLS** function.

Below these three structures, comes the lexical analysis where the Token rules and parser actions can be written using a top-down approach. First, the Tokens are declared, always using the word "TOKEN" before. To exemplify the creation of a grammar in JavaCC, we will create a language that consists in the declaration of integer and char variables and assignments of values to these variables. All the declarations come first, then the assignments. No verification will be performed since it is just an example to clarify the JavaCC syntax. To declare tokens, we use the following notation:

```
TOKEN:
{
   < [NAME] : [EXPRESSION] >
}
```

For our example of language we will have the following tokens:

```
/* Integer Literals */
```

```
TOKEN :
{
  < INTEGER: "0" | ["0"-"9"] (["0"-"9")* >
}

/*Variables, assignments and char values*/
TOKEN :
{
  < VARIABLE: (["a"-"z", "A" - "Z"])+ >
  < ASSIGNMENT: "=" >
  < CHAR: (~["\""] | "\\"" (["n","r","\\","\'","\""])) >
}
/* Types */
TOKEN:
{
  < INTEGER_TYPE : "int" >
  < CHAR_TYPE: "char" >
}
```

As we can see in the definitions above, it is not necessary to explicit the word TOKEN for each one. It is usually separated to be better organized and easier to understand. Although the token's agroupation is not relevant, the order in which they are declared is. When an input matches more than one token specification, the one declared first will be considered. There is also another kind of regular expression production, which is the SKIP. Whatever matches the regular expression defined in the SKIP scope will not be treated by the parser. Example:

```
SKIP:
{
  "\n"
  \| "\t"
}
```

After the token declaration, comes the grammar rules. The rules are declared as methods, that can have return values or not. The structure of a method is the following:

```
[type] [name] ()
{}
{
  /* Rules */
```

```
}
```

The empty braces in the beginning of the method can be filled with variable declarations in Java. More Java code can be added in the middle of the rules by using braces. Inside the next braces, it is possible to assign tokens, regular expressions or even methods to the variables declared earlier. To refer to the tokens, we use its name between angular brackets. Example:

```
void parser()
{ int number; }
{
  number = <INTEGER>
}
```

The first method defined will be the entrance to the parser and it can contain methods inside that will be expanded later in the rules. The entrance for the language we are using as an example would be as follows:

```
void start()
{}
{
  declarations() assignments() <EOF>
}
```

EOF is a default token. It is important to guarantee that the file will be parsed until the end. By the definition of our first method, we assure that the declarations will obligatorily be in the beginning, and the assignments at the end. Next, we expand the two methods to address all the possibilities:

```
void declarations()
{}
{
  ((<INTEGER_TYPE> | <CHAR_TYPE>) <VARIABLE>)*
}

void assignments()
{}
{
  (<VARIABLE> <ASSIGNMENT> (<CHAR> | <INTEGER>))*
}
```

The multiplicity can be defined with the standard characters "*", "?", "+", just as in the lexer. This example is just one possible approach to define these

rules. For example, you can use another non-terminal to describe a value that will be assigned to a variable. In this case, the assignments() rule would be expanded as follows:

```
void assignments()
{}
{
  (<VARIABLE> <ASSIGNMENT> assignable() )*
}

void assignable():
{}
{
  <CHAR> | <INTEGER>
}
```

Usage with Java

In order to call the parser in a Java program, an object of the MyGrammar class needs to be instantiated:

```
MyGrammar parser = new MyGrammar(input);
```

Then, once there is an instance of the parser, it is possible to call the first method of the parser:

```
parser.start();
```

This code has a Java syntax and is placed in the main class presented previously. Between the declarations of PARSER_BEGIN and PARSER_END, any Java code can be placed to manipulate the results of the parsing.

```
PARSER_BEGIN(MyGrammar)
/* Imports */
public class MyGrammar {
    public static void main(String args []){
        /* Code to read the input */
        MyGrammar parser = new MyGrammar(input);
        parser.start();
       /* Java code to manipulate the parser results */
  }
```

```
10 }
11 PARSER_END(MyGrammar)
```