MARFL: An Intensional Language for Demand-Driven Management of Machine Learning Backends

Vashisht Marhwal

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Computer Science at

Concordia University

Montréal, Québec, Canada

March 2023

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By: **Vashisht Marhwal**

Entitled: **MARFL: An Intensional Language for Demand-Driven Management of Machine Learning Backends**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Dr. Yann-Gaël Guéhéneuc

_____ Examiner
Dr. Weiyi Shang

_____ Examiner
Dr. Yann-Gaël Guéhéneuc

_____ Supervisor
Dr. Joey Paquet

_____ Supervisor
Dr. Serguei A. Mokhov

Approved by _____
Dr. Lata Narayanan, Chair of Department

_____ 20 _____ _____
Dr. Mourad Debbabi, Dean
Gina Cody School of Engineering and Computer Science

Abstract

MARFL: An Intensional Language for Demand-Driven Management of Machine Learning Backends

Vashisht Marhwal

Artificial Intelligence (AI) is a rapidly evolving field that has transformed numerous industries and one of its key applications, Pattern Recognition, has been instrumental to the success of Large Language Models like ChatGPT, Bard, etc. However, scripting these advanced systems can be complex and challenging for some users. In this research, we propose a simpler scripting language to perform complex pattern recognition tasks.

We introduce a new intensional programming language, MARFL, which is an extension of the LUCID family supported by General Intensional Programming System (GIPSY). Our solution focuses on providing syntax and semantics for MARFL, which enables scripting of Modular A* Recognition Framework (MARF)-based applications as context aware, where the notion of context represents fine-grained configuration details of a given MARF instance. We adapt the concept of context to provide an easily comprehensible language that can perform complex pattern recognition tasks on a demand-driven system such as GIPSY. Our solution is also generic enough to handle other machine learning backends such as PyTorch or TensorFlow in the future.

We also provide a complete implementation of our approach, including a new compiler component and MARFL-specific execution engines within GIPSY. Our work extends the use of intensional programming to modeling and executing scripted pattern recognition tasks, which can be used for implementing different algorithmic specifications. Additionally, we utilize the demand-driven distributed computing capabilities of GIPSY to enable an efficient and scalable execution.

Acknowledgments

I would like to express my sincere gratitude to my supervisors, Dr. Joey Paquet and Dr. Serguei A. Mokhov, for their invaluable guidance and support throughout my research. Their feedback and encouragement have been instrumental in shaping this thesis.

I am also grateful to Concordia University for providing me with the resources and facilities necessary to carry out this research. I appreciate the efforts of the faculty and staff who have contributed to my education over my academic years.

To my friends, thank you for always being there for me, whether it was to listen to my frustrations or celebrate my successes. Your support has meant the world to me.

To my family, thank you for your love, encouragement, and unwavering belief in me. Your support has been the foundation of my success, and I couldn't have done it without you.

Finally, I want to give a special thanks to my girlfriend, Saumya. You have been my constant source of motivation, inspiration, and love throughout this journey. Your unwavering support, patience, and understanding have been invaluable. Thank you for being my rock, my guiding light and for always believing in me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Pattern recognition is a fast-growing area within the field of data analysis, which focuses on developing algorithms capable of identifying and interpreting complex patterns and regularities within datasets. It involves a multi-stage process that begins with the acquisition of raw and unprocessed data, which is then analyzed to detect underlying patterns and structures. Once identified, these patterns can be used to derive meaningful insights and inform decision-making in a range of domains, from finance and marketing to healthcare and engineering. At its core, pattern recognition is a powerful tool that enables us to extract valuable knowledge from large and complex datasets, providing a means to unlock new opportunities and solve some of the most pressing challenges.

In psychology and cognitive science, it can be used to analyze brain activity data, enabling researchers to identify patterns and connections between different regions of the brain. Biometric recognition systems use pattern recognition algorithms to analyze and identify unique physical and behavioral characteristics, such as fingerprints, facial features, or voice patterns. This technology is used in security systems, law enforcement, and access control applications. Pattern recognition algorithms are also used in forensic investigation to analyze and compare patterns found in evidence, such as fingerprints, DNA, and handwriting. Overall, pattern recognition provides a powerful tool for analyzing and understanding complex data

in a wide range of domains, enabling us to extract valuable insights and solve real-world problems.

This thesis presents an effort to expand the use of intensional programming paradigm and the science behind it to implement a formal model for scripting pattern recognition tasks for researchers and scientists. Our proposed solution is built to be a practical and easy to follow context-aware improvement over existing scripting methods. Our work elaborates on the required syntactical and semantic constructs of context definitions for the configuration of a new intensional programming language, i.e., Modular A* Recognition Framework Lucid (MARFL). Our proposed solution offers a practical, context-aware approach that inherits some properties from other Lucid dialects of intensional programming languages, e.g., GIPL and Forensic Lucid as well as context navigation operators like @ and # to switch and query context. We build the initial syntax, semantics, and brief type system to express the MARF configuration-as-context in the new language. The MARFL compiler and run-time system is designed within an established intensional evaluation platform called General Intensional Programming System (GIPSY). Intensionality and compatibility within the intensional programming paradigm makes this an easier to follow reproducible approach in the research community. Utilizing the GIPSY environment enables the integration of parallel intensional approach with a highly efficient demand-driven distributed computing platform.

In this chapter we begin by providing the reader with a brief overview of the research domain in Section 1.1, explaining the essential concepts of IPP and Lucid in Section 1.1.1 and Section 1.1.2. Followed by the principal motivations in Section 1.2 with an example of few motivational real world scenarios. Then, we present our problem statement and do a gap analysis for existing technologies in Section 1.3. Then, we provide high-level objectives for this dissertation in Section 1.4. In Section 1.5, we define the primary objectives of our thesis followed by a summary of the chapter in Section 1.6 which also outlines the structure for the rest of this dissertation in Section 1.7.

1.1 Research Domain

In order to describe and understand the domain of this research work, we discuss the two main concepts which is involved in our work – the intensional programming paradigm and the LUCID intensional programming language.

1.1.1 Intensional Programming Paradigm

The intensional programming paradigm finds its roots in intensional logic. Intensional logic is a family of mathematical formal systems that allows expressions evaluation to be dependent on an inherent context [11]. It first emerged while researching in natural language understanding and one of the early research into the concept of intension by Carnap [12] suggests that the intention of a natural language expression is its true meaning, whose value relies on the context in which it is uttered. The extension of that statement is a set of values, each associated with the context of utterance. There are a wide variety of intensional logics, where each variety can cover different problems pertaining to *time*, *belief*, *space*, etc. As described in Paquet's thesis [1], the possible contexts of evaluation can be viewed as points in a multidimensional space. For instance, expressions defined under temporal logic vary in *time dimension*, i.e., it can have different values, depending on the instant in which it is evaluated. Basically, intensional logic allows the addition of dimensions to logical expressions whereas non-intensional logic is viewed as constant in all the dimensions. To navigate in a given context space, intensional operators are defined. Along with these operators, some dimension tags are required to provide placeholders along dimensions. These tags define the context for evaluating intensional expressions. For example, an expression defined as:

> E: Ice Hockey is **now** the official winter sport **here**.

Such an expression would be considered intensional because the truth value of this expression depends on the context in which its evaluated. The explicit context of this expression is [`place`: **here**, `time`: **now**], where `place` and `time` are dimensions and

here and **now** are place holder tags along those dimensions respectively. Dimension names along with tag values form the evaluation context in this expression. If we fix **now** to **present** and assume it is a *constant*, this expression will be evaluated to true if **here** is **Canada** but if the place holder tag for dimension `place` changes to some other country like **India**, then this expression might have a different value. Similarly, if we assume **here** to be *constant* as **Canada**, the expression will be evaluate to *false* if `time` is set to any year before **1994** (as hockey was only declared as the official winter sport of Canada in 1994). Hence, the core concept of context in intensional logic is the principle basis of Intensional Programming Languages (IPL).

The programming paradigm retains two important aspects of intensional logic: first, at the syntactic level, are context-switching operators knows as intensional operators, second, at semantic level, is the use of possible world semantics. IPL follows a more declarative way of programming than procedural languages while dealing with infinite entities of ordinary data values. This infinite nature of IPL allows it to describe behaviour of systems that can change with time or properties depending on more than one contextual parameters. However, traditional approaches cannot be applied to these infinite entities since that would require infinite amount of time or space. To solve this, the computation model of *eduction* is used. It is based on **lazy demand driven** strategy which delays evaluation of an expression until its value is needed. We will go into more detail about eduction in Chapter 2.

1.1.2 Lucid

Lucid, which was originally developed as a program verification language by Ashcroft and Wadge [13], evolved into a dataflow language over its several iterations. Gradually as its applicability got wider, it became a multidimensional intensional programming language [14], whose semantics is based on the possible world semantics [15–17] of intensional logic. It is a functional language in which expressions and their valuations are allowed to vary in an arbitrary number of dimensions. In fact, today it has become a family of intentional languages that promote context-aware evaluation in a given set of dimensions over a given set of tag values. Any programs written in some Lucid

dialect consist of expressions that may contain subexpressions which are evaluated at a certain context which represent a point in a multidimensional context space. As explained in the example in Section 1.1.1, each program will have a set of dimensions in which an expression can vary with a corresponding set of tags over each dimension. This context is represented as as et of $<dimension : tag>$ mappings.

The General Intensional Programming Language (GIPL), a general implementation of Lucid, provides definitions for the two fundamental operators @ and # to navigate (context switching and context query) in a given context space [1]. The GIPL is defined using just the two intensional operators @ and #. It has been demonstrated in the past works that other dialects of Lucid can be translated back into GIPL. One of the main objectives to develop Lucid and its dialects is to allow easier human comprehension of programs pertaining to a certain problem. The fundamental syntax and semantics of Lucid are rather simple, hence allowing easier compiler implementations providing the flexibility to develop various dialects for application domain specific purpose while still using the core baseline as a fundamental building block. Availability of extensive libraries and mutual advantage of eductive evaluation contributes to choosing this language for this dissertation. We will briefly discuss several dialects of Lucid with their core properties and concrete features we inherit for MARFL in Chapter 2.

1.2 Motivation

Multimedia pattern recognition applications have evolved to meet demands in many domains such as cognitive sciences, biomedical systems, forensic analysis systems, etc. Several frameworks for general or specific tasks have been developed given the expanding possibilities this field offers. One such framework, that our work is heavily based on, is Modular A* Recognition Framework (MARF). MARF is an open-source research framework, implemented in Java, which has a collection of general-purpose pattern-recognition algorithms, APIs, and concrete algorithms [18]. Although this framework was initially designed for audio recognition, it has since evolved and

is no longer only restricted to audio. In addition to handling natural language processing tasks, it can act as a re-usable JAVA library for developing applications due to its generality. We use MARF as an investigation platform to select the best fit configuration for any given pattern recognition task. Although the developers of MARF made every effort to make it accessible and easier to follow with sample applications, the usage still requires a skilled JAVA programmer who has a relatively good understanding of MARF's design structure. Often, scientists and researchers who want to use the framework's pattern recognition algorithms and applications in their simulations and experiments do not fall into this category. Thus, a need of a scripting medium that is simpler and more natural than JAVA, that can cater to the user's intent while specifying all parts or configuration context parameters of these applications, rises.

Moreover, the size of training datasets for any pattern recognition task has increased tenfold compared to what it was just 5 years back. To solve this, popular machine learning libraries like TensorFlow and PyTorch provide support for distributed computation as it can handle the scalability factor of the growing training demands. However, the usage of such libraries require explicit coding for distributed training which can be challenging and time-consuming for researchers who do not have experience with distributed computing. One of our overarching motivation other than providing a simpler way to script pattern recognition tasks, is to provide users with a framework that utilizes distributed computation without the need for explicit coding. By leveraging existing distributed computing infrastructures, our solution will enable users to execute computationally intensive pattern recognition tasks on a distributed system with no extra effort.

The foundational structure of our work is primarily motivated by Mokhov's research work in [6] that presents the initial syntax and semantics for Modular A* Recognition Framework LUCID, or MARFL for short. While his work allows defining context expressions that allow scripting MARF based applications in a context-aware environment in a given instance, it was only preliminary research. Our focus is to extend and improve the initial syntactical and semantic definitions and use them

to build a compiler on an intensional evaluation platform, such as GIPSY. The General Intensional Programming System (GIPSY) is an open-source multi-language intensional programming platform implemented primarily in JAVA [19–21]. It is used to investigate the implementation and investigation of several LUCID dialects and their properties, about which we will discuss more in Section 2.1. Its multi-tier distributed system execution architecture evaluates LUCID programs using a demand-driven distributed generator-worker architecture. We will go into more detail about its architecture as well as why GIPSY is our platform of choice in Section 2.3. Further, We aim to provide context specification for multimedia pattern recognition tasks that use the available MARF resources for its applications. We will explore topics related to GIPSY and MARF in detail in Chapter 2.

1.2.1 Motivational Scenarios

We came up with the following primary scenarios that are based on actual and potential real-life use-cases which help us set objectives for our solution. As discussed earlier, our aim is to provide researchers with a simpler scripting method to perform scalable pattern recognition tasks. Our scenarios described below assume that the user is familiar with the requirements of our solution and has access to our evaluation platform i.e GIPSY.

- **Scenario I: Speech recognition for language learning applications**

  In this scenario, the researcher is developing a language learning application that can identify if the user is correctly pronouncing words or phrases.The application uses speech recognition technology to enable learners to practice speaking and receive personalized feedback on their pronunciation. To develop a robust application which can cater to wide variety of user accents, the researcher needs to meet following requirements:

  1. Select the best fit pattern recognition pipeline for speech recognition that also meets the functional requirements for the language learning

application. The researcher will have to perform a comparative study between all available algorithms.

2. To ensure accuracy of the selected algorithm, the researcher needs to train it with a wide range of accents and dialects to ensure it can accurately recognize speech from different regions.
3. The hyper parameters of the selected algorithm need to be tuned to provide good precision as the application offers personalized feedback to the user based on input.

To meet most of these requirements, the researcher can use the solution provided in this research work to script several pattern recognition pipelines which may use different machine learning sub-systems to find the best fit for their application. Moreover, MARF already provides a collection of concrete algorithms which also include speech recognition algorithms that can be used for the language learning application. This can save the researcher time and effort as they do not have to implement the algorithms from scratch. Applications provided in MARF has been used for similar purpose in [22]. Additionally, the researcher can leverage the power of our solution to utilize distributed computation without explicit coding, allowing them to train on large datasets with ease and improve the training process.

- **Scenario II: Fixing Software Vulnerabilities Through Code Analysis**

  In this scenario, our primary user is a software tester trying to ensure that a given software source code is free of any vulnerabilities that may be exploited by attackers. To perform comprehensive quality assurance testing, the tester needs to meet the following requirements:

  1. Leverage appropriate pattern recognition algorithms to analyze the software code and identify any potential vulnerabilities. These algorithms should be able to analyze speech, audio, and text input to identify security gaps in the system.

2. Perform comprehensive testing to ensure that the software is free from vulnerabilities. This involves running the developed classification scripts again and verifying that the source code is free from vulnerabilities.

Our proposed solution can be used to leverage available speech, text, and audio algorithms available in MARF. Our work also uses a MARF based framework called MARFCAT [23] which can be used by the tester to perform vulnerability analysis or to classify exploitable code in a given dataset. As discussed before, the tester would not require to explicitly code for distributed computation since our implementation platform handles that inherently without any explicit coding.

In short, our solution should be able to script computation intensive pattern recognition tasks in a simpler way when compared to existing machine learning libraries and should be able to distribute the computation of sub processes in an inherent manner with minimal to no user interference.

1.3 Problem Statement and Gap Analysis

One of the most popular and widely used programming language for machine learning and pattern recognition tasks is PYTHON. Due to its huge community support and abundance of libraries like NumPy, SciPy, TensorFlow, PyTorch etc. that can cater to complex machine learning tasks, Python is a widely sought-after skill in the scientific programming field. Since it can be used to write programs in a Map-Reduce model, it can even handle big data analysis in a Hadoop ecosystem. But like every programming language it requires its user to learn, understand and then experiment with different libraries to achieve the required result. So a researcher will be required to understand each different library, its parameters, and methods for carrying out specific experiments. Also, PYTHON is not inherently parallel, that is the researcher will have to code, after understanding several modules it does offer, to write an experiment that can be carried out in a parallel computing model. With each passing

year, the amount of data that can be used to train a machine learning model increases. The libraries like TensorFlow do offer distributed computing via an api but it requires the user to have a basic understanding of how a distributed architecture works. To keep up with the demand and to shorten the time for training an entire pipeline, one requires either high powered GPUs or a parallel implementation for concurrent training or a combination of both. Also, it certainly does not offer the user to change scientific libraries at backend as per user configuration for comparative study.

Although MARF provides researchers and scientists with a platform for practical comparison of algorithms in a uniform environment, all its application requires the user to have some knowledge of working in JAVA programming language. Since MARF, its derivatives, and applications can be used for tasks beyond audio processing, the developers built several useful JAVA classes to handle several file types. The files can be read through a file system or an URI. Essentially these files are used for training the algorithms for any (un)supervised machine learning model that is part of the MARF pipeline. It is not reasonable or expected from the researcher to learn Java before performing any pattern recognition experiment. Therefore, we require a scripting medium for MARF applications that is human comprehensive and can manipulate its configurations.

The very first approach for scripting a contextually aware LUCID language for handling MARF configurations was given by Mokhov in 2008 [6]. This approach provided the initial syntax and semantics for MARFL. The author also provided a translated sample program which followed the proposed specifications of MARFL. However, it was only preliminary research that had practical implications on usability of MARF's resources as its future work. The author focused exclusively on context specification for multimedia pattern recognition tasks and available MARF resources for its applications. While the initial syntax was used as a basis for MARFL of this research work, we made considerable changes to the proposed syntax to satisfy our requirements and make the specifications in-line with the required GIPL syntax and implementation. At the base level, the LUCID intensional programming language has been in development for more than 40 years. Its represents a paradigm that

evolves constantly. However, its concrete applicability needs to be proven to become accepted. A lot of LUCID dialects that are context-aware with a specific purpose in mind, which we will discuss more in Chapter 2, have been created.

Overall, there are some unaddressed problems and gaps with theories, techniques and technologies used, which we will summarize below.

1. Procedural programming languages like Python:
   (a) Do not provide human comprehensible syntax at the same level as an IPL, for example from the perspective of the expression of inherently multi-dimensional expressions without having explicitly express extensionally the required procedures to augment their values.
   (b) Are not inherently parallel while performing complex scientific experiments.
   (c) Do not offer the user to dynamically change library backends or configuration.
2. The proposed MARFL:
   (a) Requires formal concrete syntax that can handle several context-aware specifications and its subsequent parameters in MARF.
   (b) Needs several additions in concrete syntax to handle libraries other than MARF.
3. GIPSY:
   (a) Needs update to the runtime system to support MARFL.
   (b) Needs a compiler for MARFL.
   (c) Needs a MARFL Semantic Analyzer that can account for new node types in AST corresponding to the MARFL specific extensions introduced.
   (d) Needs an *eductive* evaluation interpreter which can read the AST to generate specific demands.

(e) Needs an execution engine that can delegate demands to problem specific sub-systems such as MARF or TensorFlow.

1.4 Proposed Solution

This research work focuses on the refinement and implementation of MARFL concrete syntax introduced in [6], followed by a formal implementation of the refined syntax and updated semantics into the GIPSY compiler and runtime system. At the same time, we update the Intensional Programming Paradigm with new context operators introduced in MARFL.

The majority of this dissertation is focused in creating the MARFL dialect of Lucid so that an easy comprehensible scripting medium for pattern recognition tasks can be provided to researchers for performing and evaluating experiments. MARFL is a combination of the syntax and its operational semantics inference rules inside a uniform demand-driven environment.

Thus, our approach is tailored to addressing a subset of problems and gaps outlined in the previous section. Specifically:

1. Addressing the conventional programming language gaps.

   One of the major goals of this research work is to provide a scripting medium that is human comprehensible and yet can perform the complex scientific tasks such as training (un)supervised machine learning models for classification of data in several domains. We want to reduce verbosity of a pattern recognition script by using the inherent context-aware model of Lucid. We also aim to resolve the drawbacks of conventional approach that are usability and lack of inherent parallel computing. we benefit by using a parallel demand-driven context-aware evaluation platform that the current approach lacks. Choosing a Lucid approach over a platform like GIPSY is major solution to these problems.

2. Addressing the MARFL improvements.

The Lucid family of languages is built around intensional logic that takes the notion of context as explicit, central, as well as a first-class value [24, 25] that can be passed around as function parameters and have a set of operators defined upon. Our solution greatly draws on this notion by providing a context-aware hierarchical way to specify MARF configurations for model specification. We use the higher-order context hierarchy to specify different parameters and its sub-parameters in our configuration. Such a proposition has been already done in [26] to define and modify expressions in a cyberforensic context.

With regards to syntax and semantics for MARFL, we benefit in large part, as it has been initially defined by Mokhov in [6]. The initial syntax is largely based on its predecessors and codecessor Lucid dialects, such as GIPL, Indexical Lucid [27], Lucx [2], and Forensic Lucid [26] bound by high order intensional logic (HOIL) [24, 25] that is behind them. This work continues to formally specify the operational semantics of MARFL language extending the previous related work [6].

We inherit the definitions of specification of hierarchical context expressions and the operators used while modelling them from Forensic Lucid. Implementing MARFL in GIPSY provides us a demand-driven evaluation of Lucid programs in a more efficient way when compared to other conventional languages like Python for pattern recognition tasks.

3. Addressing the GIPSY gaps.

   GIPSY is used as a demand-driven evaluation platform for several Lucid dialects. We need to provide a compiler for MARFL that can interpret a MARFL Lucid program. We also need to provide the semantic analyser based on the operational semantics defined for MARFL. The analyser should be able to handle new node types in AST which are introduced from MARFL specific extensions. We aim to adapt similar methods defined in [3, 5, 28, 29] for building the semantic analyser for predecessor dialects like Lucx. Following that, we need to update framework with an Interpreter that can handle and produce the

intermediary configuration files required for a MARF instance.

1.5 Thesis Objectives

Given our motivational scenarios in Section 1.2.1, we list our major objectives that will help accomplish our goals. Each of these objectives are necessary to describe our solution, its functionality and eventual execution on a demand-driven platform. The objectives are given a specific identifier so we can refer them and make it easier for the reader to follow.

1. **O1: MARFL syntax and operational semantics** (Chapter 3)

   Our first primary objective towards fulfilment of an intensional scripting medium is to provide the concrete syntax as well as operational semantics of our proposed language, MARFL. Based on our scenarios, the user needs to have a simple intensional way to script pattern recognition tasks which do not require extensive knowledge of programming. The specifications should also include ways to execute multiple tasks so that the user can perform an extensive comparative analysis of used pattern recognition algorithms.

2. **O2: MARFL compiler** (Chapter 4, Section 4.1)

   Following **O1**, we need to provide a compiler that can conform to the rules of syntax defined, perform semantic analysis by doing type checking, and ensure that the program semantics are correct. It needs to be implemented inside the GIPSY compiler framework, GIPC. This essential component compiles the MARFL program written by the user in any of the discussed scenario and is responsible for giving an intermediate representation of the program which can be used in further execution.

3. **O3: MARFL-specific execution engine** (Chapter 4, Section 4.2.2)

   To execute a MARFL program, we need to implement and redesign some basic architectural components that are able to evaluate and understand the

new MARFL specifications. To facilitate execution of different machine learning sub-systems, our implementation platform needs to be updated with functionality to handle these sub-systems.

4. **O4: Executing MARFL Programs** (Chapter 4, Section 4.3)

   Following our implementation objectives **O1** through **O3**, the system should be able to run the pattern recognition task specified inside a MARFL program and provide the user with results. As described in our motivational scenarios, the user should be allowed to script these tasks without the need of having extensive knowledge of any particular programming language essentially by passing the need of writing or re-writing long lines of code. Therefore, this is one of our two major functional objectives.

5. **O5: Distributed execution of a MARFL program** (Chapter 5, Section 5.4)

   As discussed in our motivation for this research work as well as potential scenarios, the system should be able to perform computationally intensive tasks in a distributed manner so as to reduce training or classification times. This should require no explicit coding from the user and must be handled automatically by the implementation platform. This is our second major functional objective which will help fulfil our overarching goal of doing this research work.

1.6 Summary

To summarise, we will show that the intensional approach with a Lucid-based dialect to a pattern recognition task is a far more practical and comprehensible approach for researchers and scientists as compared to using plain Python or Java libraries. Building upon the concept of first-class context value, introduced in Lucx [29] and then inherited by Forensic Lucid [26], we will define a novel Lucid dialect with a new set of predefined context operators, that can script high order context-aware programs which can run inside the GIPSY environment. We will then define the

operational semantics of our dialect, subsequently integrating a semantic analyser inside GIPSY to allow use specific SIPL nodes introduced in our dialect. Unlike the conventional approaches, our evaluation method will be inherently parallel due to GIPSY's demand-driven general eduction engine (GEE) so the user does not have to code separate parallelized programs.

1.7 Thesis Organisation

We begin by reviewing the relevant background knowledge in Chapter 2 where we start with the notion of Intensional Logic, Lucid, and its several dialects, followed by specifications about the MARF package. Then we talk in-depth about the GIPSY architecture. Chapter 3 presents the theoretical basis of this work, including the syntax and operational semantics for MARFL. Chapter 4 specifies the implementation details of how MARFL compiler is constructed, integration of semantic analyser with parser, and the several updates to the GIPSY runtime system like addition of a MARFL specific interpreter, etc. Chapter 5 presents the evaluation methods used to test the modules implemented in Chapter 4 and the subsequent results of our experiments. Chapter 6 concludes our research work with a discussion about potential future work.

Chapter 2

Background

2.1 The Lucid Programming Language Family

In this chapter we first review the background on the several LUCID dialects as instantiations of various HOIL realizations giving an overview in Section 2.1.1 followed by the several dialects in Section 2.1.3. Section 2.2 talks about the MARF platform, from which this work heavily inspires from. We then explain GIPSY, the evaluation platform used for this research work in Section 2.3. We then provide a summary of the findings in Section 2.4. The use of LUCID is crucial in this thesis due to its ability to efficiently process streams of data and its simple syntax, as well as its strong mathematical and theoretical foundation, making it well-suited for representing and reasoning within large amounts of information.

2.1.1 Lucid Overview

As discussed in Section 1.1.2, LUCID [13, 14, 30–32] is a dataflow intensional and functional programming language which has now evolved to a family of languages that are built using intensional logic upon a context-aware demand-driven parallel computation model [33]. A LUCID program is an expression that may have subexpressions that need to be evaluated at a certain context. Essentially, it is a *declarative* language declared in an embedded *where* clause. The very first generic

version of LUCID, the General Intensional Programming Language (GIPL) [1], defines two basic operators @ and # to navigate (switch and query) in the context space. This context space is where the expressions are evaluated and can potentially yield different results based on their coordinates within this space, which is essentially a set of dimensions. It has been demonstrated that other languages within the LUCID family that utilize intensional programming can be converted into GIPL [1,4,34,35]. However, our work more closely follows the implementation presented by LUCX [2] and FORENSIC LUCID [26] where newly introduced AST nodes are handled by the semantic analyzer instead of requiring specific translation rules.

2.1.1.1 Fundamental Syntax and Semantics

E	::=	id
	|	E(E,...,E)
	|	E[E,...,E](E,...,E)
	|	**if** E **then** E **else** E **fi**
	|	**#** E
	|	E @ [E:E]
	|	E @ E
	|	E where Q end;
	|	[E:E,...,E:E]
	|	**iseod** E;
Q	::=	**dimension** id,...,id;
	|	id = E;
	|	id(id,....,id) = E;
	|	id[id,...,id](id,....,id) = E;
	|	QQ

Figure 1: GIPL syntax expressions

The fundamental syntax and semantics of LUCID were given by Ashcroft and Wadge in the 70s-90s which evolved with each iteration [13]. The syntax is rather simple which allows an easier compiler implementation as well as human comprehensible LUCID programs. As shown in GLU [36–38], LUCID can easily be integrated with other imperative programming languages which allows it to benefit from the eductive evaluation model. These factors were important in the decision to use this language in this research work. We provide examples of syntax and semantics of LUCID dialects that directly affect our work. LUCID has inherited several features

from its other dialects, and MARFL is an extension of same with its own new specific context operators. We define the syntax and semantics of MARFL in much more detail in Chapter 3.

2.1.1.2 Lucid Syntax

Figure 1 and Figure 2 presents the syntax for expressions, definitions, and operators for a hypothetical language that is a unification of the all the current Lucid dialects. This is only done for illustratory purposes so the reader can understand and familiarise with the necessary features of the Lucid dialects we inherit from.

op	::=	*intensional-op*
	|	*data-op*
intensional-op	::=	*i-unary-op*
	|	*i-binary-op*
i-unary-op	::=	**first** | **next** | **prev**
i-binary-op	::=	**fby** | **wvr** | **asa** | **upon**
data-op	::=	*unary-op*
	|	*binary-op*
unary-op	::=	! | − | **iseod**
binary-op	::=	*arith-op*
	|	*rel-op*
	|	*log-op*
arith-op	::=	`+` | `-` | ∗ | `/` | `%`
rel-op	::=	`<` | `>` | `<=` | `>=` | `==` | `!=`
log-op	::=	&& | `||`

Figure 2: GIPL operators

2.1.1.3 Operational Semantics of Lucid

Our chosen environment for implementing MARFL is GIPSY. GIPL is the generic counterpart of all the Lucid dialects. It only has two standard intensional operators: @ and #. In E @ C, @ is used for evaluating the expression E in context C, and #d is used to determine the position in dimension d of the current context of evaluation in the context space [1]. Specific Intensional Programming Languages (SIPL) are the other Lucid dialects which have their own attributes and objectives. The operational

semantics of GIPL are presented in Figure 3.

$$\mathbf{E_{cid}} : \frac{\mathcal{D}(id) = (\texttt{const}, c)}{\mathcal{D}, \mathcal{P} \vdash id : c} \quad (1)$$

$$\mathbf{E_{opid}} : \frac{\mathcal{D}(id) = (\texttt{op}, f)}{\mathcal{D}, \mathcal{P} \vdash id : id} \quad (2)$$

$$\mathbf{E_{did}} : \frac{\mathcal{D}(id) = (\texttt{dim})}{\mathcal{D}, \mathcal{P} \vdash id : id} \quad (3)$$

$$\mathbf{E_{fid}} : \frac{\mathcal{D}(id) = (\texttt{func}, id_i, E)}{\mathcal{D}, \mathcal{P} \vdash id : id} \quad (4)$$

$$\mathbf{E_{vid}} : \frac{\mathcal{D}(id) = (\texttt{var}, E) \qquad \mathcal{D}, \mathcal{P} \vdash E : v}{\mathcal{D}, \mathcal{P} \vdash id : v} \quad (5)$$

$$\mathbf{E_{op}} : \frac{\mathcal{D}, \mathcal{P} \vdash E : id \qquad \mathcal{D}(id) = (\texttt{op}, f) \qquad \mathcal{D}, \mathcal{P} \vdash E_i : v_i}{\mathcal{D}, \mathcal{P} \vdash E(E_1, \ldots, E_n) : f(v_1, \ldots, v_n)} \quad (6)$$

$$\mathbf{E_{fct}} : \frac{\mathcal{D}, \mathcal{P} \vdash E : id \qquad \mathcal{D}(id) = (\texttt{func}, id_i, E') \qquad \mathcal{D}, \mathcal{P} \vdash E'[id_i \leftarrow E_i] : v}{\mathcal{D}, \mathcal{P} \vdash E(E_1, \ldots, E_n) : v} \quad (7)$$

$$\mathbf{E_{c_T}} : \frac{\mathcal{D}, \mathcal{P} \vdash E : true \qquad \mathcal{D}, \mathcal{P} \vdash E' : v'}{\mathcal{D}, \mathcal{P} \vdash \texttt{if } E \texttt{ then } E' \texttt{ else } E'' : v'} \quad (8)$$

$$\mathbf{E_{c_F}} : \frac{\mathcal{D}, \mathcal{P} \vdash E : false \qquad \mathcal{D}, \mathcal{P} \vdash E'' : v''}{\mathcal{D}, \mathcal{P} \vdash \texttt{if } E \texttt{ then } E' \texttt{ else } E'' : v''} \quad (9)$$

$$\mathbf{E_{tag}} : \frac{\mathcal{D}, \mathcal{P} \vdash E : id \qquad \mathcal{D}(id) = (\texttt{dim})}{\mathcal{D}, \mathcal{P} \vdash \#E : \mathcal{P}(id)} \quad (10)$$

$$\mathbf{E_{at}} : \frac{\mathcal{D}, \mathcal{P} \vdash E' : id \qquad \mathcal{D}(id) = (\texttt{dim}) \qquad \mathcal{D}, \mathcal{P} \vdash E'' : v'' \qquad \mathcal{D}, \mathcal{P}\dagger[id \mapsto v''] \vdash E : v}{\mathcal{D}, \mathcal{P} \vdash E \ @E' \ E'' : v} \quad (11)$$

$$\mathbf{E_w} : \frac{\mathcal{D}, \mathcal{P} \vdash Q \ : \ \mathcal{D}', \mathcal{P}' \qquad \mathcal{D}', \mathcal{P}' \vdash E : v}{\mathcal{D}, \mathcal{P} \vdash E \texttt{ where } Q : v} \quad (12)$$

$$\mathbf{Q_{dim}} : \frac{}{\mathcal{D}, \mathcal{P} \vdash \texttt{dimension } id \ : \ \mathcal{D}\dagger[id \mapsto (\texttt{dim})], \mathcal{P}\dagger[id \mapsto 0]} \quad (13)$$

$$\mathbf{Q_{id}} : \frac{}{\mathcal{D}, \mathcal{P} \vdash id = E \ : \ \mathcal{D}\dagger[id \mapsto (\texttt{var}, E)], \mathcal{P}} \quad (14)$$

$$\mathbf{Q_{fid}} : \frac{}{\mathcal{D}, \mathcal{P} \vdash id(id_1, \ldots, id_n) = E \ : \ \mathcal{D}\dagger[id \mapsto (\texttt{func}, id_i, E)], \mathcal{P}} \quad (15)$$

$$\mathbf{QQ} : \frac{\mathcal{D}, \mathcal{P} \vdash Q \ : \ \mathcal{D}', \mathcal{P}' \qquad \mathcal{D}', \mathcal{P}' \vdash Q' : \mathcal{D}'', \mathcal{P}''}{\mathcal{D}, \mathcal{P} \vdash Q \ Q' \ : \ \mathcal{D}'', \mathcal{P}''} \quad (16)$$

Figure 3: Extract of operational semantics rules of GIPL [1]

Following is the description of the GIPL semantic rules as presented in [1]:

$$\mathcal{D} \vdash E : v \quad (17)$$

tells that under the definition environment $\mathcal{D}$, expression E would evaluate to value v.

$$\mathcal{D}, \mathcal{P} \vdash E : v \quad (18)$$

specifies that in the definition environment $\mathcal{D}$, and in the evaluation context $\mathcal{P}$ (sometimes also referred to as a *point* in the context space), expression E evaluates to v. The definition environment $\mathcal{D}$ retains the definitions of all of the identifiers that appear in a LUCID program, as created with the semantic rules 12–15 in Figure 3. It is therefore a partial function

$$\mathcal{D} : \mathbf{Id} \rightarrow \mathbf{IdEntry} \tag{19}$$

where **Id** is the set of all possible identifiers and **IdEntry**, summarized in Table 1, has five possible kinds of values, one for each of the kinds of identifier [33, 39]:

- *Dimensions* define the coordinate pairs, in which one can navigate with the `#` and `@` operators. Their **IdEntry** is simply (`dim`) [1].

- *Constants* are external entities that provide a single value, regardless of the context of evaluation. Examples are integers and Boolean values. Their **IdEntry** is $(\texttt{const}, c)$, where c is the value of the constant [1].

- *Data operators* are external entities that provide memoryless functions. Examples are the arithmetic and Boolean functions. The constants and data operators are said to define the *basic algebra* of the language. Their **IdEntry** is $(\texttt{op}, f)$, where f is the function itself [1].

- *Variables* carry the multidimensional streams. Their **IdEntry** is $(\texttt{var}, E)$, where E is the LUCID expression defining the variable. It should be noted that this semantics makes the assumption that all variable names are unique. This constraint is easy to overcome by performing compile-time renaming or using a nesting level environment scope when needed [1].

- *Functions* are non-recursive user-defined functions. Their **IdEntry** is $(\texttt{func}, id_i, E)$, where the id_i are the formal parameters to the function and E is the body of the function [1].

Table 1: Possible identifier types [1]

type	form
dimension	(`dim`)
constant	(`const`, c)
operator	(`op`, f)
variable	(`var`, E)
function	(`func`, id_i, E)

The evaluation context $\mathcal{P}$, which is changed when the `@` operator is evaluated, or a dimension is declared in a `where` clause, associates a *tag* (i.e., an index) to each relevant dimension. It is, therefore, a partial function

$$\mathcal{P} : \mathbf{Id} \to \mathbf{N} \tag{20}$$

Each type of identifier can only be used in the appropriate situations. Identifiers of type `op`, `func`, and `dim` evaluate to themselves (Figure 3, rules 2, 3, 4). Constant identifiers (`const`) evaluate to the corresponding constant (Figure 3, rule 1). Function calls, resolved by the $\mathbf{E_{fct}}$ rule (Figure 3, rule 7), require the renaming of the formal parameters into the actual parameters (as represented by $E'[id_i \leftarrow E_i]$). The function $\mathcal{P}' = \mathcal{P} \dagger [id \mapsto v'']$ specifies that $\mathcal{P}'(x)$ is v'' if $x = id$, and $\mathcal{P}(x)$ otherwise. The rule for the `where` clause, $\mathbf{E_w}$ (Figure 3, rule 12), which corresponds to the syntactic expression E `where` Q, evaluates E using the definitions Q therein. The additions to the definition environment $\mathcal{D}$ and context of evaluation $\mathcal{P}$ made by the $\mathbf{Q}$ rules (Figure 3, rules 13, 14, 15) are local to the current `where` clause. This is represented by the fact that the $\mathbf{E_w}$ rule returns neither $\mathcal{D}$ nor $\mathcal{P}$. The $\mathbf{Q_{dim}}$ rule adds a dimension to the definition environment and, as a convention, adds this dimension to the context of evaluation with the tag 0 (Figure 3, rule 13). The $\mathbf{Q_{id}}$ and $\mathbf{Q_{fid}}$ simply add variable and function identifiers along with their definition to the definition environment (Figure 3, rules 14, 15) [1, 33, 39].

As an extension to GIPL, Lucx's semantics introduced the *context as a first-class value*, as described by the rules in Figure 4. The semantic rule 22 (Figure 4) creates a context as a semantic item and returns it as a context $\mathcal{P}$ that can then be used by

the rule 23 to navigate to this context by making it override the current context. The

$$\mathbf{E_{\#(cxt)}} : \frac{}{\mathcal{D}, \mathcal{P} \vdash \# : \mathcal{P}} \quad (21)$$

$$\mathbf{E_{construction(cxt)}} : \frac{\begin{array}{l}\mathcal{D}, \mathcal{P} \vdash E_{d_j} : id_j \qquad \mathcal{D}(id_j) = (\texttt{dim}) \\ \mathcal{D}, \mathcal{P} \vdash E_{i_j} : v_j \qquad \mathcal{P}' = \mathcal{P}_0 \dagger [id_1 \mapsto v_1] \dagger \ldots \dagger [id_n \mapsto v_n]\end{array}}{\mathcal{D}, \mathcal{P} \vdash [E_{d_1} : E_{i_1}, E_{d_2} : E_{i_2}, \ldots, E_{d_n} : E_{i_n}] : \mathcal{P}'} \quad (22)$$

$$\mathbf{E_{at(cxt)}} : \frac{\mathcal{D}, \mathcal{P} \vdash E' : \mathcal{P}' \qquad \mathcal{D}, \mathcal{P} \dagger \mathcal{P}' \vdash E : v}{\mathcal{D}, \mathcal{P} \vdash E \mathbin{@} E' : v} \quad (23)$$

$$\mathbf{E_{.}} : \frac{\mathcal{D}, \mathcal{P} \vdash E_2 : id_2 \qquad \mathcal{D}(id_2) = (\texttt{dim})}{\mathcal{D}, \mathcal{P} \vdash E_1.E_2 : tag(E_1 \downarrow \{id_2\})} \quad (24)$$

$$\mathbf{E_{tuple}} : \frac{\mathcal{D}, \mathcal{P} \vdash E : id \qquad \mathcal{D} \dagger [id \mapsto (\texttt{dim})] \qquad \mathcal{P} \dagger [id \mapsto 0] \qquad \mathcal{D}, \mathcal{P} \vdash E_i : v_i}{\mathcal{D}, \mathcal{P} \vdash \langle E_1, E_2, \ldots, E_n \rangle E : v_1 \ fby.id \ v_2 \ fby.id \ \ldots \ v_n \ fby.id \ \texttt{eod}} \quad (25)$$

$$\mathbf{E_{select}} : \frac{E = [\texttt{d} : \texttt{v'}] \qquad E' = \langle \texttt{E}_1, \ldots, \texttt{E}_n \rangle \texttt{d} \qquad \mathcal{P}' = \mathcal{P} \dagger [d \mapsto v'] \qquad \mathcal{D}, \mathcal{P}' \vdash E' : v}{\mathcal{D}, \mathcal{P} \vdash select(E, E') : v} \quad (26)$$

$$\mathbf{E_{at(s)}} : \frac{\mathcal{D}, \mathcal{P} \vdash \mathcal{C} : \{\mathcal{P}_1, \ldots, \mathcal{P}_2\} \qquad \mathcal{D}, \mathcal{P}_{i:1\ldots m} \vdash E : v_i}{\mathcal{D}, \mathcal{P} \vdash E \mathbin{@} C : \{v_1, \ldots, v_m\}} \quad (27)$$

$$\mathbf{C_{box}} : \frac{\begin{array}{l}\mathcal{D}, \mathcal{P} \vdash E_{d_i} : id_i \qquad \mathcal{D}(id_i) = (\texttt{dim}) \\ \{E_1, \ldots, E_n\} = dim(\mathcal{P}_1) = \ldots = dim(\mathcal{P}_m) \\ E' = \texttt{f}_p(\texttt{tag}(\mathcal{P}_1), \ldots, \texttt{tag}(\mathcal{P}_m)) \qquad \mathcal{D}, \mathcal{P} \vdash E' : true\end{array}}{\mathcal{D}, \mathcal{P} \vdash Box[E_1, \ldots, E_n | E'] : \{\mathcal{P}_1, \ldots, \mathcal{P}_m\}} \quad (28)$$

$$\mathbf{C_{set}} : \frac{\mathcal{D}, \mathcal{P} \vdash E_{w:1\ldots m} : \mathcal{P}_m}{\mathcal{D}, \mathcal{P} \vdash \{E_1, \ldots, E_m\} : \{\mathcal{P}_1, \ldots, \mathcal{P}_w\}} \quad (29)$$

$$\mathbf{C_{op}} : \frac{\mathcal{D}, \mathcal{P} \vdash E : id \qquad \mathcal{D}(id) = (\texttt{cop}, f) \qquad \mathcal{D}, \mathcal{P} \vdash C_i : v_i}{\mathcal{D}, \mathcal{P} \vdash E(C_1, \ldots, C_n) : f(v_1, \ldots, v_n)} \quad (30)$$

$$\mathbf{C_{sop}} : \frac{\mathcal{D}, \mathcal{P} \vdash E : id \qquad \mathcal{D}(id) = (\texttt{sop}, f) \qquad \mathcal{D}, \mathcal{P} \vdash C_i : \{v_{i_1}, \ldots, v_{i_k}\}}{\mathcal{D}, \mathcal{P} \vdash E(C_1, \ldots, C_n) : f(\{v_{1_1}, \ldots, v_{1_s}\}, \ldots, \{v_{n_1}, \ldots, v_{n_m}\})} \quad (31)$$

Figure 4: Extract of operational semantics of LUCX [2, 3]

semantic rule 21 expresses that the **#** symbol evaluates to the current context. When used as a parameter to the context calculus operators, this allows for the generation of contexts relative to the current context of evaluation [1, 2, 29]

2.1.2 Eductive Model of Computation

Cargill at the University of Waterloo and May at the University of Warwick, independently developed the first operational model for computing LUCID programs [40]. It was directly based on the formal semantics of LUCID, which was based on Kripke's models and possible-worlds semantics [41, 42]. Ostrum later extended this technique with the implementation of the LUTHID interpreter [43] which was later adopted by by Faustini and Wadge [44] in their design of PLUCID interpreter.

This evaluation model is now known as eduction which helps us now for distributed execution [24] of Lucid programs [9, 10, 45–48].

```
#typedecl
Nat42;

#JAVA
class Nat42
{
    private int n;

    public Nat42()
    {
        n = 42;
    }

    public Nat42 inc()
    {
        n++;
        return this;
    }

    public void print()
    {
        System.out.println("n = " + n);
    }
}

#OBJECTIVELUCID

(N @.d 2).print[d]()
where
    dimension d;
    N = Nat42[d]() fby.d N.inc[d]();
end
```

Figure 5: The natural-numbers problem in Objective Lucid [4]

Figure 6 [5] represents the trace of an eduction tree during execution of a natural-numbers problem in a Objective Lucid program. To understand eduction, the problem was re-written in Objective Lucid and is presented in Figure 5. As highlighted in Figure 6, the outermost boxes labeled `{d:0}` etc. represent the current context of evaluation, demands for values to be computed under that context are represented as gray rectangular boxes with expressions, and the red boxes with the terminal bullets next to them represent the results of the computation of the expressions. In our proposed solution, such an eduction tree can be adapted to back-tracing in context evaluation, e.g., when the inner-most sub-parameters for an algorithm are traced back to the final result of the entire computation. The same method was adopted by Mokhov in his thesis, implementing Forensic Lucid for forensic evaluation [4, 26].

The concept of eduction can be described as a "tagged-token demand-driven

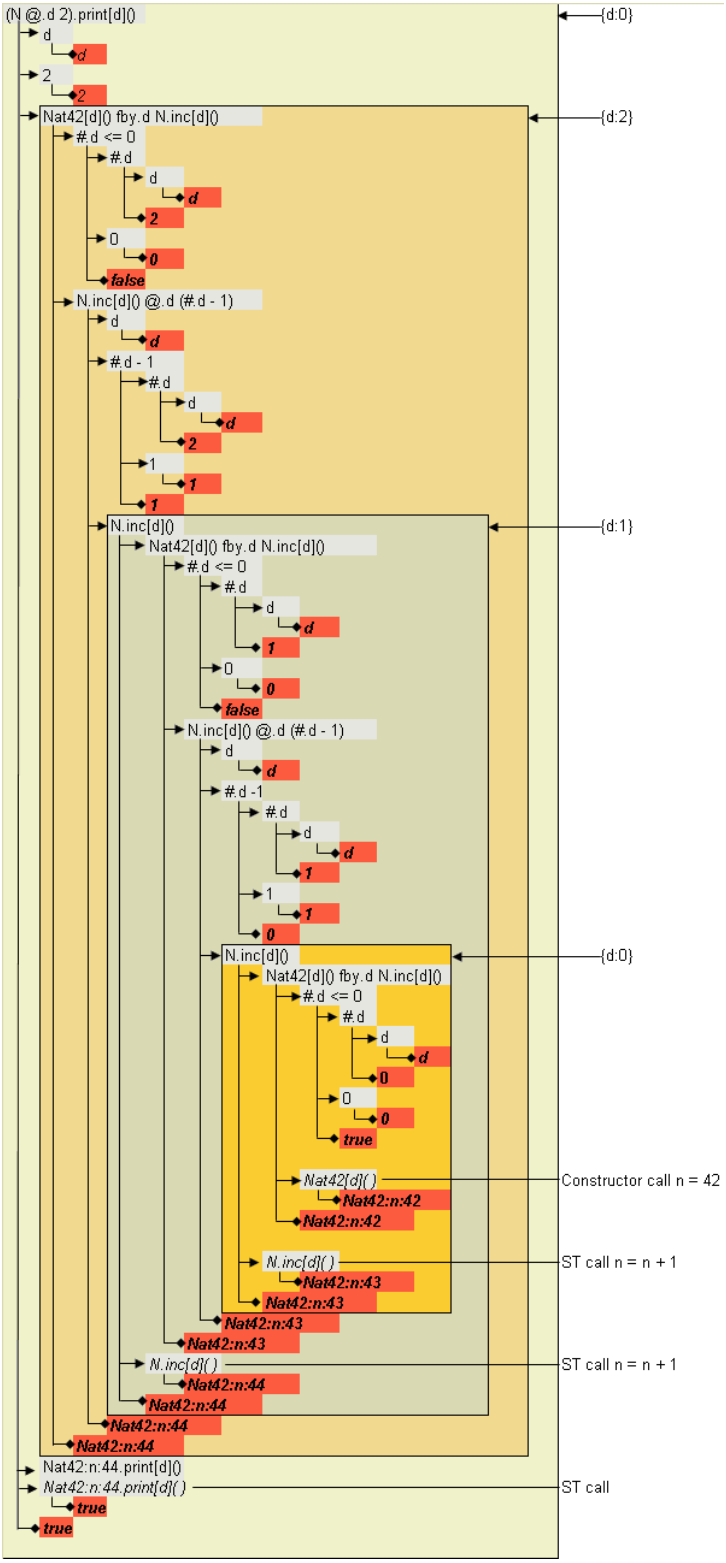

Figure 6: Eduction tree as a trace for the natural-numbers problem in Objective Lucid [5]

dataflow" [49] computing paradigm. This model of execution is built upon on the notion of generation, propagation, and consumption of *demands* and their resulting *values*. LUCID programs are declarative in nature where every identifier is defined as a HOIL expression using other identifiers and an underlying algebra. When value of any identifier is generated, it results in an initial demand. The eduction engine then generates requests for the identifiers that make up this expression, using the defining expression of this identifier. Operators are then applied to these identifiers in their embedded expressions. These demands in turn generate other demands, until some demands eventually evaluate to some values, which are then propagated back in the chain of demands, operators are applied to compute expression values, until eventually the value of the initial demand is computed and returned [24].

2.1.3 Lucid Dialects

We briefly review some of the core dialects that directly influence and contribute to the construction of MARFL in addition to the information provided in Section 2.1.1. More detailed information on the dialects and related publications can be found in [50].

2.1.3.1 LUCX

Wan's LUCX [2, 29] (which stands for *Lucid enriched with context*) provides a pivotal contribution to the syntax for MARFL. It is a fundamental extension of GIPL and the LUCID family as a whole that promotes the contexts as first-class values thereby creating a "true" generic LUCID language. An excerpt of its semantics is provided in Figure 4. A new collection of set operators (e.g., **`union`**, **`intersection`**, **`box`**, etc.) was defined by Wan [2, 29] on the multidimensional contexts, which then help with the parameters and sub-parameters for a particular algorithm in a MARFL program. LUCX's further specification, refinement, and implementation details were produced by Tong [3, 51] in 2007–2008 based on Wan's design [4, 24]. We will later elaborate on some of the formal definitions of context and context operators that were added

by Wan in her PhD thesis [29] in Section 3.1.3.2.

2.1.3.2 JLucid

JLucid [5, 52] was the first attempt on having intensional arrays and "free Java functions" in the GIPSY environment. The main computation was driven by Lucid language, where Java methods were peripheral and could be invoked from the Lucid segment, but not the other way around. This was one of the first instances of combination with an Object-Oriented language within GIPSY. The semantics of this approach was not completely defined, and, it was only a single-sided view (Lucid-to-Java) of the problem. JLucid eventually served as a precursor to Objective Lucid [33, 35, 53].

2.1.3.3 Objective Lucid

Objective Lucid [5, 54] was a natural extension to the JLucid language since it inherited all of the JLucid's features and introduced Java objects that can be used by Lucid. Objective Lucid expanded the notion of the Java object (a collection of members of different types) to the array (a collection of members of the same type) and first introduced the dot-notation in the syntax and operational semantics. Like in JLucid, Objective Lucid's focus was on the Lucid part being the "main" computation program and did not allow Java to call intensional functions or use intensional constructs from within a Java class. Objective Lucid was the first in GIPSY to introduce the more complete operational semantics of the hybrid OO intensional language [33, 35, 53].

2.1.3.4 JOOIP

Wu introduced JOOIP [53, 55] which greatly complements Objective Lucid by allowing Java to call the intensional language constructs closing the gap and making JOOIP a complete hybrid OO intensional programming language within the GIPSY environment. JOOIP's semantics further refines in a greater detail the operational semantics rules of Lucid and Objective Lucid in the attempt to make them

complete [33,35,53]. Especially, JAVA is a very popular and widely used language in today's application domains. By being a hybrid between JAVA and LUCID, JOOIP aimed to increase the visibility of Intensional Programming [56,57] and make it more mainstream.

2.1.3.5 FORENSIC LUCID

Mokhov introduced FORENSIC LUCID in his PhD Thesis [26], along with a proposed re-design of the GIPSY platform. FORENSIC LUCID can be used to represent the knowledge of forensic cases, including evidential description, witness accounts, assign credibility values to them, all in a common shape and form and then validate hypotheses claims against evidential data.

FORENSIC LUCID is a context-oriented language where a crime scene model comprises a state machine of evaluation and the forensic evidence and witness accounts comprise the context for its possible worlds. Like some of its predecessors, it inherits and is influenced by the productions from LUCX [2, 29], JLUCID and OBJECTIVE LUCID [5], GIPL and INDEXICAL LUCID [1]. The hierarchical contexts were first formally defined by Mokhov in his theoretical presentation of MARFL in [6] which were then incorporated into FORENSIC LUCID. We inherit and formalise these same hierarchical context productions and add our own set of context operators for implementing MARFL.

2.2 Modular A* Recognition Framework (MARF)

2.2.1 MARF Overview

Modular A* Recognition Framework (MARF), introduced in the year 2002 by Mokhov and other collaborators [18] , is an open-source project that provides pattern recognition APIs with sample implementation for (un)supervised machine learning and classification applications written in JAVA [22, 58, 59]. It serves as a testbed to verify common and novel algorithms for sample loading, pre-processing, feature

extraction, training, and classification stages of any pattern recognition experiment. MARF offers several applications that provide various configuration options inside a uniform environment that gives researchers a tool for practical comparison of algorithms. Over the years, a fair number of implementations have been added for each of the pipeline stages (cf. Figure 7, page 30) in MARF which allows to perform various pattern recognition tasks. MARF, its derivatives, and applications can now also be used beyond audio processing tasks due to the generality of the design and implementation in [48, 60, 61] and other works [7, 62].

One of its application, `FileTypeIndentApp`, was used to employ MARF's functionality for forensic analysis of file types [7]. Mokhov, based on the open-source MARF architecture, developed MARFCAT application which can be used as a static code analysis tool [63]. We discuss more about the MARFCAT application in Section 2.2.4. Some of the MARF's architectural design even influenced GIPSY [5] and MARF's utility modules are likewise in use by GIPSY.

2.2.2 MARF Architecture

MARF architecture consists of pipeline stages that communicate with each other to get the data needed in a chained manner. MARF's pipeline of algorithm implementations is illustrated in Figure 7, where the implemented algorithms are in white boxes, and the stubs or in-progress algorithms are in gray. The four basic stages of the pipeline will be discussed in more detail in Section 2.2.3

MARF's functionality can be tested via number of applications it comes with. They also serve as samples on how to use MARF's modules. One of the most prominent applications, that we will reproduce with MARFL for evaluation purposes, is `SpeakerIdentApp`—Text-Independent Speaker Identification (who, gender, accent, spoken language, etc.) [64]. Its derivative, `FileTypeIdentApp`, was used to employ MARF's capabilities for forensic analysis of file types [7] in [26].

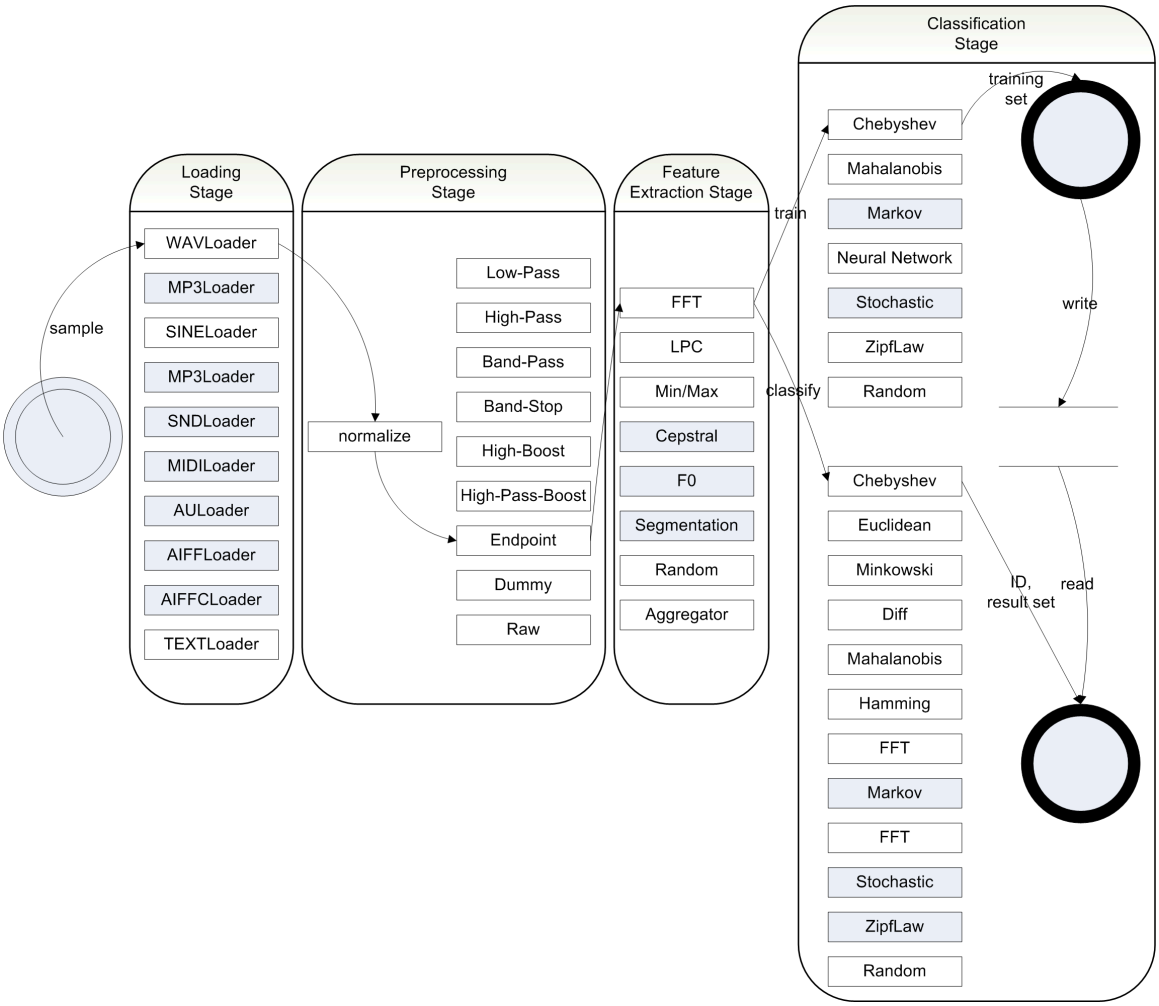

Figure 7: MARF's pattern-recognition pipeline [6]

2.2.3 Pattern Recognition Pipeline

The conceptual pattern recognition pipeline consists of four basic stages: loading, preprocessing, feature extraction, and training/classification [48, 65]. The design presented in Figure 7 depicts the core of the data flow and transformation between the stages in MARF [18, 58].

The basic steps in classical pattern recognition involve loading a sample, such as an audio recording, text, or image file, then preprocessing it with techniques like normalization and removing noise, followed by extracting the most important features, and then either training the system to learn new features for a specific task or using those features to classify the subject [7].

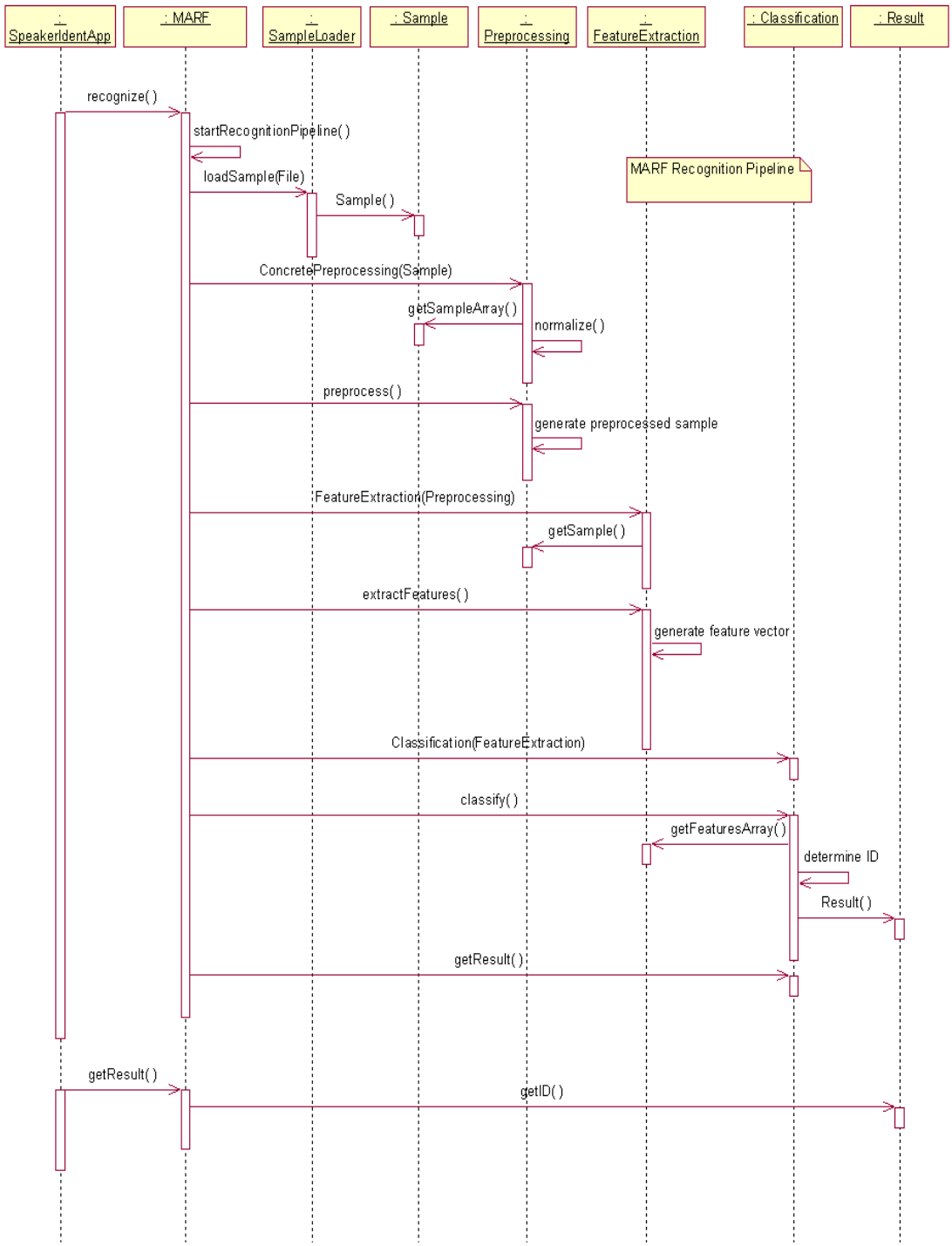

Figure 8: MARF's pattern-recognition pipeline sequence diagram [7]

The outcome of training is a set of feature vectors or clusters [22], known as training sets, which are saved for each learned subject. The outcome of classification is an instance of the `ResultSet` data structure, which is a sorted collection of IDs (`int`) and their corresponding outcome values (`double`); with the most likely outcome listed first. The most likely one is the ultimate outcome to be interpreted by the application. The steps involved in this process of classification in the main MARF module are demonstrated with the help of a sequence diagram in Figure 8 [7].

2.2.4 MARFCAT

In 2010, Mokhov demonstrated a tool to analyse static source and binary code in search for and investigation on program weaknesses and vulnerabilities during the SATE2010 workshop [63, 66, 67]. The MARF-based Code Analysis Tool, or MARFCAT [63] employed machine learning techniques along with signal processing and NLP alike for forensic investigation purposes. The core ideas and principles behind this tool were inherited from MARF's pipeline and testing methodology for various algorithms. MARFCAT machine-learns from the (Common Vulnerabilities and Exposures) CVE-based vulnerable as well as synthetic CWE-based cases to verify the fixed versions as well as non-CVE based cases from the projects written in various programming languages. MARFCAT's design from the beginning in 2010 was made independent of the language being analyzed, be it source code, bytecode, or binary.

In order to enhance the scalability of the approach, Mokhov converted the MARFCAT stand-alone application to a distributed one using an eductive model of computation (demand-driven) implemented in the General Intensional Programming System (GIPSY)'s multi-tier run-time system [9, 68–70], which can be executed distributively. To adapt the application to the GIPSY's multi-tier architecture, he created a problem-specific generator and worker tiers (PS-DGT and PS-DWT respectively) for the MARFCAT application. The generator(s) produce demands of what needs to be computed in the form of a file (source code file or a compiled binary) to be evaluated and deposit such demands into a store managed by the demand store tier (DST) as pending. Workers pickup pending demands from the store, and them

process then (all tiers run on multiple nodes) using a traditional MARFCAT instance. Once the result (a `Warning` instance) is computed, the PS-DWT deposit it back into the store with the status set to *computed*. The generator "harvests" all computed results (warnings) and produces the final report for a test cases. Since, this work is imperative to digital investigations MARF and MARFCAT both were designed to export evidence in the FORENSIC LUCID format [71].

2.3 The General Intensional Programming System

The General Intensional Programming System (GIPSY) [1,5,9,10,20,21,24,72–76], is an open-source platform implemented primarily in JAVA to investigate properties of the LUCID [30–32] family of intensional programming languages and beyond. Currently, GIPSY is being developed and maintained by the *GIPSY Research and Development Group* at Concordia University, Montreal, Canada. It follows a multi-tiered distributed architecture, which can evaluate LUCID programs by following a demand-driven distributed generator-worker approach. It is designed as a modular collection of frameworks where components related to the development (RIPE[1]), compilation (GIPC[2]), and of LUCID [32] programs are decoupled allowing easy extension, addition, and replacement of the components and subcomponents. The high-level general architecture of GIPSY is presented in Figure 9 as well as the high-level structure of the GIPC framework is in Figure 10 [4,33,34,48,65]. In this section we aim to present a general GIPSY overview followed by its key architectural design points to provide an in-depth background overview.

GIPSY [5,20,21,24,74,75] is a continued effort for the design and development of a flexible and adaptable multi-lingual programming language development framework aimed at the investigation on the LUCID family of intensional programming languages [1, 13, 14, 30, 31, 43, 50, 77, 78]. Using this platform, programs written in various dialects of LUCID can be compiled and executed in a variety of ways [8,10,24,

[1]Run-time Integrated Programming Environment, implemented in `gipsy.RIPE`
[2]General Intensional Programming Compiler, implemented in `gipsy.GIPC`

79]. GIPSY follows a modular framework approach that makes it easier to develop compiler components for other languages of intensional nature, and to execute them on a generally language-independent run-time system. As discussed in Section 2.1, LUCID is a functional "data-flow" language and, its programs can be executed in a distributed processing environment [8].

GIPSY's design includes a flexible compilers framework and a run-time system that allows for the processing of programs written in multiple dialects of LUCID, as well as the ability to mix those dialects with common imperative languages like Java, all within the same source code "program" or file. This feature sets GIPSY apart from being just a single LUCID dialect and instead positions it as a complete programming system for multiple languages.

As a result, by being multi-lingual, GIPSY's design includes the flexible compilers framework and a run-time system to allow processing of programs written in multiple dialects of LUCID as well as the ability to mix those dialects with common imperative languages, such as JAVA, all withtin the same source code "program". This feature sets GIPSY apart from being "just a LUCID dialect" into a complete programming system for multiple languages and the "meta" preprocessor language of various declarations to aid compilation [5, 79, 80].

GIPSY is the proposed platform for the compilation and distributed evaluation of the MARFL programs. GEE is the component where the distributed demand-driven evaluation takes place, subtasked to different evaluation engines implemeting the GEE framework. We rely on the GIPSY's compilers for the intensional languages like GIPL [1], LUCX [2], OBJECTIVE LUCID [5], and JOOIP [53]. We inherit the syntax and operational semantics of those languages implemented in GIPSY and draw ideas from them for the simple context specification of dimensions and tags, the navigational operators `@` and `#`, and the "dot-notation" for object properties and apply it to context spaces. The dialects that are referred to encompass a wide range of issues that MARFL takes advantage of.

Intensional programming [56, 57], in the context of the LUCID programming language, implies a declarative programming paradigm. The declarations are

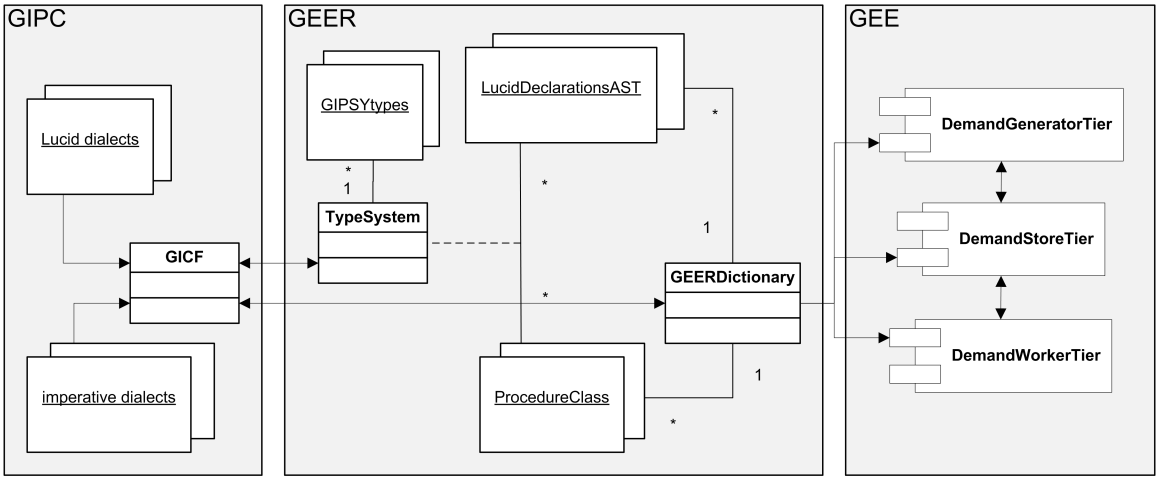

Figure 9: High-level structure of GIPSY's GEER flow overview [8]

evaluated in an inherent multi-dimensional context space [3, 10]. Initially GIPSY was a modular collection of frameworks for local execution. It subsequently evolved into a multi-tier architecture [9, 10] keeping in mind flexibility in order to cope with the fast evolution and diversity of the LUCID family of languages, thus necessitating a flexible compiler architecture, and a language-independent run-time system for the execution of LUCID programs. As a result, the GIPSY project's architecture [5, 24, 72, 75, 80] aims at providing such a flexible platform for the investigation on intensional and hybrid intensional-imperative programming [10].

The architecture of the General Intensional Programming Compiler (GIPC) is framework-based, allowing the modular development of compiler components (e.g., parser, semantic analyzer, and translator). It is based on the notion of the Generic Intensional Programming Language (GIPL) [1, 51], which is the core run-time language into which all other flavors of the LUCID (a family of intensional programming languages) language can be translated to [24]. The language-independence problem of the run-time system was solved by using the generic implementation which allowed a common representation for all compiled programs into the Generic Eduction Engine Resources (GEER). GEER is a dictionary of run-time resources compiled from a GIPL program, that had been previously generated from the original program using semantic translation rules that define how the original LUCID program is translated into the GIPL [10, 24]. A generic distributed run-time

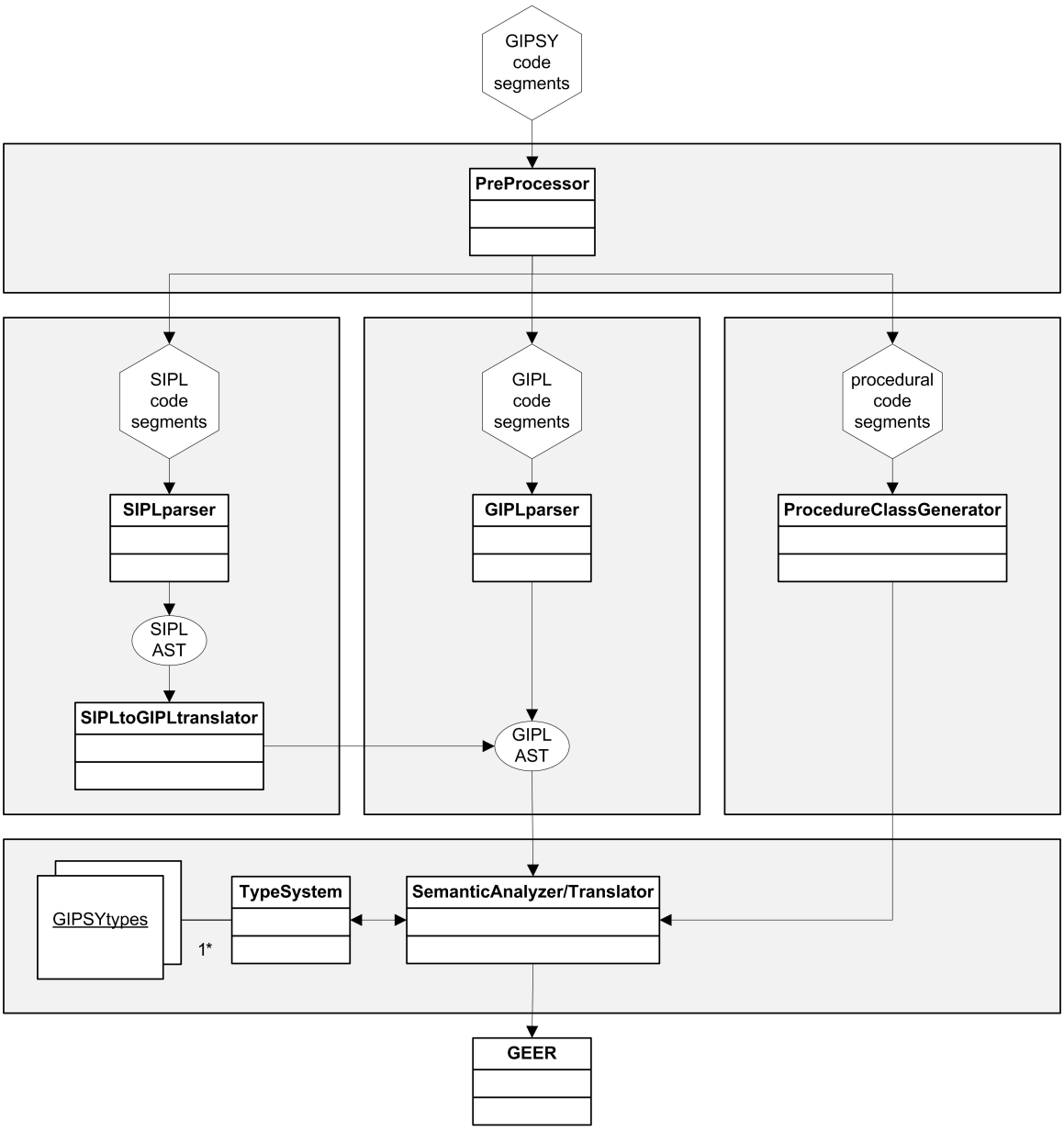

Figure 10: High-level structure of the GIPC framework [8]

system has been proposed in [9, 10].

Under the GIPC (see Figure 10) framework, GIPSY has a collection of compilers and the corresponding run-time environment under the eduction execution engine (GEE) among other things that communicate through the GEE Resources (GEER) (see the high-level architecture in Figure 9). These two are the major primary components for compilation and execution of intensional programs, which require amendments for the changes proposed in this work [4, 48, 65].

2.3.1 General Intensional Program Compiler (GIPC)

The more detailed architecture of GIPC is conceptually represented at the higher level in Figure 10. It has the type abstractions and implementations that are located in the `gipsy.lang` package and serve as a glue between the compiler (the GIPC—a General Intensional Program Compiler) and the run-time system (known as the GEE—a General Eduction Engine) to do the static and dynamic semantic analyses and evaluation respectively. The abstract syntax tree (AST) generated is stored in GEER which is followed by its own type checking at run time. Since both the GIPC and the GEE use the same type system to do their analysis, they consistently apply the semantics and rules of the type system with the only difference that the GEE, in addition to the type checks, does the actual evaluation [8, 24]. As seen in Figure 10, the `Preprocessor` [5,80] is invoked first by the GIPC on incoming GIPSY program's source code stream. Preliminary program analysis, processing, and splitting the source GIPSY program into "chunks", is handled by the `Preprocessor`. Each chunk of a source GIPSY program can potentially be written in a different language and identified by a *language tag*. In that sense, a GIPSY program is a hybrid program written in different language variants in one or more source file. The `Preprocessor` after some initial parsing produces the initial parse tree, then constructs a preliminary dictionary of symbols used throughout all parts of the program. The `Preprocessor` then splits the code segments of the GIPSY program into chunks preparing them to be fed to the respective concrete compilers for those chunks. The chunks are represented through the `CodeSegment` class, instances of which the `GIPC` collects [8, 24].

GIPSY Program Segments. There are four baseline types of segments defined in a GIPSY program [8]. These are:

- `#funcdecl`: It declares function prototypes written as imperative language functions defined later or externally from this program to be used by the intensional language part. The syntactical form of these prototypes is particular to GIPSY programs and need not resemble the actual function prototype declaration they describe in their particular programming language. They serve

as a basis for static and dynamic type assignment and checking within the GIPSY type system with regards to procedural functions called by other parts of the GIPSY program, e.g., the LUCID code segments [8].

- `#typedecl`: It lists all user-defined data types that can potentially be used by the intensional part, e.g., classes. These are the types that do not explicitly appear in the matching table describing the basic data types allowed in GIPSY programs [8].

- `#<IMPERATIVELANG>` declares that this is a code segment written in whatever IMPERATIVELANG may be, e.g., `#JAVA` for JAVA, `#CPP` for C++, `#FORTRAN` for FORTRAN, `#PERL` for PERL, and `#PYTHON` for PYTHON, etc. [8].

- `#<INTENSIONALLANG>` declares that what follows is a code segment written in whatever INTENSIONALLANG may be, for example `#GIPL`, `#LUCX`, `#JOOIP`, `#INDEXICALLUCID`, `#JLUCID`, `#OBJECTIVELUCID`, `#TENSORLUCID`, `#MARFL`, `#FORENSICLUCID` [39], and `#ONYX` [81], etc., as specified by the available GIPSY implementations and stubs. An example of a hybrid program is presented in Listing 2.1. The preamble of the program with the type and function declaration segments are the main source of type information that is used at compile time to annotate the nodes in the tree to help both static and semantic analyses [8].

2.3.2 GICF Overview

The General Imperative Compiler Framework (GICF) [82] is GIPSY's compiler framework that allows for a generalized way of inclusion of any imperative languages into intensional variants within the GIPSY environment. It essentially allows the syntactical co-existence of the intensional and imperative languages in one source file by providing a `Preprocessor` that splits the intensional and imperative code chunks that are then fed to their respective compilers. the results are then gathered and linked together to form a compiled hybrid program as an instance of GEER [53].

```
#typedecl
myclass;

#funcdecl
myclass foo(int,double);
float bar(int,int):"ftp://localhost/cool.class":baz;
int f1();

#JAVA
myclass foo(int a, double b) {
        return new myclass(new Integer((int)(b + a)));
}
class myclass {
        public myclass(Integer a) {
                System.out.println(a);
        }
}

#CPP
#include <iostream>
int f1(void) {
        cout << "hello";
        return 0;
}

#OBJECTIVELUCID
A + bar(B, C)
where
A = foo(B, C).intValue();
B = f1();
C = 2.0;
end;
```

Listing 2.1: Example of a hybrid GIPSY program

Since GIPSY targets to unite most intensional paradigms in one research system, it makes an effort to be as general as possible and as compatible as possible and pragmatic at the same time [48]. GIPSY's GIPC can be extended to support compiler module for C and FORTRAN functions as it does for JAVA. Towards that goal, GICF is made extensible so that later on the language support for C++, PERL, PYTHON, shell scripts, and so on can be relatively easily added.

2.3.3 General Eduction Engine (GEE)

The primary purpose of the GEE is to evaluate compiled LUCID programs following their operational semantics either locally or distributively using the lazy demand-driven model (i.e., eduction). To address run-time scalability concerns, GEE is the component where the distributed demand-driven evaluation takes place by relying on the Demand Migration System (DMS) [70, 83] and on the multi-tier architecture

overall [9, 10, 48, 65]. The distributed system [84, 85] design architecture adopted for the run-time system is a distributed multi-tier architecture, where each tier can have any number of instances [24]. The architecture bears resemblance with a peer-to-peer architecture [9, 10], where [24]:

- Demands are propagated without knowing where they will be processed or stored.
- Any tier or node can fail without the system to be fatally affected.
- Nodes and tiers can seamlessly be added or removed on the fly as computation is happening.
- Nodes and tiers can be affected at run-time to the execution of any GIPSY program, i.e., a specific node or tier could be computing demands for different programs.

Below, we explain the important components of GEE.

1. Generic Eduction Engine Resources.

   One of the pivotal concepts of the GIPSY's' solution is *language independence* of the run-time system. In order to achieve that, the design relies on an intermediate representation that is generated by the compiler: the Generic Eduction Engine Resources (GEER). The GIPC compiles a program into an instance of the GEER(s), including a dictionary of identifiers extracted from the program [5, 73, 80]. Since the compiler framework provides with the potential to allow additions of any dialect of the Lucid language to be added through automated compiler generation taking semantic translation rules in input [28], the compiler designers need to provide a parser and a set of rules to compile and link a GEER, often by translating a specific Lucid dialect to GIPL first [24]. Our work will handle these translations inside the Semantic Analyzer like its predecessor Lucx.

As the name suggests, the GEER structure is generic, in the sense that the data structure and semantics of the GEER are independent of the source language. The engine was designed to be "source-language independent", an important feature made possible by the presence of the Generic Intensional Programming Language (GIPL) as a generic language in the LUCID family of languages [9, 10, 24]. The GEER contains, for all LUCID identifiers in a given program, typing information, rank (i.e., dimensionality information), as well as an abstract syntax tree (AST) representation of the declarative definition of each identifier [9, 10, 24]. It is this latter tree that is traversed later on by the demand generator tier in order to proceed with demand generation.

2. GIPSY Tier.

   The architecture adopted for the most recent evolution of the GIPSY is a multi-tier architecture where the execution of GIPSY programs is divided in three different tasks assigned to separate tiers [9, 10].

   Each GIPSY tier is a separate process that communicates with other tiers using *demands*, i.e., the *GIPSY Multi-Tier Architecture* operational mode is fully *demand-driven*. The demands are generated by the tiers and migrated to other tiers using the *Demand Store Tier*. We refer to a *tier* as an abstract and generic entity that represents a computational unit independent of other tiers and that collaborates with other tiers to achieve program execution as a group (GIPSY network) [9, 10, 24].

   In Figure 11 is the context use case diagram describing user interaction with the nodes and tiers to get them started to form a GIPSY software network. The user interaction is done either via command line or GUI support in the RIPE package interfacing the GMT [86].

3. GIPSY Node.

   Abstractly, a *GIPSY Node* is a computer (physical or virtual) that has been registered for the hosting of one or more *GIPSY Tiers*. GIPSY Nodes are

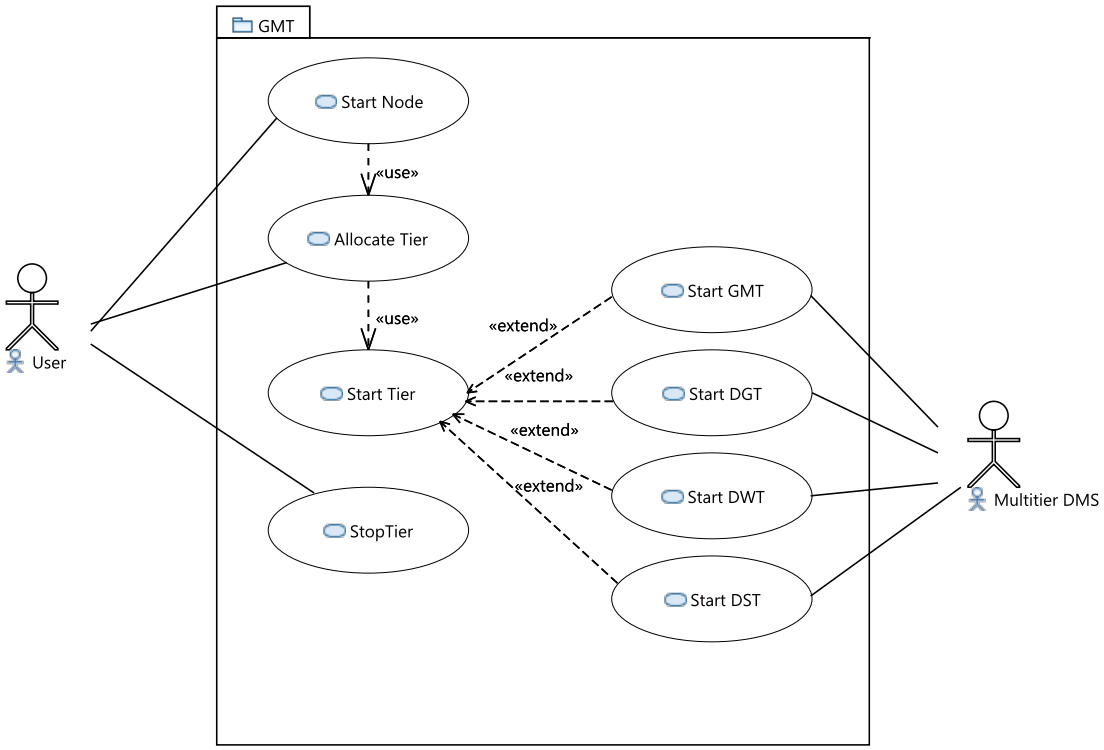

Figure 11: GMT use-case diagram

registered through a *GIPSY Manager Tier* (GMT) instance. Technically, a GIPSY Node is a controller that wraps GIPSY Tier instances, and that is remotely reporting and being controlled by a GIPSY Manager Tier [9, 10]. As shown in Figure 12, a GIPSY Node hosts one tier controller for each kind of Tier. The *Tier Controller* acts as a factory that will, upon necessity, create *instances* of this Tier, which provide the concrete operational features of the Tier in question. This model permits scalability of computation by allowing the creation of new Tier instances as existing tier instances get overloaded or lost [9, 10, 24].

4. GIPSY Instance.

   A *GIPSY Instance* is a set of interconnected GIPSY Tiers deployed on GIPSY Nodes executing GIPSY programs by sharing their respective GEER instances. A GIPSY Instance can be executed across different GIPSY Nodes, and the same GIPSY Node may host GIPSY Tiers that are members of separate GIPSY

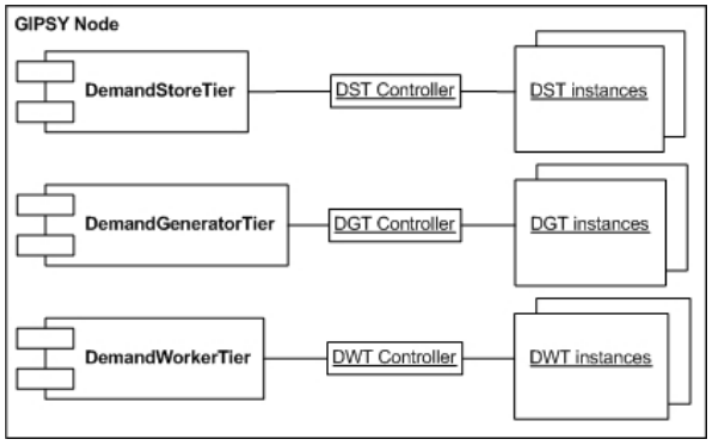

Figure 12: Design of the GIPSY node[9, 10]

Instances [9, 10, 24]. In Figure 12 is Paquet's rending of the described design [9]. GIPSY Instances form the *GIPSY software networks*, similar in a way to software-defined networks [87].

5. Demand Generator Tier.

   The *Demand Generator Tier* (DGT) generates demands according to the program declarations and definitions stored in one of the instances of GEER that it hosts. The demands generated by the Demand Generator Tier instance can be further processed by other Demand Generator Tier instances (in the case of *intensional demands*) or *Demand Worker Tier* instances (in the case of *procedural demands*), the demands being migrated across tier instances through a *Demand Store Tier* instance. Each DGT instance hosts a set of GEER instances that corresponds to the LUCID programs it can process demands for. A demand-driven mechanism allows the Demand Generator Tier to issue *system demands* requesting for additional GEER instances to be added to its *GEER Pool* (a local collection of cached GEERs it has learned), thus enabling DST instances to process demands for additional programs in execution on the GIPSY networks they belong to [9, 10, 24].

6. Demand Store Tier.

   The *Demand Store Tier* (DST) acts as a tier middleware in order to migrate demands between tiers. In addition to the migration of the demands and values across different tiers, the Demand Store Tiers provide persistent storage of demands and their resulting values, thus achieving better processing performances by not having to re-compute the value of every demand every time it is re-generated after having been processed. From this latter perspective, it is equivalent to the historical notion of an *intensional value warehouse* [20, 88] in the eductive model of computation (Section 2.1.2, page 23). A centralized communication point or warehouse is likely to become an execution bottleneck for large long-running computations. In order to avoid that, the Demand Store Tier is designed to incorporate a peer-to-peer architecture as needed and a mechanism to connect all Demand Store Tier instances in a given GIPSY network instance. This allows any demand or its resulting value to be stored on any available DST instance, but yet allows abstract querying for a specific demand value on any of the DST instances.

7. Demand Worker Tier.

   The *Demand Worker Tier* (DWT) processes primarily *procedural demands*, i.e., demands for the execution of functions or methods defined in a procedural language, which are only present in the case where hybrid intensional programs are being executed. The DGT and DWT duo is an evolution of the generator-worker architecture adopted in GLU [36, 37, 89]. It is through the operation of the DWT that the increased granularity of computation is achieved. Similarly to the DGT, each DWT instance hosts a set of compiled resident procedures (sequential thread procedure classes) that corresponds to the procedural demands it can process pooled locally. A demand-driven mechanism allows the Demand Worker Tier to issue *system demands* requesting for additional GEERs to be added to its GEER pool, thus achieving increased processing knowledge capacity over time, eductively [9, 10, 24].

8. General Manager Tier.

A *General Manager Tier* (GMT) is a component that enables the registration of GIPSY Nodes and Tiers, and to allocate them to the GIPSY network instances that it manages. The General Manager Tier interacts with the allocated tiers in order to determine if new tiers and/or nodes are necessary to be created, and issue system demands to GIPSY Nodes to spawn new tier instances as needed. As with DSTs, multiple GMTs are designed to be peer-to-peer components, i.e., users can register a node through any GMT, which will then inform all the others of the presence of the new node, which will then be available for hosting new GIPSY Tiers at the request of any of the GMT currently running. The GMT uses *system demands* to communicate with Nodes and Tiers [9, 10, 24], in a way similar to SNMP get and set requests [90–92]. In Figure 13 is a more

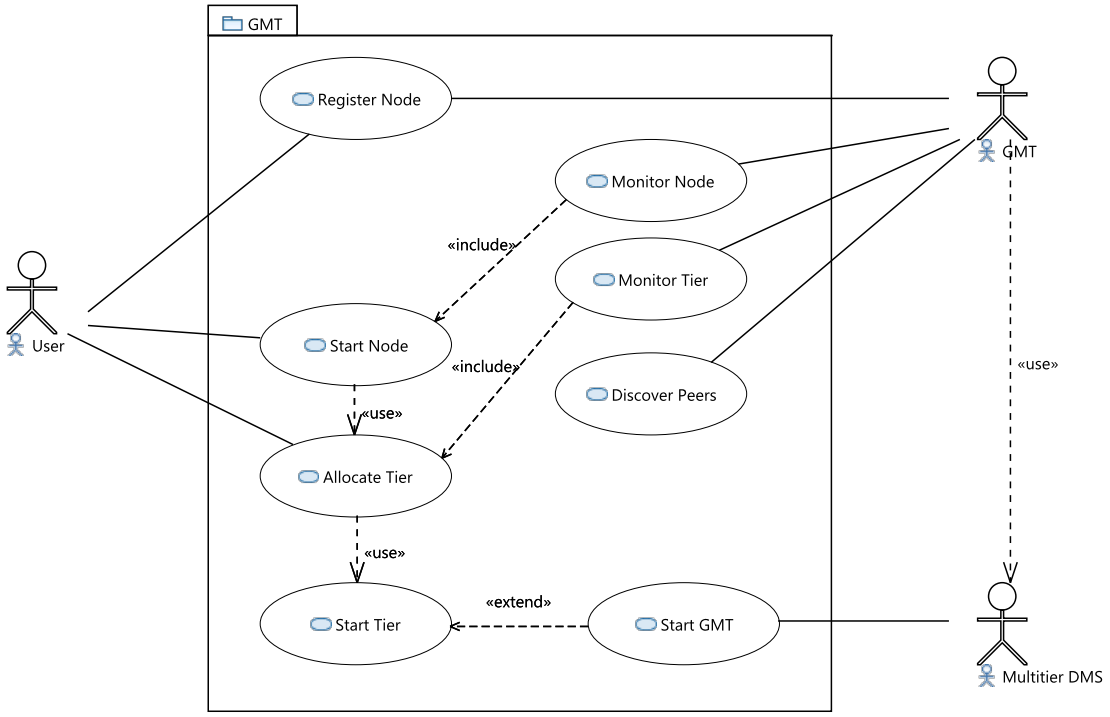

Figure 13: Detailed GMT context use case diagram

detailed GMT-oriented use case diagram of Figure 11 (where other tiers are implied with the focus on the GMT). Once started, the GMT acts as a service using the DMS, which is designed to play an active role in managing the GIPSY software network instance along with the interacting user.

2.3.4 Scalability

The GIPSY multi-tier architecture after extensive design revision, implementation and refactoring [10, 68, 69, 93] was put to scalability tests by Ji [69]. The need for such is also emphasized in the work by Fourtounis *et al.* [94] where eduction and massive parallelism of intensional evaluation are discussed in a formal model and PoC implementation for analysis.

Scalability is an important attribute of any computing system as it represents the ability to achieve long-term success when the system is facing growing demands load. The multi-tier architecture was adopted for the GIPSY runtime system for research goals such as scalability [9, 10]; therefore, upon implementation of the PoC Jini and JMS DMS, the scalability of the GIPSY runtime system was assessed and validated by Ji in his mater's thesis [69, 95].

Researchers use *scalability* to denote the capability for the long-term success of a system in different aspects, such as the ability of a system to hold increasing amount of data, to handle increasing workload gracefully, and/or to be enlarged easily [69, 95, 96]. Ji did testing which showed that Jini DMS was better or worse over the JMS circumstances in terms of scaling out of GIPSY Nodes onto more physical computers and their available CPUs for the tiers, then the amount of memory used on the nodes before DSTs run out of memory, amount of demands they could handle, network delays and turnaround time, and other characteristics [69].

The scalability aspect is very important to this work as there is always a possibility of a need to do heavy-weight pattern recognition tasks and process vast amounts of different types of data efficiently, robustly, and effectively.

2.4 Summary

To summarize this chapter, we first established that expressions written in all Lucid dialects correspond to higher-order intensional logic (HOIL) expressions with dialect-specific instantiations. The contextual expression can be passed as parameters and returned as results of a function and constitute the multi-dimensional constraint

on the LUCID expression being evaluated. Followed by, an explanation of MARF architecture and pipeline and how it is a pivotal contributor to this research work. Then we end our discussion by giving the necessary background on the General Intensional Programming System (GIPSY) and its core components like GIPC and GEE. We explain in detail the GIPC and GEE frameworks with the goal of flexibility (data types, languages, replaceable components) and scalability (multi-tier DMS) in mind, important for the MARFL project presented in this thesis.

Chapter 3

MARFL Specifications and Design

This chapter discusses the concrete syntax, operational semantics, as well as provide definitions of operators introduced for MARFL. The inherited operators from other languages such as Lucx along with their definitions will also be reviewed.

MARFL is a computational context specification language that allows scripting of MARF-based applications as context-aware, where the notion of context represents fine-grained configuration details of a given MARF instance, which allows us to execute it onto the GIPSY runtime system. It can also extend to other machine learning backends such as TensorFlow. The language itself is general enough to specify any parameters or sub-parameters for a context-oriented system model. MARFL is based on Lucid [13, 14, 30] and its various dialects that allow natural expression of various phenomena, inherently parallel, and most importantly, context-aware, i.e., the notion of context is specified as a first-class value [2, 3, 51].

As previously mentioned (Chapter 1), the first formal approach to provide syntax and operational semantics for MARFL was given by Mokhov in [6]. However, their approach was only limited to presenting a conceptual and theoretical definitions for the language. The aim of this research work is to extend, improve and implement the syntax and operational semantics provided in [6] inside the GIPSY environment.

This chapter presents the summary of the design, and formalization of the syntax and semantics of our proposed language. That includes the design and requirements considerations of the MARFL language (Section 3.1), including higher-order contexts

(Section 3.1.2), syntax (Section 3.2) and semantics (Section 3.3).

3.1 MARFL Language Requirements and Design Considerations

This section presents concepts and considerations in the design of the MARFL language. The end goal is to define our MARFL language where its syntactic constructs and expressions concisely model properties of algorithms present for a given MARF instance. The implementing system, GIPSY, provides the ability to perform evaluations in a distributed manner based on several configurations that are modelled in an intensional manner.

3.1.1 Core Language Properties, Features, and Requirements

We define and use MARFL to specify context-aware configurations for MARF representing the parameters for available algorithms for each of the stages in a MARF pipeline described in Section 2.2.3. One of the goals is to be able to "script" the machine learning tasks using available machine learning backends as expressions and run that "script" through an evaluation system that provides the results of specified pattern recognition task [6].

MARFL builds its design by aggregating the features and semantics of multiple Lucid dialects mentioned in Chapter 2 along with its own extensions. Lucx's context calculus with contexts as first-class values [2] and operators on simple contexts and context sets are adopted to manipulate hierarchical contexts in MARFL. Additionally, MARFL inherits the properties of Forensic Lucid, Objective Lucid and JOOIP and their comprising dialects for the arrays and structural representation of parameters for providing configurations. Hierarchical contexts in MARFL follow the examples given in [6] and by overloading both `@` and `#` (defined further in Section 3.2, page 54) to accept different types as their left and right arguments.

One of the basic requirements while designing MARFL is that the final language should be a conservative extension of the previous LUCID dialects (valid programs in those languages are valid programs in MARFL). This is helpful when complying with the compiler (GIPC) and the run-time subsystem (GEE) frameworks within the implementing system, the General Intensional Programming System (GIPSY) [21, 75]. The partial translation rules (Section 3.2.1.2) are provided when implementing the language compiler within GIPSY, such that the run-time environment (General Eduction Engine, or GEE) can execute it with minimal changes to GEE's implementation

3.1.2 Higher Order Context

Higher-order context (HOC) hierarchy was introduced by Mokhov as a theoretical concept in [6] which was later adopted in his PhD thesis [26] titled *Intensional Cyberforensics* where HOC was used to model evidential statements for forensic specification evaluations. As mentioned briefly in Chapter 1, LUCID dialects rely on higher-order intensional logic (HOIL) to provide context-oriented multidimensional reasoning of intensional expressions. HOIL combines functional programming with various intensional logics to allow explicit context expressions that can be evaluated as first-class values which can be passed as parameters to functions and return as results with set of operators defined on contexts [24, 25]. Mokhov's HOC interpretation for MARFL in [6] allows evaluation at arbitrary nesting of the configuration context with some defaults in the intermediate and leaf context nodes [24].

3.1.3 Formal Syntax and Semantics Definitions

The MARFL language includes the necessary syntax and semantics constructs as detailed in the following sections. We start by defining the common terminology used subsequently in the chapter throughout the formal definition of MARFL syntax and semantics and their description.

3.1.3.1 Structural Operational Semantics

We use structural operational semantics to describe the semantics of MARFL. Paquet [1] in 1999 has given a structural operational semantics of GIPL in 1999. The subsequently produced work on Lucx [2,3], JOOIP [55], other intensional languages used this semantics style consistently ever since. The basic operational semantics description has rules in the form of $\frac{Premises}{Conclusions}$ with *Premises* defined as possible computations, which, if take place, *Conclusions* will also take place [2,97].

Following the steps and identifiers presented in Section 2.1.1.3, [1,2] we thus define the following:

Table 2: MARFL identifier types in $\mathcal{D}$

type	form
dimension	$(\texttt{dim})$
constant	$(\texttt{const}, c)$
operator	$(\texttt{op}, f)$
variable	$(\texttt{var}, E)$
context operator	$(\texttt{cop}, f)$
context set operator	$(\texttt{sop}, f)$
MARFL operator	$(\texttt{mop}, f)$

Definition environment: The *definition environment* $\mathcal{D}$ stores the definitions of all of the identifiers that appear in a MARFL program. As in Equation 19, it is a partial function

$$\mathcal{D} : \textbf{Id} \rightarrow \textbf{IdEntry} \tag{32}$$

where **Id** is the set of all possible identifiers and **IdEntry**, summarized in Table 2, has possible kinds of values for each identified kind. (This table is an extended version of Table 1 with additional entries we define further.) Per Equation 17, $\mathcal{D} \vdash E : v$ means the expression E evaluates to the value v under $\mathcal{D}$ [1].

Evaluation context: The current evaluation context $\mathcal{P}$ (sometimes also referred to as a *point* in the context space) is an additional constraint put on evaluation, so per Equation 18, $\mathcal{D}, \mathcal{P} \vdash E : v$ specifies that given $\mathcal{D}$, and in $\mathcal{P}$, expression E evaluates to v [1].

Identifier kinds: The following identifier kinds can be found in $\mathcal{D}$:

1. *Dimensions* define simple coordinate pairs, which one can query and navigate with the `#` and `@` operators. Their **IdEntry** is simply (`dim`) [1].
2. *Constants* (**IdEntry** = (`const`, c), where c is the value of the constant) are external entities that provide a single value independent of the context of evaluation. Examples are integers, strings, and Boolean values, etc. [1].
3. *Data operators* are external entities that provide memoryless functions. Examples are the arithmetic and Boolean functions. The constants and data operators are said to define the *basic algebra* of the language. Their **IdEntry** is (`op`, f), where f is the function itself [1].
4. *Variables* carry the multidimensional streams. Their **IdEntry** is (`var`, E), where E is the MARFL expression defining the variable. Variable names are unique [1]. Finite variable streams can be bound by the special beginning-of-data (**bod**) and end-of-data (**eod**) markers (**eod** was introduced in [98]).
5. *Context operators* (`cop`, f) is a LUCX-style simple context operator type [2, 3]. These operators help us to manipulate context components in set-like operations, such as union, intersection, difference, and the like. Both simple context and context set operators are defined further.
6. *Context set operators* (`sop`, f) is a LUCX-style context set operator type [2, 3]. Both simple context and context set operators are defined further.
7. *ML subsystem* is a MARFL dimension type which provides the machine learning sub-system chosen by the user for implementing the specified pattern recognition pipeline. This dimension can vary over tags such as MARF, TensorFlow, PyTorch, etc.
8. *Sample Loader* is a MARFL dimension type which encodes the type of data the pattern recognition pipeline is going to use. This dimension can vary over tags such as WAV, MP3, TEXT, etc.

9. *Preprocessing* is a MARFL dimension type which specifies the preprocessing algorithm for the loaded data type. It can vary over available algorithms in the machine learning subsystem selected.

10. *Feature Extraction* is a MARFL dimension type which can specify the feature extraction algorithm to be used in the pipeline. This dimension can vary over tags such as FFT, Random Feature Extraction, etc. Essentially over all the available algorithms in the selected subsystem.

11. *Classification* is a MARFL dimension type which specifies the classification algorithm to be used for the pipeline. It can vary over available algorithms in the machine learning subsystem selected.

    MARFL *operators* $(\texttt{mop}, f)$ is a MARFL context operated defined further in Section 3.2.2

3.1.3.2 Context Types.

A general definition of context is [2, 3]:

Definition 1. *Context: A context c is a finite subset of the relation: $c \subset \{(d, x) | d \in DIM \wedge x \in T\}$, where DIM is the set of all possible dimensions, and T is the set of all possible tags [2, 3].*

Definition 2. *Tag: Tags are the indices to mark positions in dimensions to identify a context. [2, 3].*

Definition 3. *Simple context: A simple context is a collection of $\langle dimension : tag \rangle$ pairs, where there are no two such pairs having the same dimension component. Conceptually, a simple context represents a point in the context space, i.e., this is the traditional* GIPL *context $\mathcal{P}$. A simple context having only one $\langle dimension : tag \rangle$ pair is called a micro context. It is the building block for all the context types [2, 3].*

Syntactically, a simple context is $[E : E, ..., E : E]$ [2, 3], i.e., $[E_{d_1} : E_{t_1}, ..., E_{d_n} : E_{t_n}]$ where E_{d_i} evaluate to dimensions and E_{t_i} evaluate to tags.

Definition 4. *Context set: A context set (also known as "non-simple context") is a set of simple contexts. Context sets represent regions of the context space, which can be seen as a set of points, considering that the context space is discrete. Formally speaking, context set is a set of* $\langle dimension : tag \rangle$ *mappings that are not defined by a function [2, 3].*

Syntactically, context set is $\{E, ..., E\}$, where $E \rightarrow [E : E, ..., E : E]$.

Following the above fundamental definitions, Section 3.2 explains the complete syntax specification and Section 3.3 explains the operational semantics specification for MARFL.

3.2 Concrete MARFL Syntax

The concrete syntax of the MARFL language is presented in Figure 14. As stated before, it is influenced by the productions from Lucx [2, 29], JLucid and Objective Lucid [5], GIPL and the hierarchical contexts given in [6] which were then implemented in Forensic Lucid [26]. Presenting this concrete syntax partially fulfils our Objective O1.

In Figure 14, we first start by providing common top-level syntax expressions E of the language from identifiers, operators, arrays, the **`where`** clause, dot-notation, and so on. Then we provide the Q **`where`** productions containing various types of declarations, including the common GIPL dimensions and the new constructs for MARF built-in dimension types such as *ml_subsystem, sampleLoader, etc.* In Figure 15 we elaborate on the syntactical details on the hierarchical MARFL context specification of the *machine learning id* (MLid), *sample loader id* (SLid), *preprocessing id* (PRid), *feature extraction id* (FEid), and *classification id* (CLid). The various parameters involved in MARF's contexts require a hierarchy of contexts, subcontexts, and sub-subcontexts, with each level providing greater detail in the configuration settings. This contextuality results in a specific configuration instance with the required settings. In cases where some dimensions are missing in the MARFL specification, default values from the relevant module are used to fill them in. The

modules themselves represent higher-order dimensions that can be added or removed as the system evolves. However, user-defined variables, identifiers, and functions are not supported in this context, and have therefore been excluded from the syntax and semantics. The dimension identifiers currently used are constants derived from MARF's available modules and are considered reserved words in MARFL.

In Figure 16 is a syntax specification of different types of operators, such as arithmetic, logic, and intensional operators, in their unary and binary forms as well as context operators(cxtop) for MARF specifications.

$$
\begin{array}{lcll}
E & ::= & id & (33)\\
 & | & E(E, ..., E) & (34)\\
 & | & E[E, ..., E](E, ..., E) & (35)\\
 & | & \textbf{if } E \textbf{ then } E \textbf{ else } E \textbf{ fi} & (36)\\
 & | & \textbf{\#} \ E & (37)\\
 & | & E \ \textbf{@} \ E & (38)\\
 & | & E \ bin\text{-}op \ E & (39)\\
 & | & un\text{-}op \ E & (40)\\
 & | & E \ i\text{-}bin\text{-}op \ E & (41)\\
 & | & i\text{-}un\text{-}op \ E & (42)\\
 & | & E \textbf{ where } Q \textbf{ end}; & (43)\\
 & | & [E : E, ..., E : E] & (44)\\
 & | & cxtop \ E & (45)\\
 & | & [E, ..., E] & (46)\\
Q & ::= & \textbf{dimension } id, ..., id; & (47)\\
 & | & id = E; & (48)\\
 & | & E.id = E; & (49)\\
 & | & ml\_subsystem \ id = MLid; & (50)\\
 & | & sampleLoader \ id = SLid \ E; & (51)\\
 & | & preprocessing \ id = PRid \ E; & (52)\\
 & | & featureExtraction \ id = FEid \ E; & (53)\\
 & | & classification \ id = CLid \ E; & (54)\\
 & | & QQ & (55)
\end{array}
$$

Figure 14: Concrete MARFL syntax

MLid	::=	*MARF*	(56)
	|	*TensorFlow*	(57)
SLid	::=	*WAV*	(58)
	|	*MP3*	(59)
	|	*TEXT*	(60)
	|	*SINE*	(61)
	|	*AU*	(62)
	|	*MIDI*	(63)
PRid	::=	*FFT-LOW-PASS-FILTER*	(64)
	|	*RAW-PREPROCECESSING*	(65)
FEid	::=	*FFT*	(66)
	|	*RANDOM-FEATURE-EXTRACTION*	(67)
CLid	::=	*CHEBYSHEV-DISTANCE*	(68)
	|	*RANDOM-CLASSIFICATION*	(69)

Figure 15: Concrete MARFL syntax ($MLid, SLid, PRid, FEid, CLid$)

bin-op	::=	*arith-op* | *logical-op* | *bitwise-op*	(70)
un-op	::=	`+` | `-`	(71)
arith-op	::=	`+` | `-` | `*` | `/` | `%` | `^`	(72)
logical-op	::=	`<` | `>` | `>=` | `<=` | `==` | `in` | `&&` | `||` | `!`	(73)
bitwise-op	::=	`|` | `&` | `~` | `!|` | `!&`	(74)
i-bin-op	::=	`@` | *i-bin-op-forw* | *i-logic-bitwise-op*	(75)
i-bin-op-forw	::=	`fby` | `upon` | `asa` | `wvr`	(76)
i-logic-bitwise-op	::=	`and` | `or` | `xor`	(77)
i-un-op	::=	`#` | *i-bin-un-forw*	(78)
i-bin-un-forw	::=	`first` | `next` | `iseod`	(79)
cxtop	::=	*train* | *classify*	(80)

Figure 16: Concrete MARFL syntax (operators)

3.2.1 Core Operators

We define the basic set of the classic intensional operators [1] that are present to familiarize the reader what they do.

3.2.1.1 Definitions of Core Operators

The operators are defined below to give a more complete picture. The classical operators **first**, **next**, **fby**, **wvr**, **upon**, and **asa** were previously defined in [1] and earlier works (see [56, 57] and papers and references therein).

Definition 5. *Let X be a stream of natural numbers. Let Y be another stream of Boolean values;* true *is cast to 1 and* false *to 0 when used together with X in one stream.*

$$\begin{aligned} X &= (x_0, x_1, \ldots, x_i, \ldots) \\ &= (0, 1, \ldots, i, \ldots) \\ Y &= (y_0, y_1, \ldots, y_i, \ldots) \\ &= (true, false, \ldots, true, \ldots) \end{aligned}$$

Definition 6. ***first****: a stream of the first element of the argument stream.*

$$\textbf{first}\ X = (x_0, x_0, \ldots, x_0, \ldots)$$

Definition 7. ***next****: a stream of elements of the argument stream after the first.*

$$\textbf{next}\ X = (x_1, x_2, \ldots, x_{i+1}, \ldots)$$

Definition 8. ***fby****: the first element of the first argument stream followed by all of the second argument stream.*

$$X\ \textbf{fby}\ Y = (x_0, y_0, y_1, \ldots, y_{i-1}, \ldots)$$

Definition 9. ***and****: a logical AND stream of truth values of its arguments.*

$$X\ \textbf{and}\ Y = (x_0\ \textbf{\&\&}\ y_0, x_1\ \textbf{\&\&}\ y_1, x_2\ \textbf{\&\&}\ y_2, \ldots, x_{i+1}\ \textbf{\&\&}\ y_{i+1}, \ldots)$$

Definition 10. ***or****: a logical OR stream of truth values of its arguments.*

$$X \ \mathbf{or}\ Y = (x_0 \ \mathbf{||}\ y_0, x_1 \ \mathbf{||}\ y_1, x_2 \ \mathbf{||}\ y_2, \ldots, x_{i+1} \ \mathbf{||}\ y_{i+1}, \ldots)$$

Definition 11. ***xor****: a logical XOR stream of truth values of its arguments.*

$$X \ \mathbf{xor}\ Y = (x_0 \oplus y_0, x_1 \oplus y_1, x_2 \oplus y_2, \ldots, x_{i+1} \oplus y_{i+1}, \ldots)$$

Definition 12. ***wvr*** *(stands for* whenever*): **wvr** chooses from its left-hand-side operand only values in the current dimension where the right-hand-side evaluates to* true*. This is a classical* Indexical Lucid *operator.*

$$\begin{aligned} X \ \mathbf{wvr}\ Y = & \\ & \mathbf{if}\ \mathbf{first}\ Y \neq 0 \\ & \mathbf{then}\ X\ \mathbf{fby}\ (\mathbf{next}\ X\ \mathbf{wvr}\ \mathbf{next}\ Y) \\ & \mathbf{else}\ (\mathbf{next}\ X\ \mathbf{wvr}\ \mathbf{next}\ Y) \end{aligned}$$

Definition 13. ***asa*** *(stands for* as soon as*): **asa** returns the value of its left-hand-side as a first point in that stream as soon as the right-hand-side evaluates to* true*. This is a classical* Indexical Lucid *operator.*

$$X \ \mathbf{asa}\ Y = \mathbf{first}\ (X \ \mathbf{wvr}\ Y)$$

Definition 14. ***upon*** *(stands for* advances upon*): unlike **asa**, **upon** switches context of its left-hand-side operand if the right-hand side is* true*. This is a classical* Indexical Lucid *operator.*

$$X \textbf{ upon } Y = X \textbf{ fby } ($$
$$\textbf{if } \textbf{first } Y \neq 0$$
$$\textbf{then } (\textbf{next } X \textbf{ upon next } Y)$$
$$\textbf{else } (X \textbf{ upon next } Y))$$

Definition 15. *"." (dot): is a scope membership navigation operator.*

The "." operator is employed in multiple uses, so we include it into our syntax and semantics.

- From indexical expressions [1, 99] with the operators defined in the preceding and following sections it facilitates operations on parts of the evaluation context to denote the specific dimensions of interest to work with.
- Additional use in MARFL (inspired from FORENSIC LUCID) and introduced in [6], is the contextual depth navigation, similar in a way to the OO membership navigation.

Definition 16. *#: queries the current context of evaluation.*

Traditionally[1] # is defined as:

$$\# = 0 \text{ fby } (\# + 1)$$

for N tag set, that were infinite and ordered, essentially being akin to array indices into the current dimension X, and $\#X$ returned the current implicit index within X.

Subsequently,

$$\#.E = 0 \ \textbf{fby}.E(\# + 1)$$
$$\#C = C$$
$$\#S = S$$

$\#.E$ was introduced to allow querying any part of the current multidimensional context where E evaluates to a dimension, a part of the context. C and S are Lucx simple contexts and context sets.

Definition 17. `@` *: switches the current context of evaluation.*

Traditionally [1],a `@` is defined as:

$$X \texttt{ @ } Y = \texttt{if } Y = 0 \texttt{ then first } X \texttt{ else (next } X) \texttt{ @ } (Y - 1)$$

for N tag set in a single dimension. Subsequently,

$$X \texttt{ @ } .dE$$
$$X \texttt{ @ } C$$
$$X \texttt{ @ } S$$

are also provided. X `@` $.dE$ is the indexical way to work with parts of the context [1]; C and S are the simple contexts and context sets [2].

`@` is likewise adapted to work inside MARFL. For example, WAV `@` [**channels** : 1] - switches WAV loader's configuration dimensions **channels** to mono, essentially returning a loader with the new configuration setting.

3.2.1.2 Definition of Core Operators by Means of @ and

This step follows the same tradition as the most of the SIPLs in GIPSY (Section 2.1.3), which are translated into GIPL. The existing rules are applied here for the majority of our operators to benefit from the existing interpretation available in GEE (Section 2.3.3) for evaluation purposes. Following the steps similar to Paquet's [1]. There is a mix of classical operators that were previously defined in [1, 56], such as **first**, **next**, **fby**, **wvr**, **upon**, and **asa**.The collection of the translated operators is summarized in Figure 17. Following the same reasons of (im)practicality as in the PoC implementation of Lucx's parser and semantic analyzer [3], we don't translate some MARFL constructs. Instead, dedicated interpreter plug-ins are designed to work directly with the untranslated constructs, and their operators.

$$
\begin{aligned}
\textbf{first}\ X &= X@0 && (81)\\
\textbf{last}\ X &= X@(\#@(\#\textbf{iseod}(\#)-1)) && (82)\\
\textbf{next}\ X &= X@(\#+1) && (83)\\
\textbf{prev}\ X &= X@(\#-1) && (84)\\
X\ \textbf{fby}\ Y &= \textbf{if}\ \#=0\ \textbf{then}\ X\ \textbf{else}\ Y@(\#-1) && (85)\\
X\ \textbf{pby}\ Y &= \textbf{if}\ \textbf{iseod}\ \#\ \textbf{then}\ X\ \textbf{else}\ Y@(\#+1) && (86)\\
X\ \textbf{wvr}\ Y &= X@T\ \textbf{where} && (87)\\
&\qquad T = U\ \textbf{fby}\ U@(T+1)\\
&\qquad U = \textbf{if}\ Y\ \textbf{then}\ \#\ \textbf{else}\ \textbf{next}\ U\\
&\quad \textbf{end}\\
X\ \textbf{asa}\ Y &= \textbf{first}\ (X\ \textbf{wvr}\ Y) && (88)\\
X\ \textbf{ala}\ Y &= \textbf{last}\ (X\ \textbf{rwvr}\ Y) && (89)\\
X\ \textbf{upon}\ Y &= X@W\ \textbf{where} && (90)\\
&\qquad W = 0\ \textbf{fby}\ (\textbf{if}\ Y\ \textbf{then}\ (W+1)\ \textbf{else}\ W)\\
&\quad \textbf{end}\\
X\ \textbf{and}\ Y &= X\&\&Y && (91)\\
X\ \textbf{or}\ Y &= X||Y && (92)\\
X\ \textbf{xor}\ Y &= \textbf{not}((X\ \textbf{and}\ Y)\ \textbf{or}\ \textbf{not}\ (X\ \textbf{or}\ Y)) && (93)
\end{aligned}
$$

Figure 17: Operators translated to GIPL-compatible definitions [1]

3.2.2 MARFL Context Operators

While the operators presented so far in the preceding sections are designed to support MARFL context in addition to their traditional behavior, the operators presented here were designed to work specifically with MARFL to specify if the intention is to

train or classify in the pipeline. These operators were introduced in the initial syntax provided by Mokhov in [6] and are defined as:

Definition 18. ***train*** *is an extension of a free function that translates to a concrete subsystem's function for starting the training procedure. It is a pre-defined procedural demand which can start the training phase on the loaded dataset within the selected subsytem.*

Definition 19. ***classify*** *is also an extension of a free function that translates to a concrete subsystem's function for starting the classification procedure. It is a parametrized procedural meta function corresponding to the selected subsystem which creates dynamic procedural demands.*

3.3 Operational Semantics

Like the syntax, the operational semantics of MARFL takes advantage of the semantic rules of GIPL [1] (Section 2.1.1.3, page 19), Indexical Lucid [100], Objective Lucid [5], and Lucx [2], JOOIP [55], and Forensic Lucid [26]. It also takes inspiration from the work in [6], augmented with the new operators, and definitions. We specify resulting semantic definitions in MARFL along with the explanation of the rules and the notation. We use the same notation as the referenced languages to maintain consistency in defining our rules.

The rules are grouped in several figures: the basic core rules are in Figure 3, additional rules for MARFL are shown in Figure 18. What follows are notes on the additional details of rules of interest.

1. The evaluation context environment $\mathcal{P}$ defines a point in the multidimensional context space at which an expression E is to be evaluated [1, 53]. $\mathcal{P}$ changes when the `@` operator is evaluated, or a dimension is declared in a `where` clause. It is a set of $\langle dimension : tag \rangle$ mappings, associating each dimension with a tag index over this dimension. $\mathcal{P}$ is thus a partial function:

$$\mathcal{P} : \mathbf{E_I} \rightarrow \mathbf{T} \tag{100}$$

$$\mathbf{E_{cid}} \quad : \quad \frac{\mathcal{D}(id) = (\texttt{const}, c)}{\mathcal{D}, \mathcal{P} \vdash id : c} \tag{94}$$

$$\mathbf{E_{opid}} \quad : \quad \frac{\mathcal{D}(id) = (\texttt{op}, f)}{\mathcal{D}, \mathcal{P} \vdash id : id} \tag{95}$$

$$\mathbf{E_{op}} \quad : \quad \frac{\mathcal{D}, \mathcal{P} \vdash E : id \qquad \mathcal{D}(id) = (\texttt{op}, f) \qquad \mathcal{D}, \mathcal{P} \vdash E_i : v_i}{\mathcal{D}, \mathcal{P} \vdash E(E_1, \ldots, E_n) : f(v_1, \ldots, v_n)} \tag{96}$$

$$\mathbf{E_w} \quad : \quad \frac{\mathcal{D}, \mathcal{P} \vdash Q \ : \ \mathcal{D}', \mathcal{P}' \qquad \mathcal{D}', \mathcal{P}' \vdash E : v}{\mathcal{D}, \mathcal{P} \vdash E \ \texttt{where} \ Q : v} \tag{97}$$

$$\mathbf{Q_{id}} \quad : \quad \frac{}{\mathcal{D}, \mathcal{P} \vdash id = E \ : \ \mathcal{D}\dagger[id \mapsto (\texttt{var}, E)], \mathcal{P}} \tag{98}$$

$$\mathbf{QQ} \quad : \quad \frac{\mathcal{D}, \mathcal{P} \vdash Q \ : \ \mathcal{D}', \mathcal{P}' \qquad \mathcal{D}', \mathcal{P}' \vdash Q' : \mathcal{D}'', \mathcal{P}''}{\mathcal{D}, \mathcal{P} \vdash Q \ Q' \ : \ \mathcal{D}'', \mathcal{P}''} \tag{99}$$

Figure 18: Operational semantics rules of MARFL: E and Q Core

where $\mathbf{E_I}$ is a set of possible dimension identifiers, declared statically or computed dynamically. $\mathbf{T}$ is a tag set.

In traditional LUCID semantics, the tag set $\mathbf{T} = \mathbf{N}$, i.e., a set of natural numbers [1]. In LUCX's extension, T = U is any enumerable tag set [2], that may include various orderings of numerical or string tag sets, both finite and infinite ([3]). In MARFL, $\mathbf{T}$ is a set of tags corresponding to individual parameter configuration for the selected algorithm in a MARF instance.

2. The initial definition environment includes the predefined operators, the constants, and $\mathcal{P}_0$ defines the initial context of evaluation [1, 2]. Thus, $\mathcal{D}_0, \mathcal{P}_0 \vdash E : v$ represents the computation of any marfl expression E resulting in value v.

3. The semantic operator $\dagger$ represents the addition of a mapping in the definition environment $\mathcal{D}$, associating an identifier with its corresponding semantic record, here represented as a tuple [1, 55].

4. The function $\mathcal{P}' = \mathcal{P}\dagger[id \mapsto v'']$ specifies that $\mathcal{P}'(x)$ is v'' if $x = id$, and $\mathcal{P}(x)$ otherwise.

5. The rule for the `where` clause, $\mathbf{E_w}$ (Figure 18, rule 97), which corresponds to the syntactic expression E `where` Q, evaluates E using the definitions Q therein.

6. The additions to the definition environment $\mathcal{D}$ and context of evaluation $\mathcal{P}$ made by the **Q** rules (Figure 19, rule 108; Figure 18, rules 98) are local to the current `where` clause. This is represented by the fact that the $\mathbf{E_w}$ rule returns neither $\mathcal{D}$ nor $\mathcal{P}$.

7. The evaluation rule for the navigation operator `@`, $\mathbf{E_{at(cxt)}}$ (106), which corresponds to the syntactic expression E `@` E', evaluates E in context E' [2].

8. The semantic rule $\mathbf{E_{construction(cxt)}}$ (105, Figure 19) evaluates $[E_{d_1} : E_{i_1}, \ldots, E_{d_n} : E_{i_n}]$ to a simple context [2]. It specifically creates a context as a semantic item and returns it as a context $\mathcal{P}$ that can then be used by the rule 106 to navigate to this context by making it override the current context.

9. The semantic rule 96 is valid for the definition of the context operators, where the actual parameters evaluate to values v_i that are contexts $\mathcal{P}_i$.

10. The semantic rule 104 expresses that the `#` symbol evaluates to the current context. When used as a parameter to the context calculus operators, this allows for the generation of contexts relative to the current context of evaluation [1, 2, 29].

3.4 Summary

To summarize, we start by explaining the core concept of higher order context which was initially implemented in Lucx and is adapted into our research work. We provide the formal syntax for MARFL followed by the definitions for core operators as well as the newly introduced MARFL context operators. We go on to provide the core operational semantic rules as well as context specific semantic rules for our language. MARFL offers a new way to formalize pattern recognition tasks by providing the testing parameters in form of high order contexts as well as accessibility to a wider audience due to the simpler nature of Lucid-based languages. This fulfils our Objective O1 in Section 1.5.

$$\mathbf{E_{did}} \quad : \quad \frac{\mathcal{D}(id) = (\texttt{dim})}{\mathcal{D}, \mathcal{P} \vdash id : id} \tag{101}$$

$$\mathbf{E_{E.did}} \quad : \quad \frac{\mathcal{D}(E.id) = (\texttt{dim})}{\mathcal{D}, \mathcal{P} \vdash E.id : id.id} \tag{102}$$

$$\mathbf{E_{tag}} \quad : \quad \frac{\mathcal{D}, \mathcal{P} \vdash E : id \qquad \mathcal{D}(id) = (\texttt{dim})}{\mathcal{D}, \mathcal{P} \vdash \#E : \mathcal{P}(id)} \tag{103}$$

$$\mathbf{E_{\#(cxt)}} \quad : \quad \frac{}{\mathcal{D}, \mathcal{P} \vdash \# : \mathcal{P}} \tag{104}$$

$$\mathbf{E_{construction(cxt)}} \quad : \quad \frac{\begin{array}{c}\mathcal{D}, \mathcal{P} \vdash E_{d_j} : id_j \qquad \mathcal{D}(id_j) = (\texttt{dim}) \\ \mathcal{D}, \mathcal{P} \vdash E_{i_j} : v_j \qquad \mathcal{P}' = \mathcal{P}_0 \dagger [id_1 \mapsto v_1] \dagger \ldots \dagger [id_n \mapsto v_n]\end{array}}{\mathcal{D}, \mathcal{P} \vdash [E_{d_1} : E_{i_1}, E_{d_2} : E_{i_2}, \ldots, E_{d_n} : E_{i_n}] : \mathcal{P}'} \tag{105}$$

$$\mathbf{E_{at(cxt)}} \quad : \quad \frac{\mathcal{D}, \mathcal{P} \vdash E' : \mathcal{P}' \qquad \mathcal{D}, \mathcal{P} \dagger \mathcal{P}' \vdash E : v}{\mathcal{D}, \mathcal{P} \vdash E \mathbin{@} E' : v} \tag{106}$$

$$\mathbf{E_{dot}} \quad : \quad \frac{\mathcal{D}, \mathcal{P} \vdash E_2 : id_2 \qquad \mathcal{D}(id_2) = (\texttt{dim})}{\mathcal{D}, \mathcal{P} \vdash E_1.E_2 : tag(E_1 \downarrow \{id_2\})} \tag{107}$$

$$\mathbf{Q_{dim}} \quad : \quad \frac{}{\mathcal{D}, \mathcal{P} \vdash \texttt{dimension}\ id\ :\ \mathcal{D} \dagger [id \mapsto (\texttt{dim})], \mathcal{P} \dagger [id \mapsto 0]} \tag{108}$$

(109)

Figure 19: Operational semantics rules of MARFL: E and Q Core Context

Chapter 4

Implementing MARFL in GIPSY

This chapter discusses the proposed software design and implementation aspects behind MARFL (presented in the preceding chapter). This includes specific contributions to GIPSY in terms of its GIPC and GEE frameworks redesign to support the MARFL compilation and run-time. The architectural design centers around the MARFL parser and semantic analyzer as well as various re-design details of GEE to support MARFL execution. W e present the necessary architectural design concepts, frameworks, and some of their PoC implementation.

4.1 MARFL Compiler

The general design approach (Section 2.3.1, page 37) for adding a new SIPL compiler calls for implementing the `IIntensionalCompiler` interface augmented with a specific IPL parser and a semantic analyzer. Accordingly, we add the corresponding new MARFL compiler framework to GIPC of GIPSY. We create a JavaCC [101] grammar to generate a MARFL parser that is then integrated into the GIPC framework followed by an extension of the MARFL semantic analyzer for specific MARFL AST nodes introduced in this work, such that GEE can evaluate them at run time. We likewise introduce the `MARFL FormatTag` to handle MARFL-specific constructs. These constitute annotations of the AST nodes that allow GEE's evaluation engines to deal appropriately with them when an interpreter encounters

such nodes. The `MARFL` annotation allows to invoke the appropriate operators from the GIPSY type system at run-time.

4.1.1 MARFL Parser

Following the tradition of many GIPC's parsers, MARFL's grammar (in accordance with its syntax presented in Section 3.2, page 54) is specified in a particular grammar format. We use JAVA *Compiler Compiler* (JavaCC) [101] to generate the parser for MARFL. We give a brief overview of JavaCC and its specifications.

JavaCC along with the built-up JJTree, is the tool the GIPSY project is relying on since the first implementation [27] to create JAVA-language parsers and ASTs for source grammar files. The JAVA Compiler tool implements the same idea for JAVA, as do `lex`/`yacc` [102] (or `flex`/`bison`) for C – reading a source grammar they produce a parser that complies with this grammar and gives you a handle on the root of the abstract syntax tree. JavaCC requires a grammar specification of the target programming language, written in the syntax of JavaCC.

4.1.2 MARFL Semantic Analyzer

MARFL's semantic analyzer's design is primarily [3], because LUCX is not fully translated into GIPL and because MARFL adds new constructs, such as machine learning subsystem contexts that don't have yet any known translation algorithm into GIPL. Thus, `MARFLSemanticAnalyzer` is created to account for the new node types in AST corresponding to the extensions. `MARFLSemanticAnalyzer` capitalizes on the earlier semantic analyzers implemented by Tong [3] and Wu [28].

`MARFLSemanticAnalyzer`'s responsibility is to ensure that the static compiled AST and the `Dictionary` of identifiers adhere to the declarative aspects of the MARFL language's operational semantics described in Section 3.3, page 62. The semantic analyzer traverses the AST that came out of the JJTree tool of JavaCC top-down/depth-first to do the static type-checking, identifier scope and definition checks, initial rank analysis, and other typical semantic analysis tasks [28]. The differences

from the traditional GIPL- and INDEXICAL LUCID-based dialects are that additional type checks are done for LUCX-derived simple contexts and context sets [3].

Since not all of the semantic checks can be done at the compile time, the run-time evaluation engines does some of the run-time checks during program execution. This fulfils our Objective O2 presented in Section 1.5.

4.2 Updates to GIPSY's Frameworks' Design

To facilitate the addition and execution of MARFL into the GIPSY evaluation platform, number of design changes were carried out in various GIPSY frameworks, including GIPC and GEE to support MARFL. GEE's LUCID interpretation module (`Interpreter`) was already upgraded during the implementation of FORENSIC LUCID by Serguei Mokhov [26] to be `GIPSYContext`-aware and use other GIPSY type system's types consistently (as its previous iteration [72] was using arrays of integers to represent the context of evaluation and did not support any of the new LUCX constructs).

4.2.1 General Design Overview

In Figure 20 is a general conceptual design overview of the MARFL compilation and evaluation process involving various components and systems. Of main interest to this work are the inputs to the compiler—the MARFL fragments (subsystem algorithms that represent the stages of a pattern recognition pipeline) and programs act as input. The exact specifications for selected algorithms, choice of training or classification phase with the dataset path are combined to form a MARFL program. The complete specification is then processed by the compiler depicted as GIPC on the image (the General Intensional Program Compiler) through the invocation of the MARFL SIPL compiler that is aware of the MARFL constructs, such as the sample loader, machine learning subsystem, their properties along with operators detailed in Section 3.2, page 54. The compiler produces an intermediate version of the compiled program as an AST and a contextual dictionary of all identifiers

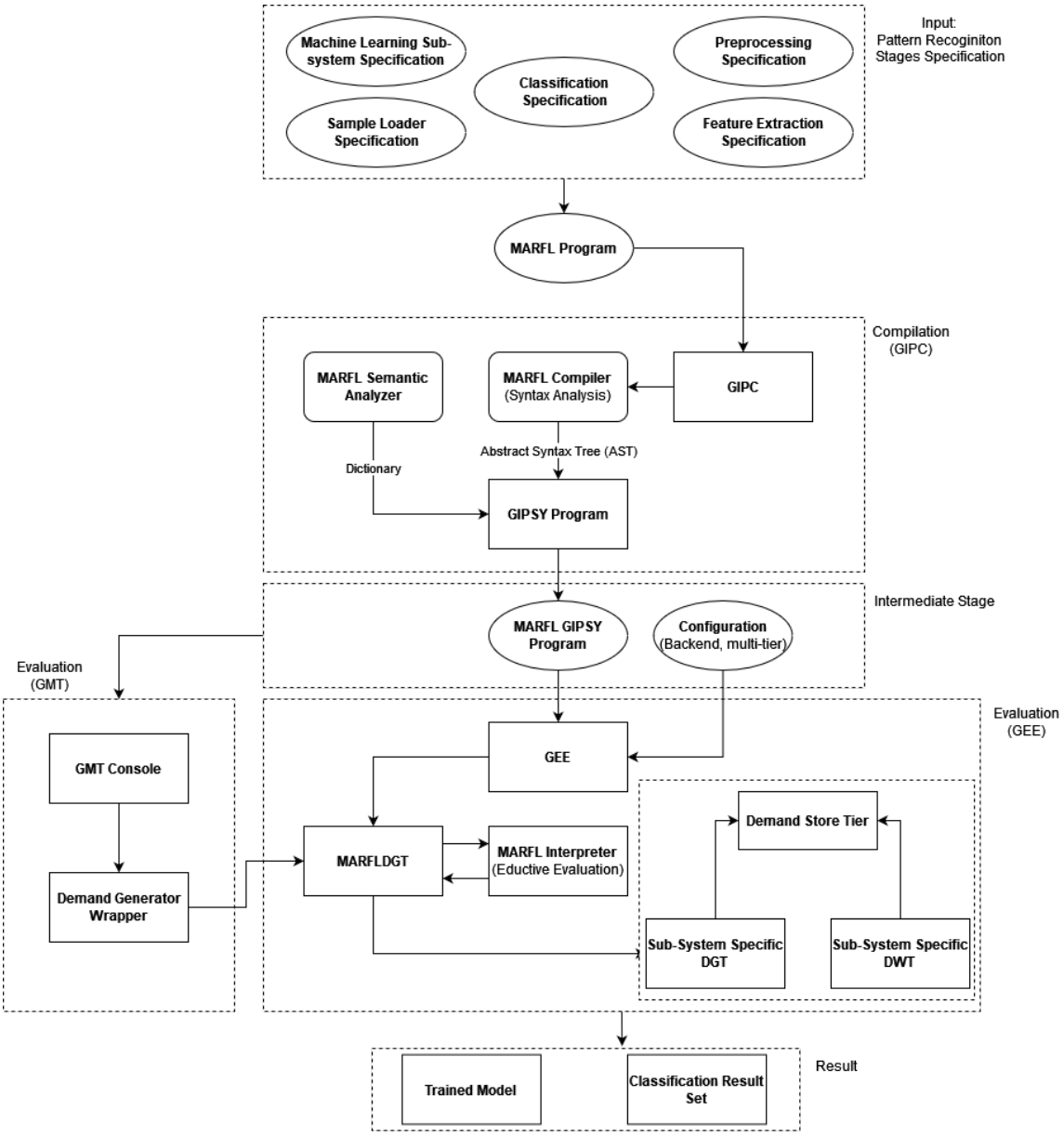

Figure 20: MARFL Compilation and Evaluation flow in GIPSY

(e.g., context identifiers for each stage), encapsulated in a GEER that evaluation engines (under the GEE component) understand. The compiled GEER and engine configuration (either a default, or a custom file) can be now fed to either the GEE for processing by one or more evaluation engines or to the General Manager Tier (GMT). Either way it reaches the `MARFLDGT` which calls for `MARFLInterpreter` for evaluation of the MARFL program. Upon evaluation, an instance of DST is started and workers are assigned using the sub-system specific DWT. The sub-system specific

DGT generates and stores demands in the DST which are then picked up by workers, which essentially replace those demands with the results using the same generated signature inside the DST. The said MARFL evaluation engines are designed to use the traditional eduction architecture while processing the GEER.

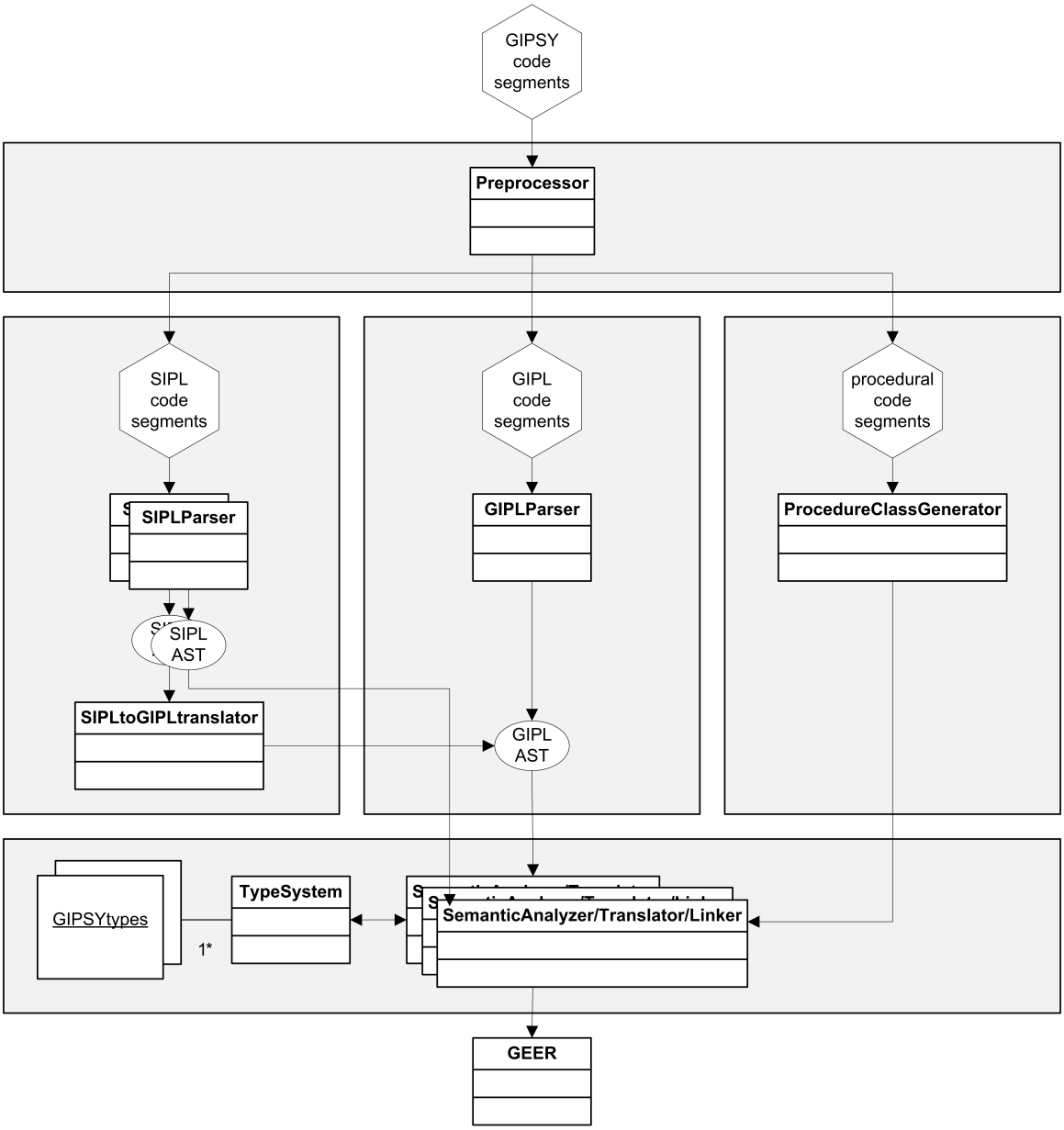

Figure 21: Updated high-level structure of the GIPC framework

In Figure 21 is the updated high-level structure of GIPC (cf. Figure 10, page 36 in Section 2.3). This design presents and follows the same framework for semantic analyzers that was introduced in [26, 29]. Introduced semantic analyzers may behave differently for different SIPLs. As stated earlier, the modifications (detailed further)

comprise the fact that MARFL SIPL produces an AST that is not necessarily translated (fully, or partially) into GIPL (as is the case of Lucx and Forensic Lucid) bypassing the `SIPLtoGIPLtranslator`. For such cases in the past, the specific semantic analyzers were created to work specifically with such ASTs. The specific semantic analyzers are aware of any additional types in the type system particular to the language in question and rely on its validating methods. In figure, the semantic analyzers, translators, and GEER linkers are conceptually represented as one class simply because these modules work together closely. At the implementation level, they are separate Java classes.

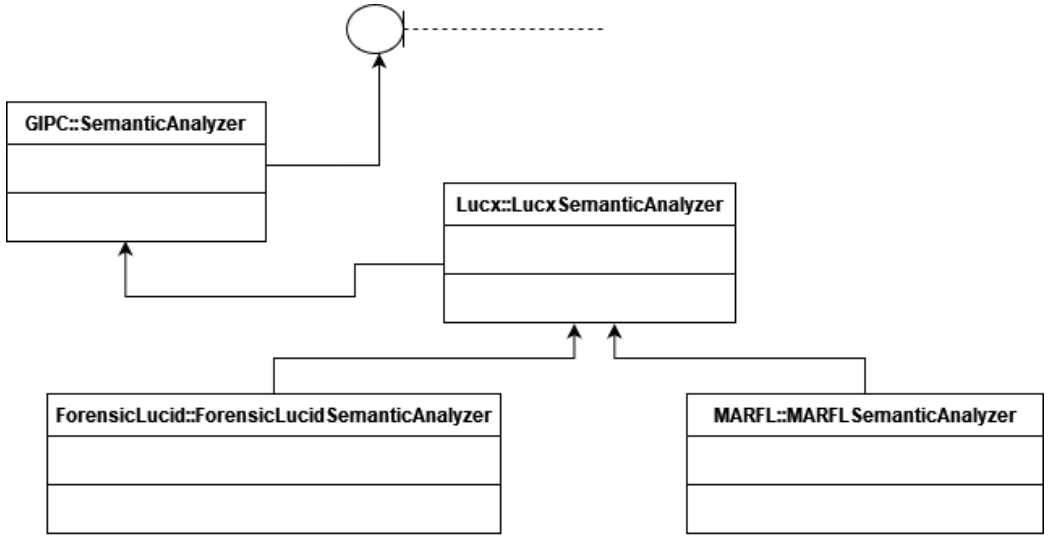

Figure 22: Semantic analyzers framework

At present, the semantic analyzers framework is represented by the `ISemanticAnalyzer` interface and has three concrete instances implementing it: `SemanticAnalyzer`, which is the general analyzer that is originally capable of handling of classical GIPL, Indexical Lucid, JLucid, Objective Lucid, and JOOIP programs (combined with the procedural code linking for the hybrid dialects in `ProcedureClassGenerator`) contributed to by Wu [28, 55] and Mokhov [5]. Then follow `LucxSemanticAnalyzer`, produced by Tong [3], the `ForensicLucidSemanticAnalyzer` produced in [26] by Mokhov, and the `MARFLSemanticAnalyzer` produced for the work presented here on MARFL following the inheritance hierarchy for convenience and code re-use for many of the semantic rules. All semantic analyzers adhere to the same

API and produce a GEER, corresponding to their syntactical and semantic annotations. In Figure 22 is the class diagram corresponding to the just described framework. Using `ForensicLucidSemanticAnalyzer` as inspiration, `MARFLSemanticAnalyzer` primarily inherits from `LucxSemanticAnalyzer` to work with simple context, and context set type checks. Additional checks are added to work with MARFL specific contexts and operators.

4.2.2 Execution Engine Redesign

Semantically speaking, the interpretation of a Lucid program happens at the generator components, DGTs, because this is where the eductive evaluation engines reside. The aforementioned compiled `GIPSYProgram` (a GEER instance) containing the annotated AST as well as the `Dictionary` of identifiers (corresponding to the initial $\mathcal{D}_0$), and definitions, is passed to GEE (Figure 20, page 69). GEE then hands them over to an interpreter. Traditionally, there was only one interpreter that could handle the compiled GIPL as its only input language. Then this design morphed, during implementation of Lucx, into a more general architecture to allow more than one interpreter and demand generator types. Thus, to add interpretation of non-translated-to-GIPL dialects like Forensic Lucid or Lucx, and in our case MARFL, the architecture is updated to replace the `Interpreter` component in the DGT based on the either a command-line option or the annotated node tag in the GEER's AST. In this process the DST, the primary demand propagation machinery, does not change. However, the worker side of DWT may occasionally change, depending on the type of procedural demands made by the user.

The framework to support this was implemented during the implementation of Forensic Lucid by Mokhov in [26]. It allowed for multiple evaluation engines interpreting new input language material. In the case of MARFL, we override standard DGT's `Executor` to be able to spawn `MARFLDGT` which is the main Java class responsible for calling the newly introduced `MARFLInterpreter` which delegates demands based on the machine learning subsystem it gets from *eductive* evaluation of the AST. `MARFLInterpreter` overrides the standard `Interpreter` and is responsible

for traversing the annotated AST in the `GIPSYProgram`. The generality of this design benefits us in implementing problem-specific (PS) DGTs for different machine learning subsystems like `TensorFlow` can have `TensorFlowDGT` and `TensorFlowDWT`, etc. Currently, we only support using MARF subsystem for our pattern recognition tasks which uses the MARFCAT PS DGT and DWT.

This non-GIPL engine software layer differentiation allows us to cater to problem-specific-tiers as well as non-GIPL languages to be able to take advantage of the GEE's distributed middleware infrastructure by improving the GEE framework making it more flexible. This allows various programming language paradigms to be included into GIPSY that are not necessarily compatible with GIPL or the translation effort to undertake is too great to be effective, error-free, or efficient as shown in [3]. The addition of following components is towards fulfilment of our Objective O3.

4.2.2.1 MARFLDGT Components

`MARFLDGT` is our main JAVA program that is responsible for generating and storing the demands into DST. It closely follows and inherits its functionalities from the legacy `DGTWrapper` that handles tiered demand generation. It is responsible for loading the compiled MARFL GIPSY program which is passed onto `MARFLInterpreter` for evaluation. As discussed in Section 4.2.2, if the evaluated system is MARF it will call `MARFCATDGT` and so on for other machine learning sub-systems. Below, we describe the functionality of various methods that are implemented for `MARFLDGT`. The class diagram for `MARFLDGT` is shown in Figure 23.

- **init()** - This method initializes the `MARFLDGT` object, setting up the necessary properties from the configuration object such as base directory for the metadata, training data filename, MARFL GIPSY program as well as a default configuration for the machine learning sub-system.

- **startTier()** - This method overrides the default method in `DGTWrapper` to create a new instance of a TA (Transport Agent) using the configuration object passed via reflection property in JAVA. It also creates a new instance of a

`DemandDispatcher` object using the implementation class name specified in the configuration file. In our case we use the Jini DMS as our dispatcher due to scalability reasons mentioned in Section 2.3.4.

- **stopTier()** - This simple method is responsible for stopping the allocated tiers during execution.
- **run()** - This is the main processing method which loads the GIPSY program, evaluates it using `MARFLInterpreter`, updates the configuration string and starts the sub-system specific DGT.
- **updateConfigString()** - This method uses a hashmap generated via evaluation in `MARFLInterpreter` to update the configuration string which tells the system to use which particular algorithm to call for specific pattern recognition stages.

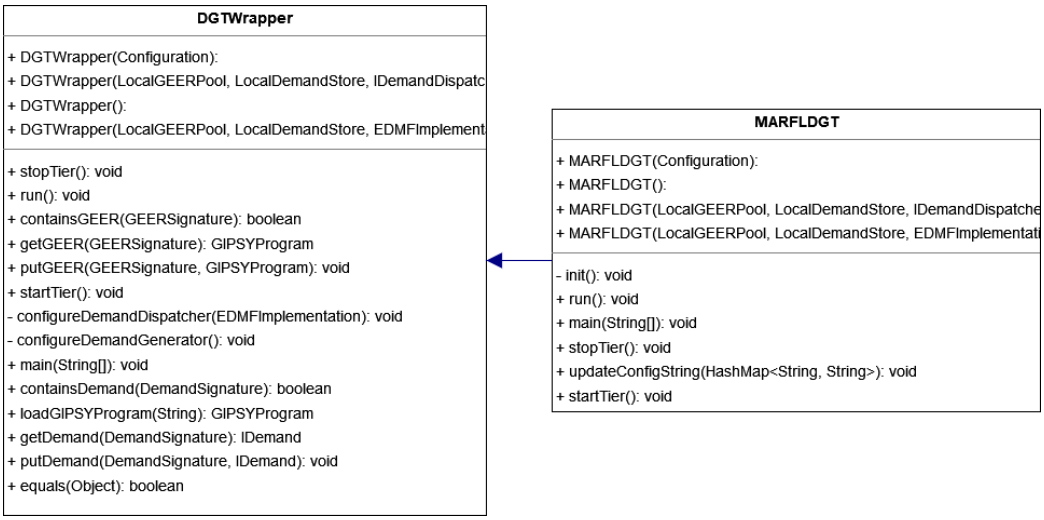

Figure 23: Class diagram for MARFLDGT

4.2.2.2 MARFL Interpreter Components

As introduced earlier, `MARFLInterpreter` is a JAVA program responsible for *eductive* evaluation of our GIPSY program by traversing the AST generated during the compilation phase of the MARFL program in GIPC. Some functionality is borrowed from the already existing `LegacyInterpreter` framework in GEE. We extend on top

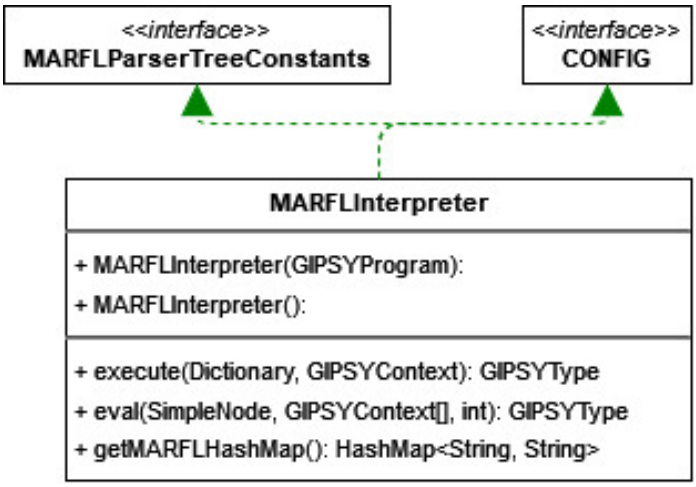

Figure 24: Class diagram for MARFL interpreter

of it so that we can evaluate the newly introduced MARFL specific AST nodes. A serialized GIPSY program is loaded into our `MARFLDGT` which then extracts the AST forwarding it to `MARFLInterpreter`. Below, we describe the functionality of various methods that are implemented inside `MARFLInterpreter`. As shown in the class diagram (Figure 24) for `MARFLInterpreter`, it uses `MARFLParserTreeConstants` generated from JavaCC to handle MARFL specific AST nodes.

- `eval()` - This is our primary method which traverses the AST. Based on the annotated AST nodes it encounters, it either updates a `HashMap`, or starts a sub-system specific DGT, or recursively calls the children of node until the entire syntax tree is traversed. The `HashMap` includes specific algorithms to be used for each pattern recognition stage as well as which sub-system to delegate these tasks to.

- `execute()` - This method is responsible for handling legacy execution and storing the results from the eval() method into a dictionary based on the context it is evaluated in.

- `getMARFLHashMap()` - This method returns the HashMap which contains the details of algorithms and sub-system based on the evaluation of AST.

4.3 Compiling and Executing MARFL on GIPSY

4.3.1 Compilation Phase - GIPC

As shown in Figure 20, we first need to compile a user generated MARFL program which states the pattern recognition task specifications. The GIPC component of GIPSY platform has been updated with the MARFL compiler, as explained in Section 4.1 to handle the compilation of our new LUCID dialect. The following command can be used to compile a MARFL program.

```
java -jar gipc.jar --marfl [FILENAME]
```

The `-marfl` tag makes sure that GIPC uses the MARFL SIPL compiler to compile the passed IPL program. The tag also ensures that GIPC calls for `MARFLSemanticAnalyzer`, described briefly in Section 4.1.2 instead of regular GIPL `SemanticAnalyzer`, to perform semantic analysis on the program. The name of MARFL program, with extension *.ipl* needs to be included as last argument to this command. Since MARFL is a conservative extension of LUCID, it can parse any previous LUCID dialects as well.

A sample MARFL program is shown in Listing 4.1. The program specifies to use MARF as its machine learning sub-system and classify the `wireshark-1.2.0_train.xml` dataset using `FFT` and `Chebyshev Distance` algorithm for feature extraction and classification respectively.

```
classify "wireshark-1.2.0_train.xml" @ [ mls, sl, pr, fe, cl ]
where
    ml_subsystem        mls = MARF;
    sampleLoader        sl = XML [ encoding : utf-8 ]
                            where
                                dimension encoding;
                            end;
    preprocessing       pr = RAW_PREPROCESSING [ cutoff: 2024, windowsize: 2048 ]
                            where
                                dimension cutofff, windowsize;
                            end;
    featureExtraction   fe = FFT [ poles: 40 ]
                            where
```

```
                                dimension poles;
                            end;
    classification      cl = CHEBYSHEV_DISTANCE [ r : 5 ]
                            where
                                dimension r;
                            end;
end
```

Listing 4.1: Sample MARFL program

Upon compilation, GIPC produces a GIPSY program with extension *.gipsy* which is a serialized JAVA object containing the annotated AST from JJTree and a dictionary of identifiers from the `MARFLSemanticAnalyzer`. The obtained AST, shown in Listing 4.2, can also be printed via debug (pass –debug option to gipc.jar) mode for user convenience.

```
AT
  CLASSIFY
   STRING : "wireshark-1.2.0_train.xml"
  WHERE
   ARRAY
    ID : mls
    ID : sl
    ID : pr
    ID : fe
    ID : cl
   ML
    ID : mls
    ML_SUBSYSTEM
     MARF
   SL
    ID : sl
    SAMPLE_LOADER
     XML
    WHERE
     SIMPLECONTEXT
      CONTEXT_ELEMENT
       ID : encoding
       ID : utf8
     DIMENSION
      ID : encoding
   PR
    ID : pr
```

```
  PRE_PROCESSING
   RAW_PREPROCESSING
  WHERE
   SIMPLECONTEXT
    CONTEXT_ELEMENT
     ID : cutoff
     INTEGER : 2024
    CONTEXT_ELEMENT
     ID : windowsize
     INTEGER : 2048
   DIMENSION
    ID : cutofff
    ID : windowsize
 FR
  ID : fe
  FEATURE_EXTRACTION
   FFT
  WHERE
   SIMPLECONTEXT
    CONTEXT_ELEMENT
     ID : poles
     INTEGER : 40
   DIMENSION
    ID : poles
 CL
  ID : cl
  CLASSIFICATION
   CHEBYSHEV_DISTANCE
  WHERE
   SIMPLECONTEXT
    CONTEXT_ELEMENT
     ID : r
     INTEGER : 5
   DIMENSION
    ID : r
```

Listing 4.2: AST generated for MARFL program

4.3.2 Execution Phase - GEE

As described in Figure 20, the execution a MARFL GIPSY program happens on a tiered architecture inside GEE. There are some preliminary services required for the correct execution of our program. We first require a middleware service such as Jini.

Jini DMS (Demand Migration System) is responsible for incorporating the demand store with the distribution architecture. Once the service is active, we assign a GMT node that can manage DST, DGT and DWT allocation. To do so one of the following command need to be executed:

```
sudo ./startMARFLGMTNode.sh
sudo ./startMARFLGMTNode.sh -n gmtd
```

The first command starts a GEE instance with a GMT console window which uses JAVA JFrames to communicate allocation of other tiers by taking further input from the user. However, the second command is more user friendly as it uses a GMT daemon to automatically start a GEE instance, bypassing the need for a GMT console window. `startMARFLGMTNode.sh` executable file uses a configuration file shown in Listing 4.3.

```
gipsy.GEE.multitier.GMT.tierID=manager
gipsy.GEE.multitier.Node.GMTConfigs=GMT.config;GMTJini.config
gipsy.GEE.multitier.Node.DSTConfigs=JiniDST.config;JMSDST.config
gipsy.GEE.multitier.Node.DGTConfigs=marflDGT.config
gipsy.GEE.multitier.Node.DWTConfigs=marfcatDWT.config
```

Listing 4.3: MARFL GMT Node Config for MARF

The configuration file communicates with GEE to start a specific DST, Jini DST in our case. It also includes the configuration files for MARFL specific DGT and a sub-system specific DWT. Once the initial setup is done, we can use the GMT daemon controller via a *gmtc.sh* script present on the GIPSY platform, to start allocation of our demand store manager, workers and demand generators by using following commands:

```
./gmtc.sh allocate NodeIndex DST JiniDST.config [number of instances to start]
./gmtc.sh allocate NodeIndex DWT marfcatDWT.config DSTIndexAtGMT [number of instances to start]
./gmtc.sh allocate NodeIndex DGT marflDGT.config DSTIndexAtGMT [number of instances to start]
```

The configuration file for DWT can be updated based on the sub-system selected for pattern recognition task. The `marflDGT.config` file is responsible for starting the `MARFLDGT`(Section 4.2.2.1) on a GEE node via a JAVA feature, reflection. Reflection

allows us to manipulate the properties inside a executed JAVA program at run-time, essentially allowing us to start a sub-system specific DGT based on AST evaluation. Other properties include the name of compiled GIPSY program, the name of the dataset to be used. It also includes default configuration properties for a machine learning sub-system. The last two properties are updated inside `MARFLDGT` based on the evaluation of AST from `MARFLInterpreter`. A sample of default `marflDGT.config` for MARF sub-system is shown in Listing 4.4.

```
gipsy.GEE.multitier.wrapper.impl=gipsy.GEE.multitier.DGT.marfl.MARFLDGT
gipsy.GEE.multitier.DGT.DemandDispatcher.impl=jini.rmi.JiniDemandDispatcher
marfcat.meta.basedir=.
marfcat.meta.filename=wireshark-1.2.0_train.xml
marfcat.args=--batch-ident -nopreprep -raw -fft -cheb --dgt
marfl.gipsy=wireshark-classify.gipsy
```

Listing 4.4: Default Configuration for MARF

4.4 Summary

This chapter focuses on the architectural design details for the evaluating system of GIPSY along with the related work and PoC implementation aspects. This includes the necessary updates, modifications, and contributions to the design of the GIPSY platform; which includes the updated compiler (GIPC) and the updates to the semantic analyzers framework; the run-time system (GEE) with the additional engines for MARFL. The JavaCC grammar specification of the MARFL parser can be found in GIPSY's source code repository. Its generated compiler sources are integrated in `gipsy.GIPC.intensional.SIPL.MARFL`. We described the new classes and their corresponding methods for problem-specific DGT for MARFL that is implemented inside GEE followed by the execution procedure of a MARFL program. A sample of a MARFL GIPSY program along with its generated AST is also discussed for better understanding. Our research Objective O4 is fulfilled by the procedure described in Section 4.3.

Chapter 5

Evaluation and Results

In Chapter 3, we presented the formal syntax and operational semantics for our proposed intensional language, MARFL. We also explained the necessary architectural modifications to the GIPSY platform to enable the compilation and execution of MARFL programs, which can perform pattern recognition tasks in Chapter 4. Moreover, we outlined the possible steps to execute a compiled MARFL program on a single machine or a distributed architecture.

This chapter starts by comprehensively describing the evaluation methodology that we will follow in the following sections. We present the results of our experiments that provide an objective means of verifying that the requirements stated in Section 1.5 are met.

5.1 Evaluation Methodology

In this section, we provide the reader with methods we follow in this research work to evaluate that our solution truly solves the problems we mentioned in Section 1.3 and that our proposed solution meets the objectives we described in Section 1.5. We propose the following to evaluate our solution:

- In order to measure the ability of our proposed language, MARFL and to verify our Objective O1, we will represent three different pattern recognition

tasks, which were initially done using JAVA, in MARFL. We will compare the number of lines of code required to script these tasks using our solution versus the original implementation. This comparison will highlight the ease of use and efficiency of our *intension* based solution compared to using conventional approach.

- In order to assess the effectiveness of the new execution engine and to verify our Objectives O3 and O4, we will conduct a performance evaluation by computing the execution time of our three proposed MARFL programs. This will allow us to compare the execution of our solution to existing JAVA based implementations and determine how this architectural re-design affects execution. To ensure a fair comparison, we will use the same hardware configuration and dataset for all evaluations. We will also compare the output obtained to original implementation to ensure that our solution is producing comparable or better results than their conventional implementations.

- To evaluate the effectiveness of distributed computing in pattern recognition and to verify our Objective O5, we propose measuring the execution time of pattern recognition tasks on distributed systems and comparing it to the non-distributed version. To conduct this evaluation, we will use a distributed architecture consisting of multiple distinct machines, each with its own processing power and memory.

5.2 Evaluation Environment

5.2.1 Environment Specifications

To ensure the reliability and consistency of our experimental results, it was crucial to conduct our research on a homogeneous distributed architecture. In order to achieve this, we deploy our GIPSY architecture and our proposed solution within on Amazon Web Services (AWS) cloud platform. To briefly explain our experimental environment, we created six dedicated Linux-based EC2 (Elastic Cloud Compute)

virtual machine instances. Each instance was assigned a specific role to fulfill during the execution of our experiments. The suitability of EC2 instances for scientific computational tasks has been highlighted in previous research [103]. We chose the same hardware configuration, displayed in Table 3, for each instance.

Configuration	Component	Specifications
EC2 Instance	Memory	32 GB
	Processor	Intel Xeon Platinum 8124M CPU @3.00GHz × 16
	OS	Ubuntu 22.04.3 LTS x86-64

Table 3: Hardware configuration for each EC2 instance

One of our EC2 instance will act as GMT, whereas other instances are assigned the role of acting as workers in DWT. To ensure proper communication between them, we also setup a Network File System (NFS) which allows us to share the same dataset and the required configuration files between them. For communication between the DWT nodes, we setup a Virtual Private Cloud [104] and add all our instances to it. VPC essentially ensures that all of our instances are part of the same network. Figure 25 demonstrates how our EC2 instances are setup to emulate a distributed architecture.

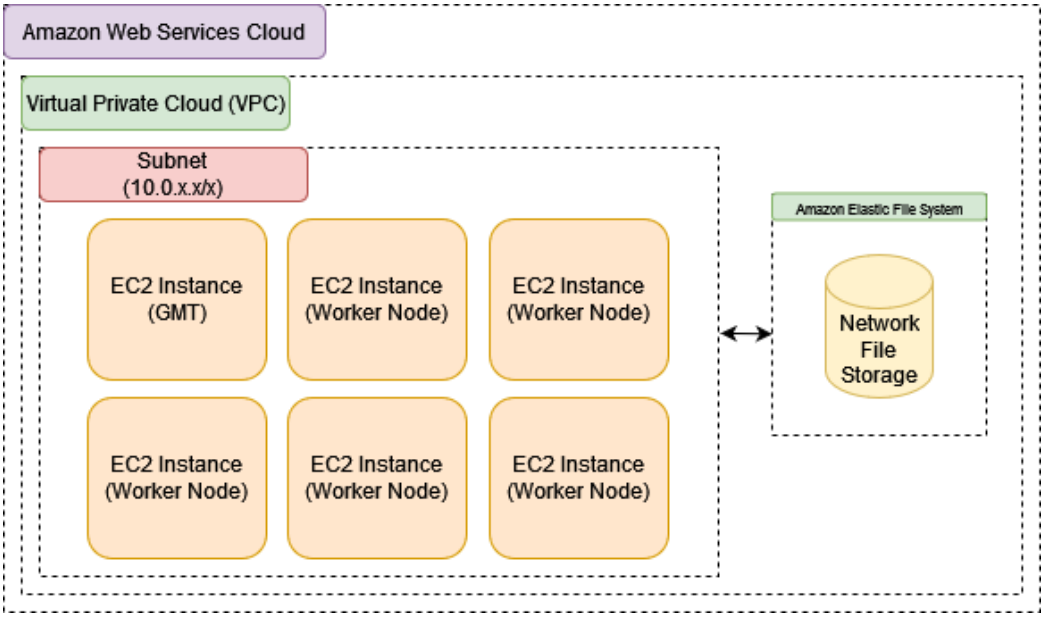

Figure 25: AWS EC2 instances setup

5.2.2 Environment Design

Before we discuss our proposed experiments to evaluate our MARFL execution engine, we would like to briefly familiarize the reader with our evaluation design. As shown in Figure 26, the execution of a MARFL program can be depicted as a linear dataflow graph which consists of three steps: (1) The fetching of training data points from the DST by the DWT; (2) Calling machine learning sub-system specific training or classification methods; (3) Storing the results received from the training or classification method.

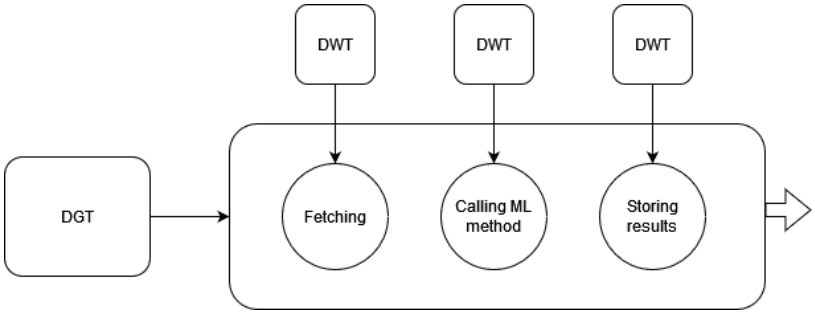

Figure 26: MARFL execution pipeline on GIPSY

In our architecture, demands can exist in two states: PENDING and COMPUTED. The PENDING state indicates that a demand needs to be processed, while the COMPUTED state indicates that the demand's processing has finished. The DGT is responsible for generating demands in the order specified by the data flow graph of the pipeline. When demands enter the DST, they are initially in the PENDING state. The DGT generates fetch demands, which specify the processing of specific data points from the dataset. Each file is assigned a demand signature, typically as a hash code, and then stored in the DST.

Once the DGT has created the fetch demands, the DWTs take on the responsibility of executing these demands. The number of demands directly correspond to the number of training data points in the dataset provided. There can be multiple instances of DWTs. The DWTs perform the required training or classification tasks and store the corresponding results back into the DST. Upon storing the results, the demand transitions into the processed state. The pipeline begins with the DGT generating fetch demands, and the DWTs start their processing. The DGT waits for

these demands to be processed by the DWTs. Once a DWT completes its processing, it generates a result demand to store the obtained result set. This marks the end of the pattern recognition pipeline for that particular demand.

After the results are achieved, both the DGT and DWTs wait for new demands to be processed, while the DST holds the previous demand and its corresponding results. This allows the architecture to handle subsequent demands efficiently, as the DGT and DWTs are ready to process new demands, and the DST retains the historical demand and result data.

5.3 Evaluation of MARFL-specific Engine to Execute MARFL Programs

To complete our Objectives O1 to O4 which includes verification that we can compile and execute MARFL programs on the GIPSY platform following the additions discussed in Section 4.2, we will perform three different pattern recognition tasks. The subsequent sections will describe the following aspects for each task:

- Brief description about the pattern recognition task: An overview of the pattern recognition task is provided, highlighting its relevance as discussed in Section 1.2.1.
- Comparison between the selected task implemented in a conventional language and MARFL: We compare the number of lines of code required to script the same task in a conventional language as compared to MARFL which highlights the specific benefits of MARFL in terms of efficiency and ease of implementation. We also emulate the results achieved for the same machine learning task when done using a conventional approach.

Successful execution of the following experiments will fulfill our Objectives O1, O2, O3, and O4 of allowing a user to script, compile and execute any pattern recognition task in an *intensional* manner. These experiments highlight the ease of our solution as opposed to conventional approaches.

5.3.1 Vulnerable Code Classification in Common Vulnerabilities and Exposures (CVE)

As described in our motivational scenarios Section 1.2.1, MARFL can be used for scripting pattern recognition tasks that can classify vulnerabilities in a given set of source codes. Prior to our solution, as discussed in Section 2.2.4, Mokhov demonstrated a tool called MARF-based Code Analysis Tool, or MARFCAT [63], which inherits MARF's pipeline and testing methodology to learn from the (Common Vulnerabilities and Exposures) CVE-based vulnerable cases. We employ the services of the same MARFCAT application but via a MARFL program to classify vulnerable source code based on CVE. Since the design of MARFCAT is general enough to handle all kinds of pattern recognition tasks, our solution allows a user to just script any training or classification tasks in an *intensional* way as opposed to writing new and lengthy JAVA applications.

As shown in Listing 4.1, a MARFL program that can classify exploits and vulnerabilities from CVE based source codes consists of only 20 lines whereas the original MARFCAT JAVA based implementation has well over 1500 lines of code. The dataset specified in the program, `wireshark-1.2.0_train.xml`, consists of 2401 source files that need to classified for various kind of CVE described vulnerabilities [105].

Once the execution is accomplished, the DWT shows classification results as an output for each file in the dataset. A excerpt of the DWT classification results is shown in Appendix A which shows File ID, the configuration used for classification as specified in the MARFL program, as well as the classified subject and its ID.

In order to ensure the consistency of our findings with the original implementation, we compare our result output with the conventional approach of using the MARFCAT application. The comparison showed that our implementation produced a similar output like original implementation.

5.3.2 Speaker Identification

`SpeakerIdentApp` is a text-independent speaker identification application, based on MARF's API and its implementation. As the name suggests, it can be used for identification of speakers' as of who they are, their gender, and accent through machine learning. Based on the pattern recognition pipeline algorithms available in MARF for this task, we have scripted a MARFL program as shown in Listing 5.3.2. As described in our first motivational scenario (Section 1.2.1), a researcher might need to identify speakers to build a language learning application. MARFL allows them to script these tasks without any prior knowledge about MARF or the `SpeakerIdentApp` itself. The researcher can also perform comparative studies by using different configurations to find the best fit for their use case. The program's current training data comprises 32 unique speakers, each of whom contributed voice samples in the `.wav` format. This results in a total of 320 audio files available for training purposes.

```
train "speaker_ident_train.xml" @ [ mls, sl, pr, fe, cl ]
where
    ml_subsystem        mls = MARF;
    sampleLoader        sl = WAV [ channels : 2, bitrate : 16]
                            where
                                dimension channels, bitrate;
                            end;
    preprocessing       pr = FFT_LOW_PASS_FILTER [ windowsize: 2048 ]
                            where
                                dimension windowsize;
                            end;
    featureExtraction   fe = FFT [ poles: 40 ]
                            where
                                dimension poles;
                            end;
    classification      cl = CHEBYSHEV_DISTANCE [ r : 6 ]
                            where
                                dimension r;
                            end;
end
```

The MARF application, `SpeakerIdentApp`, the source code for which can be found at https://sourceforge.net/projects/marf/files/MARF/MARF%20Framework, consists of over 1100 lines of code whereas the same the speaker identification task can be performed using a MARFL script with only 20 lines of code. The resulting machine learning model obtained from executing the MARFL program shown in Listing 5.3.2 can be used to identify speakers from the testing portion of the same dataset. An excerpt of results from this classification task is shown in Appendix B. To verify our obtained results, we compared our output with the classification results for the same configuration as in [22].

5.3.3 Detection and Classification of Malware in Network Traffic

Machine learning algorithms can be used to detect and classify malware in network traffic by training models on known malware pcap data. Mokhov created an extension of MARFCAT called MARFPCAT (*MARF-based PCap Analysis Tool*) [106], which specifically analyzed pcap traces in the given network traffic data. This application can be used as a evidence feed tool for network forensics related investigations about malware and scanning. Most of the ideas in MARFCAT are still applicable here where the same approach was used to machine-learn, detect, and classify vulnerable or weak code fast and with relatively high precision. For our third experiment, we provide a MARFL program that can use a previously trained model to classify these malware in given pcap data using MARF as its machine learning sub-system. We use 800 pcap data files to train a machine learning model using our MARFL program.

The original JAVA implementation of MARFPCAT is about 1500+ lines of code whereas we can perform the same classification or training task using a MARFL script (shown in Listing 5.3.3) in less than 20 lines of code.

```
classify "marfpcat_test.xml" @ [ mls, sl, pr, fe, cl ]
where
    ml_subsystem        mls = MARF;
    sampleLoader        sl = XML [ encoding : utf-8 ]
```

```
                            where
                                dimension encoding;
                            end;
    preprocessing       pr = FFT_LOW_PASS_FILTER [ cutoff: 2024, windowsize: 2048 ]
                            where
                                dimension cutofff, windowsize;
                            end;
    featureExtraction   fe = MINMAX [ poles: 40 ]
                            where
                                dimension poles;
                            end;
    classification      cl = COSINE_SIMILARITY_MEASURE [ r : 6 ]
                            where
                                dimension r;
                            end;
end
```

5.4 Evaluation of MARFL Programs on a Distributed Architecture

In this section, we aim to evaluate a MARFL program on a distributed architecture. To conduct this evaluation, we have chosen a specific experiment from Section 5.3, focusing on the pattern recognition task of classifying vulnerable source code from the CVE dataset (Section 5.3.1).

For this evaluation, we will replicate the experiment using the distributed computation capabilities provided by the GIPSY platform. As per our objective O5, our solution should enable users to perform complex pattern recognition computations on a distributed architecture without requiring any additional code modifications in their scripts. The environment specifications for this evaluation were described in Section 5.2.1.

To begin, we will first allocate additional GIPSY nodes on a single machine. Currently, the GIPSY architecture allows the allocation of only one DWT per GIPSY node, but we can allocate an unlimited number of GIPSY nodes on a system. The same MARFL program shown in Listing 4.1 will be used for this evaluation. We will

vary the number of DWTs available and compare the execution times for the pattern recognition task. All these extra workers allow concurrent computing while on same physical machine and distributed computation when allocated on different machines.

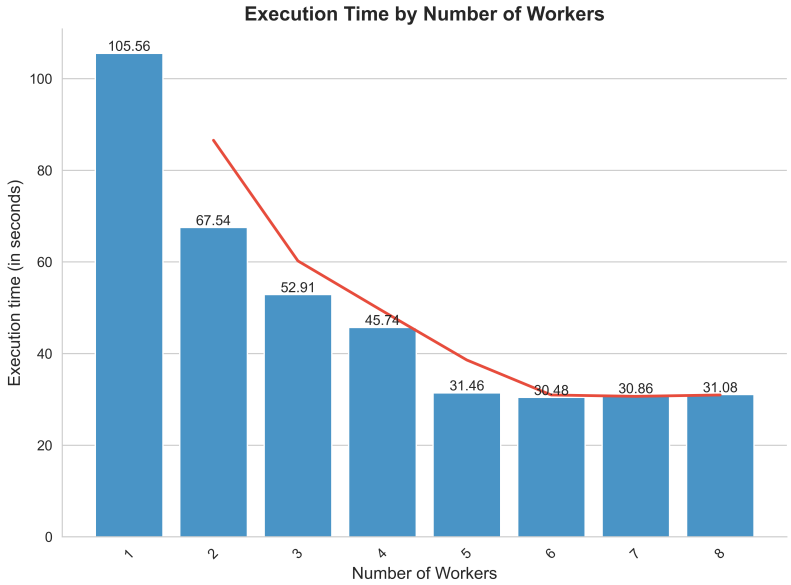

Figure 27: Execution time by number of workers

5.4.1 Results for Single Instance Execution

1. Figure 27 illustrates the total execution time for each available number of workers on GIPSY nodes. As depicted in the figure, the execution time decreases almost by 36% just by the addition of another worker. All the timings mentioned include the time required for GIPSY node registration and specific tier allocation.

2. For the same pattern recognition task, the original implementation of the MARFCAT app takes 78.23s to execute without any distributed computations. However, since MARFL can inherently execute on a distributed architecture, addition of just one more worker node results in faster execution as compared to non-distributed original implementation.

3. We also notice that for this specific task, the execution time begins to plateau after five allocated DWTs, indicating that we do not require more than five

workers. It is important to note that the number of workers needed may vary depending on the complexity and size of the dataset used in the selected pattern recognition task.

4. We observe that as we increase the number of available workers, there is a very slight increase in the total execution time. This slight increase can be attributed to the allocation time required for these additional workers. However, the impact on the overall execution time is negligible.

Followed by, we performed this experiment again but this time the workers were allocated on six different Amazon EC2 instances as demonstrated in Figure 25. Since we implemented a Network File System (NFS), each of the EC2 instance will have access to the same training dataset. NFS also helps us in keeping track of demands in the DST over all the instances, since the configuration file of our Jini middleware DST (`RegDSTTA.config`) is accessible to all machines.

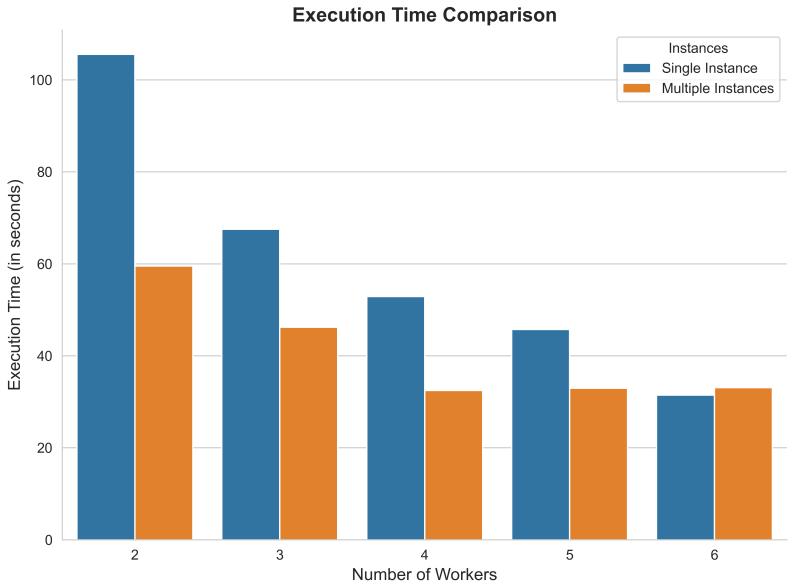

Figure 28: Execution time comparison: single instance vs. multiple instances

5.4.2 Results for Multiple Instance Execution

1. Figure 28 presents the total execution time for different numbers of DWTs, each of which are deployed across an EC2 instance. It provides a comparison between

the execution times of DWTs on a single instance versus multiple instances. The figure clearly demonstrates that when each DWT has dedicated processing power on its own instance, the execution time of the pattern recognition task is further reduced making our solution even faster.

2. We clearly demonstrate that employing two workers in the DWT, each with their own dedicated processing power, yields better executions times compared to workers running on a single instance sharing computation resources. As with the increase in workers on single instance, we notice that our executions times start to flatten when we allocate four workers in DWT, surpassing the performance achieved with five workers running on a single instance.

3. However, it is important to note that as we increase the number of workers and corresponding instances, there is a trade-off to consider. The communication overhead between distinct instances introduces a slight increase in execution time compared to workers running on the same instance. While the performance benefits of using distinct instances for each worker are evident, it becomes particularly justified when dealing with larger datasets and more complex algorithms. In such scenarios, the computational requirements and resource demands are higher.

4. In summary, we showcase the advantage of distributing the workload across distinct multiple worker machines. By leveraging the processing capabilities of individual instances, the parallel execution of DWTs significantly enhances the overall performance for the pattern recognition task.

With these results, we have successfully achieved our Objective O5, demonstrating that users can leverage the distributed computation capabilities of the GIPSY platform without the need to add any specific code to their MARFL scripts. Overall, this evaluation highlights the effectiveness of the GIPSY platform's distributed architecture in improving the execution time of pattern recognition tasks, showcasing its potential for enhancing the performance of MARFL programs.

5.5 Summary

In this chapter, we start by describing our evaluation methodology, which serves as an objective means of verifying the completion of the objectives outlined in Section 1.5. Following this, we provide the hardware specifications and brief design details of our evaluation environment. Then, we provide the MARFL scripts for three separate pattern recognition tasks which were originally implemented in JAVA. We also present the corresponding execution results for these tasks. Subsequently, we evaluate the execution times for one of our tasks by varying the allocated DWTs, aligning with our objective of demonstrating that MARFL programs can be executed on a distributed architecture without necessitating additional coding from the user, thanks to the capabilities of the GIPSY platform. Moving forward, the next chapter will present our concluding remarks. We will also outline the limitations that we encountered during our research and discuss potential future work in this field.

Chapter 6

Conclusion and Future Work

In Chapter 1, we introduced the problem for this research work and provided related motivational scenarios in Section 1.2. We also described our objectives for this dissertation in Section 1.5. Chapter 2 presented the background of our research which included detailed introduction to the Lucid family of intensional languages and the evaluation platform used. In Chapter 3 we present our new Lucid dialect introduced for the fulfillment of our objectives. We go in detail about the syntax and semantics of our language, providing definitions wherever required for newly introduced operators or rules. Chapter 4 focuses on design and implementation of our solution inside the selected GIPSY platform. We provide overview on evaluation flow of a MARFL program from compilation to execution phase. Chapter 5 presented the results of various experiments conducted to evaluate the fulfillment of our objectives. Eventually, in this chapter, we start by describing the our conclusion to our dissertation in Section 6.1, followed by some limitations and potential future work to be considered in Section 6.2.

6.1 Conclusion

In this section, we conclude our research based on the conducted experiments and the results that we achieved in Chapter 5. Below we address the fulfillment of the objectives that we stated in Chapter 1.

- In our research, we devised a solution which enables researchers to perform complex pattern recognition tasks through an intensional scripting approach without requiring knowledge of any specific machine learning framework. We built upon the syntax and semantics of this language and used the JavaCC tool to develop a compiler that can effectively compile these scripts. As a result, our solution achieves the goals outlined in Objective O1 and O2, providing the expected Abstract Syntax Tree and an intermediate representation for these tasks. This approach streamlines the pattern recognition process, making it more accessible to researchers without expertise in machine learning frameworks or coding in general, and ultimately provides them with a feasible solution.

- We introduced a dedicated module within the GIPSY architecture specifically designed to execute MARFL programs. This module includes a MARFL-specific DGT responsible for loading and interpreting the intermediate object produced by compiling a MARFL program. This interpretation is handled by the MARFL interpreter that is designed to traverse the annotated AST, thus enabling it to generate demand for the relevant machine learning sub-system in an *eductive* manner. Additionally, the interpreter sets the evaluation context for each stage of the pattern recognition pipeline using a configuration string that is passed to the relevant machine learning sub-system. These additions to the architecture ensure that it can effectively execute MARFL programs, fulfilling Objective O3.

- We successfully executed three different MARFL programs presented in Chapter 5. To achieve this, we utilized the MARFCAT DGT and DWT modules, both of which are available on the GIPSY platform. By using our solution, we were able to create concise and intensional scripts that efficiently accomplished complex pattern recognition tasks. Our approach eliminates the need for lengthy lines of code and enables researchers to obtain training models and classification result sets with ease. Overall, the successful execution of these MARFL programs confirms the fulfillment of our research Objective O4.

- By integrating our solution inside the GIPSY platform, we not only take advantage of the advanced GIPC and GEE framework but also the distributed architecture. By leveraging these capabilities, we were able to distribute our pattern recognition tasks efficiently without significant user intervention. Unlike complex coding requirements in frameworks such as TensorFlow or PyTorch, our approach enables researchers to initiate training or classification processes in a distributed manner with ease. Through our results in Section 5.4, we successfully demonstrated that our solution can execute pattern recognition tasks on a distributed platform without the need for additional code, fulfilling Objective O5. This makes our devised solution scalable to increasing demand particularly in the context of large and complex datasets.

6.2 Limitations and Future Work

While our research has made significant strides in addressing the problem at hand, and has successfully achieved the objectives outlined in Chapter 1, there are still some limitations that need to be addressed. Most of these limitations can be overcome with potential future work on this solution. Some of those limitations and future works are listed below:

- Our current solution works with a wide variety of algorithms for each pattern recognition stage but it is still limited by the algorithms available inside the sub-system package used. Moreover, addition of any new algorithms or machine learning sub-system would require the developer to update the syntax and provide a new compiler. Future work could focus on exploring ways to improve the flexibility and adaptability of our approach to handle new and emerging algorithms and machine learning sub-systems. This could involve investigating techniques for delegating algorithm selection to the execution engine, as well as exploring ways to incorporate external libraries and frameworks into our solution.

- Currently we only support MARF as our potential machine learning sub-system. While our syntax implementation allows for the use of other sub-systems, such as TensorFlow, the lack of sub-system-specific modules within GIPSY presents a challenge for researchers looking to work with other systems. To address this limitation, future work could focus on implementing sub-system-specific modules for other popular frameworks and integrating them into the GIPSY, particularly sub-system specific DGT and DWT like already present for MARFCAT.

- At present, our presented solution is capable of only executing pattern recognition tasks for a single configuration. MARFL syntax allows users to script tasks for multiple configurations of the pattern recognition pipeline within the same program. Unfortunately, our current execution engine is not equipped to handle evaluations involving multiple configurations. To address this limitation, future work involves updating the execution engine to enable users to execute a single MARFL program that can perform a specific task for multiple machine learning configurations. This would facilitate the users to obtain a comparative analysis between different pipeline configurations. By executing the same task using various configurations, users can assess the performance of different approaches and select the optimal configuration for their specific use case.

- While our approach successfully enabled the execution of MARFL programs on a distributed architecture without requiring any additional code from the user, there is still room for improvement in the allocation and management of GIPSY nodes. Currently, the user can run executable scripts which can somewhat automate the allocation of appropriate DST, DGT and DWT on GIPSY nodes. However, in future we can look into completely automating this node allocation process so the user only has to run the MARFL program and wait for the results. Future work could focus on automatically allocating and managing GIPSY nodes, potentially enabling users to simply run a MARFL

program and wait for the results. This could involve investigating techniques for dynamically scaling the number of nodes based on the size and complexity of the dataset or developing new approaches for optimizing node allocation and utilization.

- In our research, we only focused on executing pattern recognition tasks using MARFL on a distributed computation but we did not extensively test the fault tolerance of the GIPSY platform. As a result, there may be scenarios where GIPSY nodes fail during mid-computation, resulting in incomplete or inaccurate results. To address this limitation, future work could focus on exploring techniques for improving the fault tolerance of the evaluation platform. One potential solution involves leveraging Kubernetes, as explored by Zahraei in his master's thesis [107]. By integrating Kubernetes and other fault-tolerance solutions, we can improve the management and utilization of GIPSY nodes for computation, potentially minimizing the impact of node failures and improving the overall accuracy and reliability of our approach.

6.3 Summary

In this chapter, we concluded our research by providing the list of achieved experiments and the results that we conducted in this research in Section 6.1. Finally, we talk about the limitations that affected our research findings followed by future work for this research in Section 6.2. We consider the broader implications of our work and provide a clear rationale for why this avenue of research is valuable.

Bibliography

[1] J. Paquet, "Scientific intensional programming," Ph.D. dissertation, Department of Computer Science, Quebec City, Canada, 1999.

[2] K. Wan, "Lucx: Lucid enriched with context," Ph.D. dissertation, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2006.

[3] X. Tong, "Design and implementation of context calculus in the GIPSY," Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, Apr. 2008, http://spectrum.library.concordia.ca/975704/.

[4] S. A. Mokhov, J. Paquet, and M. Debbabi, "Designing Forensic Lucid – an intensional specification language for automated cyberforensic reasoning," 2014, being prepared for submission to Journal of Digital Forensics, Security, and Law (JDFSL).

[5] S. A. Mokhov, "Towards hybrid intensional programming with JLucid, Objective Lucid, and General Imperative Compiler Framework in the GIPSY," Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, Oct. 2005, ISBN 0494102934; online at http://arxiv.org/abs/0907.2640.

[6] ——, "Towards syntax and semantics of hierarchical contexts in multimedia processing applications using MARFL," in *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC).*

Turku, Finland: IEEE Computer Society, Jul. 2008. doi: 10.1109/COMPSAC.2008.206. ISSN 0730-3157 pp. 1288–1294.

[7] S. A. Mokhov and M. Debbabi, "File type analysis using signal processing techniques and machine learning vs. `file` unix utility for forensic analysis," in *Proceedings of the IT Incident Management and IT Forensics (IMF'08)*, ser. LNI140, O. Goebel, S. Frings, D. Guenther, J. Nedon, and D. Schadt, Eds. GI, Sep. 2008. ISBN 978-3-88579-234-5. ISSN 1617-5468 pp. 73–85.

[8] S. A. Mokhov, J. Paquet, and X. Tong, "A type system for hybrid intensional-imperative programming support in GIPSY," in *Proceedings of the Canadian Conference on Computer Science and Software Engineering (C3S2E'09)*. New York, NY, USA: ACM, May 2009. doi: 10.1145/1557626.1557642. ISBN 978-1-60558-401-0 pp. 101–107.

[9] J. Paquet, "Distributed eductive execution of hybrid intensional programs," in *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC'09)*. IEEE Computer Society, Jul. 2009. doi: 10.1109/COMPSAC.2009.137. ISBN 978-0-7695-3726-9. ISSN 0730-3157 pp. 218–224.

[10] B. Han, S. A. Mokhov, and J. Paquet, "Advances in the design and implementation of a multi-tier architecture in the GIPSY environment with Java," in *Proceedings of the 8th IEEE/ACIS International Conference on Software Engineering Research, Management and Applications (SERA 2010)*. IEEE Computer Society, May 2010. doi: 10.1109/SERA.2010.40. ISBN 978-0-7695-4075-7 pp. 259–266, online at http://arxiv.org/abs/0906.4837.

[11] D. Dowty, R. Wall, and S. Peters, *Introduction to Montague Semantics*. Dordrecht, The Netherlands: D. Reidel, 1981.

[12] R. Carnap, *Meaning and Necessity: a Study in Semantics and Modal Logic*. University of Chicago Press, Chicago, USA, 1947.

[13] W. W. Wadge and E. A. Ashcroft, *Lucid, the Dataflow Programming Language*. London: Academic Press, 1985.

[14] E. A. Ashcroft, A. A. Faustini, R. Jagannathan, and W. W. Wadge, *Multidimensional Programming*. London: Oxford University Press, Feb. 1995, ISBN: 978-0195075977.

[15] W. W. Wadge, "Possible WOOrlds," in *Intensional Programming I*, M. A. Orgun and E. A. Ashcroft, Eds., vol. Intensional Programming I. World Scientific, May 1995, pp. 56–62, invited Contribution.

[16] W. W. Wadge and A. Yoder, "The Possible-World Wide Web," in *Intensional Programming I*, M. A. Orgun and E. A. Ashcroft, Eds., vol. Intensional Programming I. World Scientific, May 1995, pp. 207–213.

[17] B. Mancilla and J. Plaice, "Possible worlds versioning," *Mathematics in Computer Science*, vol. 2, no. 1, pp. 63–83, 2008. doi: 10.1007/s11786-008-0044-8

[18] S. Mokhov, I. Clement, S. Sinclair, and D. Nicolacopoulos, "Modular Audio Recognition Framework," Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2002–2003, project report, http://marf.sf.net, last viewed April 2012.

[19] The GIPSY Research and Development Group, "The General Intensional Programming System (GIPSY) project," Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2002–2014, http://newton.cs.concordia.ca/~gipsy/, last viewed April 2014.

[20] J. Paquet and P. G. Kropf, "The GIPSY architecture," in *Proceedings of Distributed Computing on the Web*, ser. Lecture Notes in Computer Science, P. G. Kropf, G. Babin, J. Plaice, and H. Unger, Eds., vol. 1830. Springer Berlin Heidelberg, 2000. doi: 10.1007/3-540-45111-0_17 pp. 144–153.

[21] J. Paquet, S. A. Mokhov, E. I. Vassev, X. Tong, Y. Ji, A. H. Pourteymour, K. Wan, A. Wu, S. Rabah, B. Han, B. Lu, L. Tao, Y. Ding, C. L. Ren, and The GIPSY Research and Development Group, "The General Intensional Programming System (GIPSY) project," Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2002–2014, http://newton.cs.concordia.ca/~gipsy/, last viewed January 2014.

[22] S. A. Mokhov, "Study of best algorithm combinations for speech processing tasks in machine learning using median vs. mean clusters in MARF," in *Proceedings of C3S2E'08*, B. C. Desai, Ed. Montreal, Quebec, Canada: ACM, May 2008. doi: 10.1145/1370256.1370262. ISBN 978-1-60558-101-9 pp. 29–43.

[23] S. A. Mokhov, J. Paquet, M. Debbabi, and Y. Sun, "MARFCAT: A fast approach to static vulnerable/weak code analysis and classification," [online], 2014, being prepared for submission to Journals of Computers and Security, Elsevier; online pre-print at http://arxiv.org/abs/1207.3718.

[24] S. A. Mokhov and J. Paquet, "Using the General Intensional Programming System (GIPSY) for evaluation of higher-order intensional logic (HOIL) expressions," in *Proceedings of the 8th IEEE / ACIS International Conference on Software Engineering Research, Management and Applications (SERA 2010)*. IEEE Computer Society, May 2010. doi: 10.1109/SERA.2010.23. ISBN 978-0-7695-4075-7 pp. 101–109, pre-print at http://arxiv.org/abs/0906.3911.

[25] ——, "A type system for higher-order intensional logic support for variable bindings in hybrid intensional-imperative programs in GIPSY," 2014, being prepared for submission to the Transactions on Programming Languages and Systems (TOPLAS).

[26] S. A. Mokhov, "Intensional cyberforensics," Ph.D. dissertation, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, Sep. 2013, online at http://arxiv.org/abs/1312.0466.

[27] C. L. Ren, "Parsing and abstract syntax tree generation in the GIPSY system," Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2002, http://spectrum.library.concordia.ca/1859/.

[28] A. H. Wu, "Semantic analysis and SIPL AST translator generation in the GIPSY," Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2002, http://spectrum.library.concordia.ca/2018/.

[29] K. Wan, V. Alagar, and J. Paquet, "Lucx: Lucid enriched with context," in *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005)*. CSREA Press, Jun. 2005, pp. 48–14.

[30] E. A. Ashcroft and W. W. Wadge, "Lucid – a formal system for writing and proving programs," *SIAM J. Comput.*, vol. 5, no. 3, 1976.

[31] ——, "Erratum: Lucid – a formal system for writing and proving programs," *SIAM J. Comput.*, vol. 6, no. 1, p. 200, 1977.

[32] ——, "Lucid, a nonprocedural language with iteration," *Communications of the ACM*, vol. 20, no. 7, pp. 519–526, Jul. 1977. doi: 10.1145/359636.359715

[33] S. A. Mokhov, J. Paquet, and M. Debbabi, "Formally specifying operational semantics and language constructs of Forensic Lucid," in *Proceedings of the IT Incident Management and IT Forensics (IMF'08)*, ser. LNI, O. Göbel, S. Frings, D. Günther, J. Nedon, and D. Schadt, Eds., vol. 140. GI, Sep. 2008. ISBN 978-3-88579-234-5. ISSN 1617-5468 pp. 197–216, online at http://subs.emis.de/LNI/Proceedings/Proceedings140/gi-proc-140-014.pdf.

[34] ——, "Towards automated deduction in blackmail case analysis with Forensic Lucid," in *Proceedings of the Huntsville Simulation Conference (HSC'09)*, J. S. Gauthier, Ed. SCS, Oct. 2009. ISBN 978-1-61738-587-2 pp. 326–333, online at http://arxiv.org/abs/0906.0049.

[35] ——, "Reasoning about a simulated printer case investigation with Forensic Lucid," in *Proceedings of Digital Forensics and Cyber Crime (ICDF2C'11)*, ser. LNICST, P. Gladyshev and M. K. Rogers, Eds., no. 0088. Springer Berlin Heidelberg, Oct. 2011. doi: 10.1007/978-3-642-35515-8_23. ISBN 978-3-642-35514-1. ISSN 1867-8211 pp. 282–296, accepted and presented in 2011, appeared in 2012; online at http://arxiv.org/abs/0906.5181.

[36] R. Jagannathan and C. Dodd, "GLU programmer's guide," SRI International, Menlo Park, California, Tech. Rep., 1996.

[37] R. Jagannathan, C. Dodd, and I. Agi, "GLU: A high-level system for granular data-parallel programming," in *Concurrency: Practice and Experience*, vol. 1, 1997, pp. 63–83.

[38] N. S. Papaspyrou and I. T. Kassios, "GLU# embedded in C++: a marriage between multidimensional and object-oriented programming," *Softw., Pract. Exper.*, vol. 34, no. 7, pp. 609–630, 2004. doi: 10.1002/spe.582

[39] S. A. Mokhov and J. Paquet, "Formally specifying and proving operational aspects of Forensic Lucid in Isabelle," Department of Electrical and Computer Engineering, Concordia University, Montreal, Canada, Tech. Rep. 2008-1-Ait Mohamed, Aug. 2008, in Theorem Proving in Higher Order Logics (TPHOLs2008): Emerging Trends Proceedings. Online at: http://users.encs.concordia.ca/~tphols08/TPHOLs2008/ET/76-98.pdf and http://arxiv.org/abs/0904.3789.

[40] T. Cargill, "Deterministic operational semantics of lucid," University of Waterloo, Tech. Rep. CS 76-19, Jun. 1976, https://cs.uwaterloo.ca/research/tr/1976/CS-76-19.pdf.

[41] S. A. Kripke, "A completeness theorem in modal logic," *Journal of Symbolic Logic*, vol. 31, no. 2, pp. 276–277, 1966.

[42] ——, "Semantical considerations on modal logic," *Journal of Symbolic Logic*, vol. 34, no. 3, p. 501, 1969.

[43] C. B. Ostrum, *The Luthid 1.0 Manual.* Department of Computer Science, University of Waterloo, Ontario, Canada, 1981.

[44] A. A. Faustini and W. W. Wadge, "An eductive interpreter for the language Lucid," *SIGPLAN Not.*, vol. 22, no. 7, pp. 86–91, 1987. doi: 10.1145/960114.29659

[45] P. Swoboda, "A formalisation and implementation of distributed intensional programming," Ph.D. dissertation, The University of New South Wales, Sydney, Australia, 2004.

[46] P. Swoboda and W. W. Wadge, "Vmake and ISE general tools for the intensionalization of software systems," in *Intensional Programming II*, M. Gergatsoulis and P. Rondogiannis, Eds., vol. Intensional Programming II. World Scientific, Jun. 1999, pp. 310–320, ISBN: 981-02-4095-3.

[47] P. Swoboda and J. Plaice, "A new approach to distributed context-aware computing," in *Advances in Pervasive Computing*, A. Ferscha, H. Hoertner, and G. Kotsis, Eds. Austrian Computer Society, 2004, ISBN 3-85403-176-9.

[48] S. A. Mokhov, "Towards security hardening of scientific distributed demand-driven and pipelined computing systems," in *Proceedings of the 7th International Symposium on Parallel and Distributed Computing (ISPDC'08)*. IEEE Computer Society, Jul. 2008. doi: 10.1109/ISPDC.2008.52. ISBN 978-0-7695-3472-5 pp. 375–382.

[49] L. Verdoscia, "ALFA fine grain dataflow machine," in *Intensional Programming I*, M. A. Orgun and E. A. Ashcroft, Eds., vol. Intensional Programming I. World Scientific, May 1995, pp. 110–134.

[50] J. Paquet and S. A. Mokhov, "Furthering baseline core Lucid," [online], 2011–2014, http://arxiv.org/abs/1107.0940.

[51] J. Paquet, S. A. Mokhov, and X. Tong, "Design and implementation of context calculus in the GIPSY environment," in *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC)*. IEEE Computer Society, Jul. 2008. doi: 10.1109/COMPSAC.2008.200. ISSN 0730-3157 pp. 1278–1283.

[52] P. Grogono, S. Mokhov, and J. Paquet, "Towards JLucid, Lucid with embedded Java functions in the GIPSY," in *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005)*. CSREA Press, Jun. 2005. ISBN 1-932415-75-0 pp. 15–21.

[53] A. Wu, J. Paquet, and S. A. Mokhov, "Object-oriented intensional programming: Intensional Java/Lucid classes," in *Proceedings of the 8th IEEE/ACIS International Conference on Software Engineering Research, Management and Applications (SERA 2010)*. IEEE Computer Society, May 2010. doi: 10.1109/SERA.2010.29. ISBN 978-0-7695-4075-7 pp. 158–167, preprint at: http://arxiv.org/abs/0909.0764.

[54] S. Mokhov and J. Paquet, "Objective Lucid – first step in object-oriented intensional programming in the GIPSY," in *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005)*. CSREA Press, Jun. 2005. ISBN 1-932415-75-0 pp. 22–28.

[55] A. H. Wu, "OO-IP hybrid language design and a framework approach to the GIPC," Ph.D. dissertation, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2009, http://spectrum.library.concordia.ca/976610/.

[56] M. A. Orgun and E. A. Ashcroft, Eds., *Proceedings of ISLIP'95*, vol. Intensional Programming I. World Scientific, May 1995, ISBN: 981-02-2400-1.

[57] M. Gergatsoulis and P. Rondogiannis, Eds., *Proceedings of ISLIP'99*, vol. Intensional Programming II. World Scientific, Jun. 1999, ISBN: 981-02-4095-3.

[58] The MARF Research and Development Group, "The Modular Audio Recognition Framework and its Applications," [online], 2002–2014, http://marf.sf.net and http://arxiv.org/abs/0905.1235, last viewed May 2015.

[59] S. A. Mokhov, "Choosing best algorithm combinations for speech processing tasks in machine learning using MARF," in *Proceedings of the 21st Canadian AI'08*, ser. LNAI 5032, S. Bergler, Ed. Berlin Heidelberg: Springer-Verlag, May 2008. doi: 10.1007/978-3-540-68825-9_21 pp. 216–221.

[60] ——, "On design and implementation of distributed modular audio recognition framework: Requirements and specification design document," [online], Aug. 2006, project report, http://arxiv.org/abs/0905.2459, last viewed April 2012.

[61] S. A. Mokhov, L. W. Huynh, and J. Li, "Managing distributed MARF with SNMP," Concordia Institute for Information Systems Engineering, Concordia University, Montreal, Canada, Apr. 2007, project report. Hosted at http://marf.sf.net and http://arxiv.org/abs/0906.0065, last viewed February 2011.

[62] S. A. Mokhov and Y. Sun, "OCT segmentation survey and summary reviews and a novel 3D segmentation algorithm and a proof of concept implementation," [online], 2011–2014, online at http://arxiv.org/abs/1204.6725.

[63] S. A. Mokhov, "MARFCAT – MARF-based Code Analysis Tool," Published electronically within the MARF project, http://sourceforge.net/projects/marf/files/Applications/MARFCAT/, 2010–2015, last viewed February 2014.

[64] S. A. Mokhov, S. Sinclair, I. Clement, D. Nicolacopoulos, and the MARF Research & Development Group, "SpeakerIdentApp – Text-Independent Speaker Identification Application," Published electronically within the MARF project, http://marf.sf.net, 2002–2014, last viewed February 2010.

[65] S. A. Mokhov, E. Vassev, J. Paquet, and M. Debbabi, "Towards a self-forensics property in the ASSL toolset," in *Proceedings of the Third C* Conference on Computer Science and Software Engineering (C3S2E'10)*. New York, NY, USA:

ACM, May 2010. doi: 10.1145/1822327.1822342. ISBN 978-1-60558-901-5 pp. 108–113.

[66] S. A. Mokhov, "The use of machine learning with signal- and NLP processing of source code to fingerprint, detect, and classify vulnerabilities and weaknesses with MARFCAT," [online], pp. 49–72, Oct. 2010, online at http://arxiv.org/abs/1010.2511.

[67] V. Okun, A. Delaitre, P. E. Black, and NIST SAMATE, "Static Analysis Tool Exposition (SATE) 2010," [online], 2010, see http://samate.nist.gov/SATE2010Workshop.html.

[68] B. Han, "Towards a multi-tier runtime system for GIPSY," Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2010.

[69] Y. Ji, "Scalability evaluation of the GIPSY runtime system," Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, Mar. 2011, http://spectrum.library.concordia.ca/7152/.

[70] E. I. Vassev, "General architecture for demand migration in the GIPSY demand-driven execution engine," Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, Jun. 2005, http://spectrum.library.concordia.ca/8681/.

[71] S. A. Mokhov, J. Paquet, M. Debbabi, and Y. Sun, "MARFCAT: Transitioning to binary and larger data sets of SATE IV," [online], May 2012–2014, online at http://arxiv.org/abs/1207.3718.

[72] B. Lu, "Developing the distributed component of a framework for processing intensional programming languages," Ph.D. dissertation, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, Mar. 2004.

[73] J. Paquet, A. Wu, and P. Grogono, "Towards a framework for the General Intensional Programming Compiler in the GIPSY," in *Proceedings of the 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004)*. New York, NY, USA: ACM, Oct. 2004. doi: 10.1145/1028664.1028731. ISBN 1-58113-833-4 pp. 164–165.

[74] A. H. Wu and J. Paquet, "Object-oriented intensional programming in the GIPSY: Preliminary investigations," in *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005)*. CSREA Press, Jun. 2005. ISBN 1-932415-75-0 pp. 43–47.

[75] J. Paquet and A. H. Wu, "GIPSY – a platform for the investigation on intensional programming languages," in *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005)*. CSREA Press, Jun. 2005. ISBN 1-932415-75-0 pp. 8–14.

[76] E. Vassev and J. Paquet, "A generic framework for migrating demands in the GIPSY's demand-driven execution engine," in *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005)*. CSREA Press, Jun. 2005. ISBN 1-932415-75-0 pp. 29–35.

[77] J. Plaice, B. Mancilla, G. Ditu, and W. W. Wadge, "Sequential demand-driven evaluation of eager TransLucid," in *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC)*. Turku, Finland: IEEE Computer Society, Jul. 2008. doi: 10.1109/COMPSAC.2008.191. ISSN 0730-3157 pp. 1266–1271.

[78] T. Rahilly and J. Plaice, "A multithreaded implementation for TransLucid," in *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC)*. Turku, Finland: IEEE Computer Society, Jul. 2008. doi: 10.1109/COMPSAC.2008.191. ISSN 0730-3157 pp. 1272–1277.

[79] S. A. Mokhov and J. Paquet, "A type system for higher-order intensional logic support for variable bindings in hybrid intensional-imperative programs in GIPSY," in *9th IEEE/ACIS International Conference on Computer and Information Science, IEEE/ACIS ICIS 2010*, T. Matsuo, N. Ishii, and R. Lee, Eds. IEEE Computer Society, May 2010. doi: 10.1109/ICIS.2010.156. ISBN 978-0-7695-4147-1 pp. 921–928, presented at SERA 2010; pre-print at http://arxiv.org/abs/0906.3919.

[80] S. A. Mokhov, *Hybrid Intensional Computing in GIPSY: JLucid, Objective Lucid and GICF*. LAP - Lambert Academic Publishing, Mar. 2010, ISBN 978-3-8383-1198-2.

[81] P. Grogono, "Intensional programming in Onyx," Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, Tech. Rep., Apr. 2004.

[82] S. Mokhov and J. Paquet, "General imperative compiler framework within the GIPSY," in *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005)*. CSREA Press, Jun. 2005. ISBN 1-932415-75-0 pp. 36–42.

[83] A. H. Pourteymour, "Comparative study of Demand Migration Framework implementation using JMS and Jini," Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, Sep. 2008, http://spectrum.library.concordia.ca/975918/.

[84] S. Ghosh, *Distributed Systems – An Algorithmic Approach*. CRC Press, 2007, ISBN: 978-1-58488-564-1.

[85] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed Systems: Concepts and Design*, 4th ed. Addison-Wesley, 2005, ISBN: 0-321-26354-5.

[86] S. Rabah, S. A. Mokhov, and J. Paquet, "An interactive graph-based automation assistant: A case study to manage the GIPSY's distributed

multi-tier run-time system," in *Proceedings of the ACM Research in Adaptive and Convergent Systems (RACS 2013)*, C. Y. Suen, A. Aghdam, M. Guo, J. Hong, and E. Nadimi, Eds. New York, NY, USA: ACM, Oct. 2011–2013. doi: 10.1145/2513228.2513286. ISBN 978-1-4503-2348-2 pp. 387–394, pre-print: http://arxiv.org/abs/1212.4123.

[87] N. Foster, M. J. Freedmany, A. Guha, R. Harrisonz, N. P. Kattay, C. Monsantoy, J. Reichy, M. Reitblatt, J. Rexfordy, C. Schlesingery, A. Story, and D. Walkery, "Languages for software-defined networks," in *Proceedings of IEEE COMS'13*. IEEE, 2013.

[88] L. Tao, "Intensional value warehouse and garbage collection in the GIPSY," Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2004, http://spectrum.library.concordia.ca/8129/.

[89] I. Agi, "GLU for multidimensional signal processing," in *ISLIP'95: The 8th International Symposium on Languages for Intensional Programming*, M. A. Orgun and E. A. Ashcroft, Eds., vol. Intensional Programming I. World Scientific, May 1995, ISBN: 981-02-2400-1. [Online]. Available: citeseer.ist.psu.edu/agi95glu.html

[90] W. Stallings, *SNMP, SNMPv2, SNMPv3, and RMON 1 and 2*, 3rd ed. Addison-Wesley, 1999, ISBN: 0-201-48534-6.

[91] D. R. Mauro and K. J. Schmidt, *Essential SNMP*. O'Reilly, 2001, ISBN: 0-596-00020-00.

[92] D. Harrington, R. Presuhn, and B. Wijnen, "RFC 2571: An Architecture for Describing SNMP Management Frameworks," [online], Apr. 1999, http://www.ietf.org/rfc/rfc2571.txt, viewed in January 2008.

[93] Y. Ji, S. A. Mokhov, and J. Paquet, "Unifying and refactoring DMF to support concurrent Jini and JMS DMS in GIPSY," in *Proceedings of the Fifth*

International C Conference on Computer Science and Software Engineering (C3S2E'12)*, B. C. Desai, S. P. Mudur, and E. I. Vassev, Eds. New York, NY, USA: ACM, Jun. 2010–2013. doi: 10.1145/2347583.2347588. ISBN 978-1-4503-1084-0 pp. 36–44, online e-print http://arxiv.org/abs/1012.2860.

[94] G. Fourtounis, P. C. Ölveczky, and N. Papaspyrou, "Formally specifying and analyzing a parallel virtual machine for lazy functional languages using Maude," in *Proceedings of the 5th International Workshop on High-Level Parallel Programming and Applications*, ser. HLPP'11. New York, NY, USA: ACM, 2011. doi: 10.1145/2034751.2034758. ISBN 978-1-4503-0862-5 pp. 19–26.

[95] Y. Ji, S. A. Mokhov, and J. Paquet, "Design for scalability evaluation and configuration management of distributed components in GIPSY," 2010–2013, unpublished.

[96] A. B. Bondi, "Characteristics of scalability and their impact on performance," in *Proceedings of the 2nd international workshop on Software and performance*, 2000. doi: 10.1145/350391.350432 pp. 195–203.

[97] P. Degano and C. Priami, "Enhanced operational semantics: A tool for describing and analyzing concurrent systems," *ACM Computing Surveys*, vol. 33, no. 2, pp. 135–176, 2001.

[98] J. Paquet, "Relational databases as multidimensional dataflow," Master's thesis, Department of Computer Science, Quebec City, Canada, 1995.

[99] M. A. Orgun and W. Du, "Multi-dimensional logic programming: Theoretical foundations," *Theoretical Computer Science*, vol. 185, no. 2, pp. 319–345, 1997. doi: 10.1016/S0304-3975(97)00048-0

[100] A. A. Faustini and R. Jagannathan, "Multidimensional problem solving in Lucid," SRI International, Tech. Rep. SRI-CSL-93-03, 1993.

[101] S. Viswanadha and Contributors, "Java compiler compiler (JavaCC) - the Java parser generator," [online], 2001–2008, https://javacc.dev.java.net/.

[102] K. C. Louden, *Compiler Construction: Principles and Practice*. PWS Publishing Company, 1997, ISBN 0-564-93972-4.

[103] S. L. Garfinkel, "An evaluation of amazon's grid computing services: Ec2, s3 and sqs," Harvard Computer Science Group, Tech. Rep., 2007.

[104] B. B., A. S., B. R., and T. E., "Virtual private cloud," *Pro PowerShell for Amazon Web Services*, Sep. 2019. doi: 10.1007/978-1-4842-4850-8_5

[105] S. A. Mokhov, J. Paquet, and M. Debbabi, "The use of NLP techniques in static code analysis to detect weaknesses and vulnerabilities," in *Canadian AI'14*, ser. LNAI, vol. 8436. Springer, 2014. doi: 10.1007/978-3-319-06483-3_33 pp. 326–332, short paper.

[106] S. A. Mokhov, "MARFPCAT – MARF-based PCap Analysis Tool," Published electronically within the MARF project, 2012–2015, http://sourceforge.net/projects/marf/files/Applications/MARFCAT/.

[107] S. P. Zahraei, "A gipsy runtime system with a kubernetes underlay for the opentdip forensic computing backend," Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2023, .

Appendix A

Classification Results for Wireshark CVE Dataset

```
 File: wireshark-1.2.0/wsutil/unicode-utils.h
              Path ID: 2401
               Config: -noprep -raw -fft -cheb
      Processing time: 0d:0h:0m:0s:22ms:22ms
         Subject's ID: 8
   Subject identified: CVE-2009-2559
ResultSet: [suppressed; enable debug mode to show]
FileItem: strFileID: [2401]
strPath: [wireshark-1.2.0/wsutil/unicode-utils.h]
strFileType: [ASCII C program text]
bEmpty: [false]
oLocations: [[[]]]

       Second Best ID: 2
     Second Best Name: CVE-2010-2284
            Date/time: Tue May 09 18:05:50 GMT 2023
Outcome o (classifier-specific): 1043.9241510316906
             Distance threshold: 0.1
           Computed raw P = 1/o: 0.0
         Warning to be reported: false
          Computed normalized P: 0.0
```

Appendix B

Classification Results for Speaker Identification

```
File: speaker_ident_test/steve-test2.wav
              Path ID: 5
               Config: -noprep -fft -cheb
      Processing time: 0d:0h:0m:0s:3ms:3ms
         Subject's ID: 3
   Subject identified: Steve
ResultSet: [suppressed; enable debug mode to show]
FileItem: strFileID: [5]
strPath: [speaker_ident_test/steve-test2.wav]
strFileType: [Audio WAV file]
bEmpty: [false]
oLocations: [[[<<strLineNumber:0,,,
  oCVEs:[Steve],,,
  oCWEs:[],,,
  strExplanation:[Unset explanation],,,
  strCodeFragment:[Unset code fragment.],,,
  strType:sink (default),,,
  strIgnore:false,,,
  bEmpty:false
/>>
]]]

Expected subject's ID: 3 (possible: [3])
     Expected subject: Steve
       Second Best ID: 4
     Second Best Name: Jimmy
```

```
            Date/time: Wed May 10 00:44:18 GMT 2023
Outcome o (classifier-specific): 66.58248333611064
             Distance threshold: 0.1
           Computed raw P = 1/o: 0.015018965197677684
         Warning to be reported: true
          Computed normalized P: 0.015018965197677684
```