Triton: a Domain Specific Language for Cyber-Physical Systems

by

Bradley Wood

A thesis submitted to the
School of Graduate and Postdoctoral Studies in partial
fulfillment of the requirements for the degree of

Master of Applied Science

in

Electrical and Computer Engineering

Faculty of Engineering and Applied Science

University of Ontario Institute of Technology (Ontario Tech University)

Oshawa, Ontario, Canada

April 2021

© Bradley Wood, 2021

THESIS EXAMINATION INFORMATION

Submitted by: **Bradley Wood**

Master of Applied Science in **Electrical and Computer Engineering**

Thesis title: Triton: a Domain Specific Language for Cyber-Physical Systems

An oral defense of this thesis took place on April 8th, 2021 in front of the following examining committee:

Examining Committee:

Chair of Examining Committee	Dr. Ying Wang
Research Supervisor	Dr. Akramul Azim
Examining Committee Member	Dr. Qusay Mahmoud
Thesis Examiner	Dr. Haoxiang Lang – Ontario Tech University

The above committee determined that the thesis is acceptable in form and content and that a satisfactory knowledge of the field covered by the thesis was demonstrated by the candidate during an oral examination. A signed copy of the Certificate of Approval is available from the School of Graduate and Postdoctoral Studies.

Abstract

The design of cyber-physical systems is non-trivial and often filled with tedious, error-prone tasks that could be represented in a better way. Engineers often work with low-level languages such as C and C++, with real-time operating systems under limited computational resources, which requires extensive domain-specific knowledge. This work proposes Triton, a language focused on increasing abstraction by providing high-level domain-specific features to cyber-physical systems. We propose dedicated code blocks to handle task scheduling, constraint management, and computational offloading at the language level. Triton provides an easy way to offload tasks with bidirectional communication channels to enable continuous streaming of data between the master application and the tasks it offloads. Triton's prototype implementation targets the Java virtual machine (JVM), supporting execution on any platform with an available JVM. Experiments and example code provided shows the effectiveness of the proposed solution when compared with languages traditionally seen in cyber-physical systems development.

Keywords: cyber-physical systems; domain-specific languages; computational offloading; scheduling

Author's Declaration

I hereby declare that this thesis consists of original work of which I have authored. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I authorize the University of Ontario Institute of Technology (Ontario Tech University) to lend this thesis to other institutions or individuals for the purpose of scholarly research. I further authorize University of Ontario Institute of Technology (Ontario Tech University) to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research. I understand that my thesis will be made electronically available to the public.

____ Bradley Wood

Statement of Contributions

Part of the work described in this thesis has been published as:

Wood, B., and Azim, A.Triton: a domain specific language for cyber-physical systems. In IEEE International Conference on Industrial Technology (2021).

Wood, B., Watling, B., Winn, Z., Messiha, D., Mahmoud, Q. H., and Azim, A.Remote method delegation: a platform for grid computing. Journal of Grid Computing 18, 4 (2020), 711–725.

I was responsible for the synthesis of ideas and writing of the manuscript. The other authors contributed to software testing and review of manuscript.

Acknowledgements

I would like to thank my supervisor, Dr. Akramul Azim, for his continued support, financial commitment, and guidance during my studies. I would also like to thank my colleagues in the real-time embedded software lab group for their support, feedback, and ideas.

Contents

Abstract i

Author's Declaration ii

Acknowledgements iv

Contents v

List of Figures vii

List of Tables viii

Listings ix

Abbreviations x

1 Introduction 1
1.1 Current Industrial Challenges 2
1.2 Thesis Objective 3
1.3 Contributions 4
1.4 Thesis Organization 5

2 Background 6
2.1 Domain-Specific Languages 6
2.2 Compilers 7
2.3 Internet of Things 9
2.4 Fog and Edge Computing 9
2.5 Embedded Systems 10

3 Related Work 12
3.1 DSLs in Embedded Systems 12
3.2 Computational Offloading 15
3.3 Remote Procedure Calls 15
3.4 Communication Channels 16

4 Proposed Triton Language Design **17**
4.1 Scheduling 18
4.1.1 Constraint Management 21
4.2 Conditional Computation Offloading 22
4.2.1 Conditional Task Offloading 23
4.2.2 Communication Channels 27
4.3 Grammar 28
4.4 Remote Method Delegation 29
4.4.1 Architecture 30
4.4.2 Code Migration 31
4.4.3 Communication Channels 33
4.4.4 Security 34
4.4.5 Distributed Transparency 35
4.4.6 Failure Modes 35
4.4.7 Configuration 36
4.5 Applications and Use-Cases 37

5 System Evaluation and Experimental Analysis **38**
5.1 System Evaluation 38
5.2 Experimental Setup and Analysis 41
5.2.1 Offloading by CPU Utilization 41
5.2.2 Offloading by Algorithmic Input Value 42

6 Conclusions **46**
6.1 Future Works 48

A Antlr4 Triton Grammar **49**

B Triton Quick Reference Sheet **68**

Bibliography **71**

List of Figures

2.1 Compiler Flowchart 8
2.2 Fog Computing Architecture 10

4.1 Example System Architecture with Triton 19
4.2 3-Tier RMD Architecture 30
4.3 Job Delegation Process with RMD 32
4.4 Example RMD Program with Dependencies 33

5.1 Conditional Computational Offloading Performance 42
5.2 Unconditional Computational Offloading Performance 42
5.3 Local Task Execution Performance 43

List of Tables

3.1 Overview of Related Works 13

4.1 Offloading Metrics 26
4.2 Example Applications and Use-Cases 37

5.1 Recursive Fibonacci execution time (ms) under conditional offloading, unconditional offloading, and local execution. The results are averaged over 10 runs. 45

Listings

4.1 Periodic scheduling example with two tasks 21
4.2 Periodic scheduling example with task constraints 22
4.3 Blocking Delegate with Conditional Offloading 24
4.4 Asynchronous Delegate with Conditional Offloading 24
4.5 Approximate Compiler Output from Listing 4.3 25
4.6 Asynchronous Delegate with Communication Channel 27
5.1 Robot with PID controllers and object detection 40
5.2 Offloading Fibonacci's algorithm by Input Value 44
A.1 Antlr4 Grammar 49

Abbreviations

MQTT Message Queuing Telemetry Transport
DSL Domain Specific Language
CPS Cyber-Physical System
JVM Java Virtual Machine
HTTP Hypertext Transfer Protocol
WCET Worst-case Execution time
RPC Remote Procedure Call
RMI Remote Method Invokation
RMD Remote Method Delegation
QoS Quality of service
S2S Source-to-source
AOT Ahead-of-time
JIT Just-in-time

Chapter 1

Introduction

In recent years, an explosion in demand for IoT, embedded, and cyber-physical systems has been observed; in fact, 95% of computer chips produced are for use in embedded applications [27]. The applications of these devices are very diverse, including many industries such as smart homes, medical devices, real-time and/or safety-critical systems. However, common themes are observed between these subdomains. The hardware is often resource-constrained. Given the speed of advancement in software engineering and artificial intelligence, these devices are not future-proof. Moreover, these devices are increasingly connected to the internet and often reliant on fog or cloud computing resources. Even latency-sensitive real-time applications make use of fog computing to achieve real-time performance [32]. These embedded systems, IoT devices, and other cyber-physical systems may access powerful external computing assets with relatively low latency and cost. The advancement in availability and cost-effectiveness of cloud computing services, including software as a service (SaaS) may result in embedded systems that are increasingly reliant on them.

1.1 Current Industrial Challenges

Several industrial challenges are currently present in the development of IoT, embedded, and cyber-physical systems. Firstly, developers often work in low-level environments in languages like C, with an immense amount of domain-specific knowledge being required. Secondly, there is an over-reliance on external computing resources such as fog and cloud computing. Despite this growing expansion into these industries, software development remains a non-trivial problem on embedded hardware. Finally, changing requirements may pose problems during development. For instance, some software might be required to run on hardware it wasn't originally designed for. This could lead to scenarios where computational offloading is desired depending on the type of hardware it is running on. Of course, developers will be required to support both paradigms.

The low-level programming environments commonly used in IoT and embedded systems are unforgiving and require a high degree of domain-specific knowledge, including manual memory management, scheduling, and operating systems. On the contrary, some high-level level languages lack the ability to interface with the hardware and/or operating system in ways that low-level programming languages can. Of course, many high-level languages can call native code, but this process adds an extra level of unnecessary complexity to the application.

Many engineers supplement resource-constrained hardware by utilizing fog and cloud computing resources, including third-party APIs. However, in a specific set of circumstances, heavy reliance on these systems can result in a loss of service. This could be due to several, including network connectivity, server load, or cyber-attacks. In some cases, it could be more efficient to perform computation locally instead of offloading it to the fog or cloud computing resources.

1.2 Thesis Objective

Many of the issues present in IoT, embedded, and cyber-physical systems stem from the ease of use, clarity, and limited hardware resources. By designing a programming language specific to this domain, *can we increase code comprehension, reduce boilerplate, manage system constraints, and harness the power of fog computing resources?* This thesis paper presents Triton, a domain-specific language (DSL) with novel language features. One of the goals of this work is to design a grammar that sets clear expectations about a program's behavior. To add to this, many IoT, embedded, and cyber-physical systems with limiting computing resources either have limited features or are overly reliant on third-party APIs, cloud resources, etc. The question remains, *can we design and implement a programming language that utilizes fog or cloud computing resources without being totally reliant on them?* Whether a system is dependant on external computing resources or not, *how can developers create a singular piece of create software that behaves the same for either configuration?* Given these design questions, this work aims to tackle the following challenges:

1. To design a clean, expressive grammar that reduces boilerplate by moving features that are traditionally implemented at the API level to the language level.

2. To implement task scheduling at the language level with a constraint management system.

3. To enable location and failure transparency for computational tasks that can be offloaded to external computing resources.

4. To make computational offloading conditional, depending on contextual information.

5. To maintain communications channels between an application and its asynchronous tasks regardless of where they are executed

1.3 Contributions

This thesis's contributions aim to solve several problems prevalent in the industries of IoT, embedded, and cyber-physical systems. The contributions of this thesis can be broken down into the design and implementation of two cohesive pieces of software, Remote Method Delegation [31] (RMD) and the Triton [30] domain-specific language. In summary, the contributions of this thesis can be broken down as follows:

1. The introduction of the Remote Method Delegation platform
    (a) Designed and Implemented code migration, load balancing, and job execution.
    (b) Enabled untrusted code to be run within a secure managed environment.
    (c) Defined expressive grammar rules to enable synchronous and asynchronous computational offloading
    (d) Enabled communications channels to be bound to job requests. This feature enables an application to maintain constant communications with the tasks that it offloads.
2. The Triton domain-specific language
    (a) Developed compiler infrastructure and run-time libraries for the Java virtual machine (JVM).
    (b) Defined an expressive grammar and language features that are platform agnostic. Features are to be abstract enough to be implemented for other

platforms, including LLVM.

(c) Designed and implemented a mechanism for period task scheduling with constraint management.

(d) Enabled conditional computational offloading with communication channels.

(e) Defined and implemented a set of metrics to aid in computational offloading decisions.

1.4 Thesis Organization

This thesis has been organized into six distinct chapters. In Chapter 2, we provide relevant background information, including compiler construction, IoT, fog, and embedded computing. Works related to this thesis are discussed in Chapter 3. We discuss the state-of-the-art, including DSLs for embedded systems, compare them with Triton, and investigate other works related to computational offloading.

Next, in Chapter 4.4 we explore the features provided by our RMD platform and stress its benefits. In Chapter 4, we propose the design of Triton and specify implementation details. Next, we discuss the performance implications of both RMD and Triton's performance implications in Chapter 5. Finally, in Chapter 6, we present our conclusions and discuss future works.

Chapter 2

Background

To provide basic background on the fields and technologies that form the basis of this thesis, we discuss several topics and their applications. In this chapter, we explore each of the following: domain-specific languages, compilers, IoT, fog computing, as well as embedded systems. The types of DSLs will be discussed as well as the different types of compilers. This chapter will also explore the components required to construct a compiler.

2.1 Domain-Specific Languages

A DSL is a language designed for specific use-cases as opposed to a general-purpose language (GPL) which appeal to many problem domains and programming paradigms. DSLs are used in many domains, including web development, game development, query languages, and more. Popular examples including HTML, CSS, VHDL, SQL can be found in many software projects. There are two types of DSLs, the internal DSL and the external DSL [14].

An internal DSL (also called embedded DSLs) are DSLs that are built on top of

an existing language and its infrastructure. The language forming the foundation for the DSL is also referred to as the host language. Kotlin is commonly used as a host language due to its many language features that enable expressive syntax.

External DSLs differ in that they operate standalone, without the need for a host language or other infrastructure. HTML, VHDL, SQL, etc., are examples of external DSLs.

2.2 Compilers

Most compilers are constructed similarly. At their core, they all are programs that take human-readable source code as input and produce an executable program [13]. Of course, there are many different types of compilation strategies, including ahead-of-time (AOT), just-in-time (JIT), source-to-source (S2S), and recompilation. AOT compilers produce optimized machine code instructions from high-level language source code or bytecode. This process occurs prior to run-time. JIT compilers typically work with bytecode instructions, portable to many platforms. The job of a JIT compiler is deferred to run-time, enabling the portability of bytecode whilst achieving near-native speeds [9]. S2S compilers, as the name suggests, compile source code defined in one language into a valid source code of another language. This is commonly observed on the web, with several different languages having compilers that target JavaScript source code.

In Figure, 2.1 we show the numerous stages of a typical compiler. The first step consists of tokenization. In this step, source code is broken into a sequential stream of tokens, including keywords, identifiers, operators, etc. In the parsing phase, groups of tokens are matched to grammar rules, as defined by the programming language specification. It is in this phase where syntax errors are detected. Both lexers and

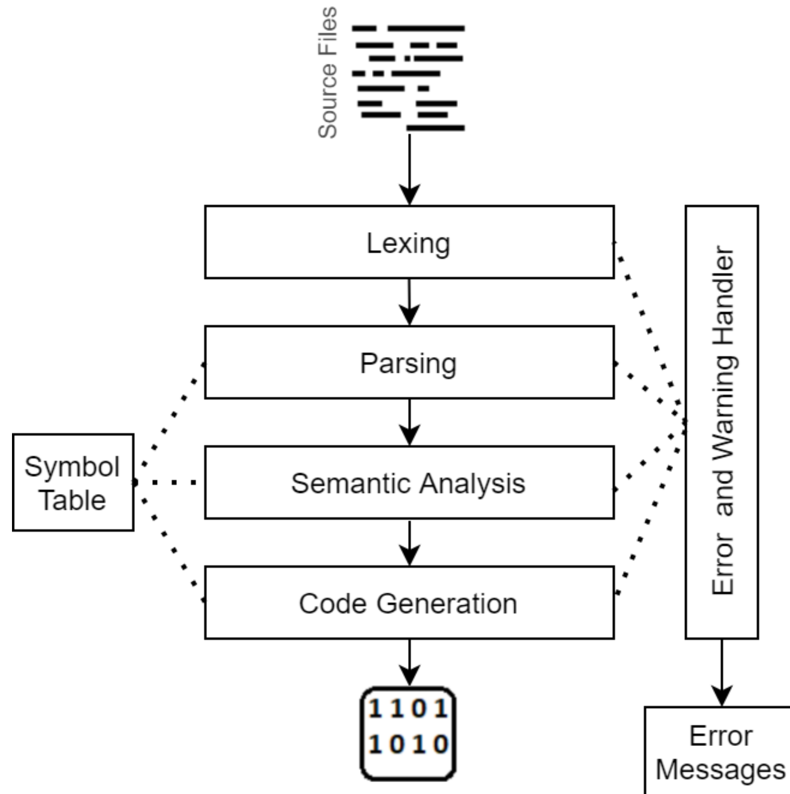

Figure 2.1: Compiler Flowchart

parsers can be written manually by a developer, but many choose to use tools such as Lex & Yacc [20] or ANTLR [23]. Before moving into code generation, the program is checked for semantic correctness. This may include verifying that variables and function calls used in expressions have been defined, type checking, and verifying that return statements are present when necessary. If all previous stages are successful, code generation may begin. This process can vary greatly depending on the target platform and architecture.

2.3 Internet of Things

The market for IoT devices has been steadily growing with increasing access to the internet and the advent of smart home appliances. Embedded devices utilizing internet resources can be considered IoT devices. These devices are usually remotely controllable via smart-phone application. IoT devices exist in many industries, including smart home, environmental monitoring, e-health, transportation, military, and industrial plant monitoring [5]. Privacy and security are commonly cited concerns of consumers and researchers. In fact, a major Botnet called *Mirai* was composed of mainly embedded and IoT devices [21]. The Botnet was designed primarily for the purpose of conducting Distributed Denial of Service (DDoS) attacks. *Mirai* gained entry into many IoT devices by using a table of commonly used (and often default) credentials.

2.4 Fog and Edge Computing

Fog computing is a logical extension to cloud computing, bringing computing resources closer to the edge of the network [32], often to a local area network. This is shown in Figure 2.2. Fog computing may provide some or all of the same features traditionally included in cloud computing services, including computing resources, storage, and network services [7]. By moving computing resources towards the edge of the network, reduced network latency will likely be observed. This adaptation is especially advantageous for resource-constrained systems and time-sensitive applications. Of course, this latency advantage comes with the cost of purchasing hardware and its maintenance. These fees are likely to exceed the cost of renting cloud computing resources, where you will only pay for the resources you utilize.

Similar to Fog Computing, Edge computing also brings computing resources closer to the location in which they are needed. Edge computing is typically done on the

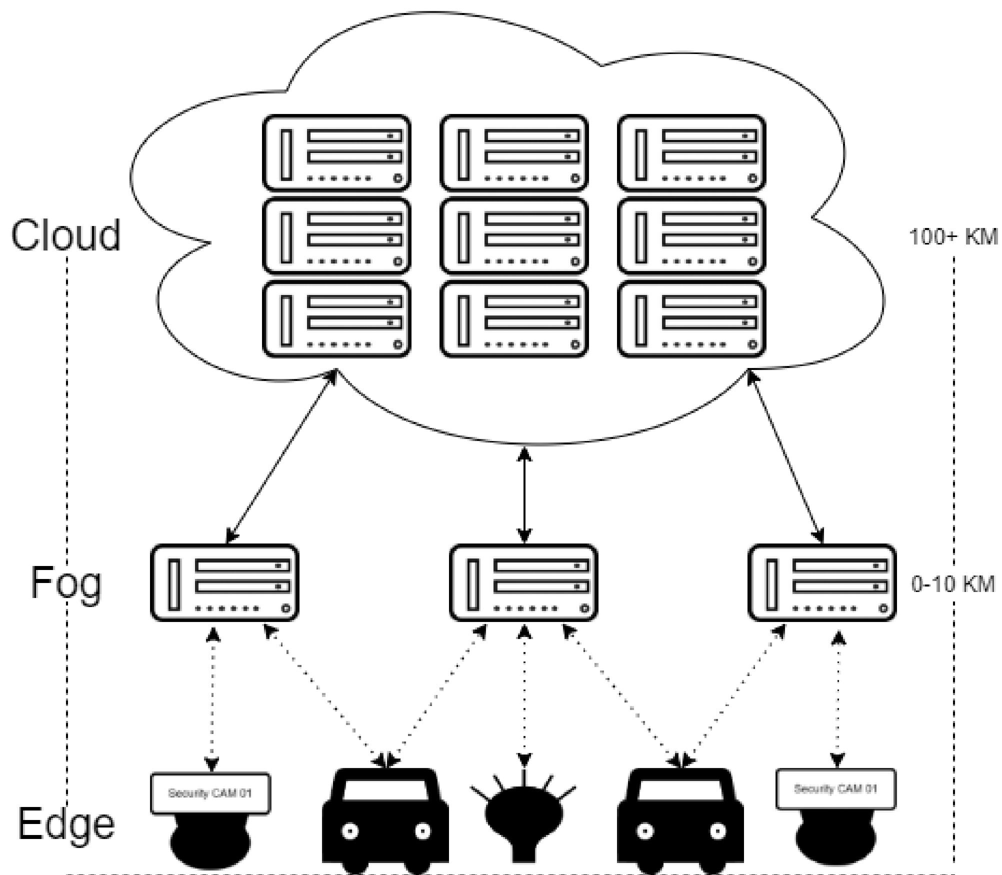

Figure 2.2: Fog Computing Architecture

same physical hardware that requires the computation [11]. This may include IoT devices or systems that are physically connected to sensors.

2.5 Embedded Systems

Embedded systems can be summarized as computer systems, including I/O devices and sensors that serve a dedicated function. IoT devices are examples of embedded systems, but not all embedded systems are IoT devices. Embedded systems do not need to be connected to the internet or any other communications hardware.

Embedded systems are used in many industries, including home appliances, control

systems, cars, and other safety-critical infrastructure. Many of these systems have real-time requirements, meaning that the device must respond to an external event and/or produce results within strict deadlines. Failure to meet real-time requirements can have several consequences, from system failure to reduction in quality of service (QoS). To ensure reliability, engineers perform worst-case execution time (WCET) for all system tasks and perform a schedulability test. This schedulability test ensures that all real-time tasks can be scheduled to complete before their real-time deadline expires.

Chapter 3

Related Work

In this Chapter, research works related to this thesis will be discussed. As this thesis presents Triton, a DSL with novel features targeting IoT, embedded, and cyber-physical systems, a diverse group of works will be discussed and compared. This includes language designs for DSLs targeting embedded systems and computational offloading. Contributions of related works will be compared to this thesis. An overview of the main works described in this section is shown in Table 3.1.

3.1 DSLs in Embedded Systems

Several works have tried to bring DSLs into the embedded development space. Most approaches include embedded DSLs or visual programming with various goals and implementation techniques. Implementation by way of transpiling or as a library (with compiler modification) is often preferred over creating a standalone language with its own compiler or modifying an existing one. Hume [15] is a DSL for real-time embedded systems that explores the expressibility of source code in resource-constrained systems. It provides automatic memory management, polymorphic types, and user-defined data

Table 3.1: Overview of Related Works

Work	Description	Limitations
Go-Lang [1]	General purpose programming language	Supports communication channels but only between threads
Java RMI [28]	Remote Procedure Calls, highly transparent	No code migration or load balancing
UML [25] AADL [10] Ivanova et al.[17]	Visual (diagramming) lang to model system	Can model system requirements, code generation not sufficient
CHARIOT [24]	A textual DSL for CPS	Models resiliency features, no computational offloading
Hume [15]	A textual DSL for resource constrained embedded systems	Limited domain-specific features
CREST [19]	Visual (diagramming) lang for CPS with python DSL	model requirements visually, but not for programming

structures whilst maintaining determinism. Our solution is different from Hume as we aim to focus on the expressibility of scheduling and constraints management in real-time embedded systems and computational offloading. The Ivory and Tower [16] languages are embedded DSLs built on top of Haskel for system programming and embedded systems development. Backends for Tower are provided for FreeRTOS [6] and AADL [10]. The authors reported achieving a dramatic increase in productivity and code quality.

In [17], the authors propose a visual programming approach to embedded systems development. Even the use of UML [25] has been proposed for automatic code generation in embedded systems [22]. Another DSL, CREST [19], models requirements visually through diagrams created with a Python DSL. This approach requires the programmer to define their diagrams in Python and cannot be used to generate code for a cyber-physical system.

Santos et al. [26] propose a high-level DSL to specify run-time adaptations in embedded systems. They evaluate their proposed DSL on a stereo navigation system and show the benefits of dynamically adaptable algorithmic parameters.

DSLs have also been proposed to model the requirements of real-time embedded systems, including their components, goals, and constraints. These DSLs are sometimes called domain-specific modeling languages. AutoModel [18] only requires high-level design requirements to synthesize structural and behaviour models from which existing Model-driven development (MDD) tools can automatically generate an implementation.

In addition to DSLs designed specifically for embedded systems and their modeling, several works were designed specifically for cyber-physical systems. One such DSL, CHARIOT [24], aims to support communication heterogeneity and model resiliency into the language. CHARIOT enforces strict separation-of-concerns between application-specific tasks and communications, thereby allowing for heterogeneous communication middleware such as MQTT, sockets, or HTTP APIs. The resiliency features of CHARIOT attempt to ensure that system goals are completed on time and that failures are detected and mitigated. Triton also implements these resilience features through its task scheduling and constraint management system.

This thesis (Triton) differs from these existing works by providing first-class support for task scheduling, constraint management, offloading computationally intensive workloads, and providing bidirectional communication channels for offloaded tasks. These features are only available in existing languages through API calls, whilst Triton enables them at the language level through highly expressible grammar rules. This approach allows for cleaner code and clarifies the behavior of a program. Visual programming approaches might be easier for inexperienced developers but have clear limitations that cannot be conquered without a more traditional programming language such as Java or C.

3.2 Computational Offloading

The idea of computational offloading is not new. It is a paradigm typically used on hardware that is underpowered for the job. This process involves requesting an external computing resource, such as the cloud, or fog nodes (closer to the edge) to execute a task instead of running it locally. This process may take shape in different forms, including migration of code and remote evaluation, or a Software as a service (SaaS) product such as Amazon Rekognition.

Many research works on the topic of computational offloading are present. The efficiency of offloading to either fog or cloud has been studied in [4] where the authors propose a method of reducing latency and energy consumption by utilizing fog computing resources. Their solution involves utilizing both fog and cloud computing resources where optimization can further prioritize for latency or energy consumption. This is similar to Triton in that computational offloading in Triton may consider both fog and cloud computing resources, depending on the scenario.

3.3 Remote Procedure Calls

Remote Procedure Calls (RPC) is one way to offload computational workloads. This programming paradigm provides the programmer with a high-level of abstraction to invoke code on a remote server [29]. An RPC system provides the developer with a handle to a function that is typically located on another machine for the purposes of invocation. The RPC framework is responsible for managing communications between machines, including sending invocation requests, transmitting and marshaling arguments, unmarshalling, and returning results. While RPC does increase abstraction, significant overhead is added to the system.

RPC systems can also be object-oriented; such systems are often called Remote

Method Invocation (RMI). An RMI client communicates with a server through the use of a stub. The programmer invokes method calls through the stub object, which is responsible for marshaling arguments, sending the invocation request, and unmarshalling and returning results.

Java RMI is one of many implementations of this technique and operates using a proxy-based architecture [28]. Java RMI also lacks many useful features that could make up for its large overhead. It does not support RPC's for static methods, nor does it support code migration or load balancing. A common interface is required to define methods or services that can be invoked remotely. Classes declared on the server-side implement these interfaces and bind instances of their type to the RMI registry. The RMI registry is responsible for declaring and invoking exported services and communicating with clients. The client will perform a lookup request on the remote server to find the appropriate object and acquire the stub that implements the common interface employing a proxy to the remote object instance.

3.4 Communication Channels

Communication channels are a tool to move data in, typically used in concurrency-based applications. Languages such Go [1] and Kotlin [2] are known to have implemented communications channels. Third-party libraries such as Quasar [3] also provide communication channels as part of its concurrency features. However powerful, these implementations of channels were not designed to support workloads that are distributed over a network. The novelty of Triton's communication channels is that they work in a distributed manner, over the internet to maintain consistent bidirectional communications with offloaded tasks.

Chapter 4

Proposed Triton Language Design

Triton is a statically typed, object-oriented DSL for IoT, cyber-physical, and embedded systems development. This is not to say that many of Triton's features cannot be used for general-purpose programming, but rather Triton solves many of the problems specific to these domains. Triton was designed to keep in mind task scheduling, sensor faults, and the general lack of computational resources on embedded hardware. Triton moves many features that are traditionally implemented through API calls to the language level. A keyword prefixes dedicated code blocks to implement scheduling, constraint management, and computational offloading. This design decision serves to make the developers' intentions clear to those who read the source code. Triton's reference implementation is provided in Java, with the compiler targeting JVM bytecode. Therefore, under the current implementation of Triton, Triton code can run on any computer system with a JVM. This may include operating systems such as Windows, Linux, and macOS, along with a variety of CPU architectures including ARM, x86, and PowerPC. With the included JSR-223 [12] compliant script engine, Triton programs or scripts can be embedded within Java. This may not be ideal for real-time systems as only a real-time JVM with a preemptible kernel may achieve

real-time performance. However, Triton's grammar is not platform-specific and could be implemented for other target platforms.

Triton models periodic tasks in dedicated code blocks nested with a 'schedule' block. This makes the intent unmistakable by setting expectations about where scheduled tasks should be located, how the program will behave, and reduce the chance of human error. Periodic tasks can be followed by a set of constraints that can be used to detect and handle various problems, including sensor failure.

Many embedded systems are resource-constrained; however, the software might be deployed on target hardware of varied capability. In resource-constrained configurations, computationally intensive workloads are often offloaded to fog or cloud computing resources. This process may not be necessary on more powerful hardware, leading to an obvious problem. Developers will have to implement and maintain software for both schemes. Triton maintains location and failure transparency to solve this problem and makes computational offloading conditional. Therefore, a singular program can be written and deployed into multiple hardware configurations. Computational offloading can be accomplished in a blocking or an asynchronous way by providing a callback block to be executed when the result is ready. One of Triton's goals is to address the example system architecture shown in Figure 4.1. The diagram shows a theoretical cyber-physical system that utilizes RMD to offload computationally expensive tasks to the cloud.

4.1 Scheduling

Triton supports first-class scheduling by providing dedicated code blocks to handle task scheduling. The advantage of scheduling at the language level instead of through operating system and API calls can be easily shown. Triton's scheduling features allow

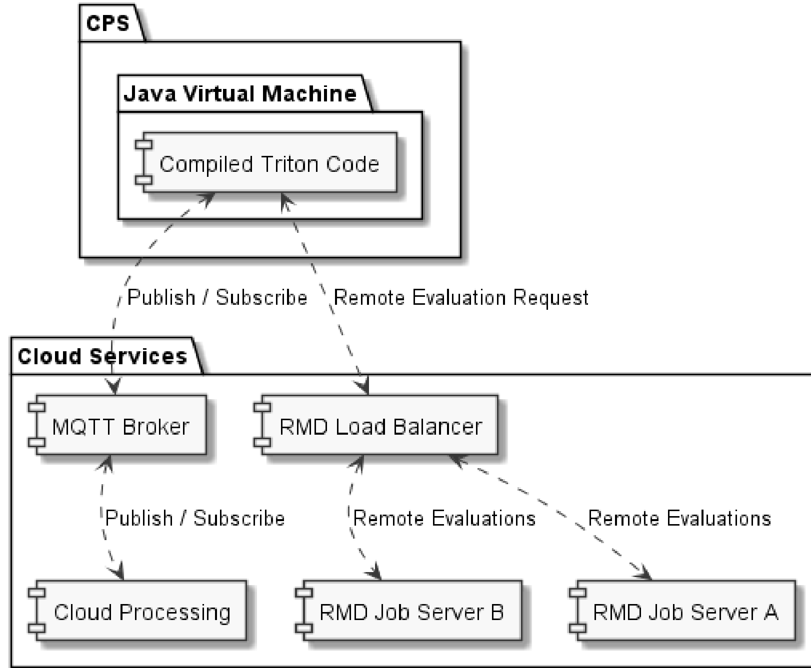

Figure 4.1: Example System Architecture with Triton

for the development of platform agnostic source code because the implementation details are handled by the compiler and/or runtime environment. Numerous additions to the compiler and runtime infrastructure of Triton will make it possible to support different scheduling algorithms (including real-time scheduling algorithms) on a variety of operating systems. This can enable existing source code to be compatible with a variety of hardware and operating system configurations. A simple compiler argument will specify the desired target platform and scheduling algorithms. The compiler will insert operating system specific calls into the generated code to enable this.

A 'schedule' block may be placed as a top-level statement or within a method and may contain multiple tasks. This block can be parameterized to specify a time unit such as nanoseconds, milliseconds, and seconds. The time unit parameter is optional and will default to milliseconds if the programmer does not specify it. In the future, support for real-time scheduling algorithms will be achieved through a schedule block

parameter or command-line compiler argument. Tasks within a scheduling block are parameterized with a period, followed by the block of code to be scheduled. The ability of tasks to complete by their specified deadline will remain a function of the available computing resources, the tasks worst-case execution time, and the scheduling algorithm.

In some cases, tasks may depend on external constraints which may not be violated. To handle constraint violations at the language level, we introduce two new code blocks. The 'constrainedBy' code block may contain many statements but must return a boolean expression. If the constraint is considered to be violated, we provide the 'constrainViolation' block, which will be executed to decide how to proceed. This code block can be used to decide how to handle the situation, including skipping or permanently aborting the task. In Listing 4.1 and 4.2, we provide example code following the proposed grammar. Listing 4.1 defines a schedule block containing two tasks with a period of 4 and 8 milliseconds. Listing 4.2 defines a schedule block containing one periodic task with a constraint and constraint handler. The constraint dictates that the task should proceed normally unless a temperature reading exceeds 125 degrees.

```
schedule {
    task(period=8) {
        ...
    }
    task(period=4) {
        ...
    }
    ...
}
```

Listing 4.1: Periodic scheduling example with two tasks

4.1.1 Constraint Management

Application-level constraints are encouraged to tackle sensor faults that may produce erroneous sensor data. We incorporate a mechanism for declaring constraints and handling constraint violations in this work. By defining two new dedicated code blocks, developers can define constraints in a 'constrainedBy' block by returning a boolean expression and then handling any constraint violations in a 'constraintViolation' block. If the result of the boolean expression is false, the constraint is considered to be violated. A constraint can exist without a 'constraintViolation' block, however, the default behaviour is to skip the task's execution until the constraint is satisfied. Multiple constraints can be chained together for a single task.

In Listing 4.2, we provide an example whereby one task has a constraint on temperature measurements from a thermometer. If the value produced by the temperature were to violate the constraint, the 'constraintViolation' block would be invoked to handle the problem. The 'constraintViolation' block can decide whether the task block should be skipped, aborted, or processed normally. This can be decided by an optional

return value for each of the following scenarios: SKIP, ABORT, and PROCESS. These scenarios can be represented at the language level as either a keyword or builtin enum type.

The default value, SKIP, is implied if the 'constraintViolation' block returns no value. Under this scenario, the task will not run if the constraint is not satisfied. The PROCESS scenario allows the execution of the task to continue unimpeded. Finally, ABORT serves to permanently stop the task from executing. This provides a straightforward and concise way to abort tasks under certain conditions, such as erroneous sensor data, which could be catastrophic.

```
1  schedule {
2      task(period=4) {
3          ...
4      } constrainedBy {
5          thermometer.getReading() < 125
6      } constraintViolation {
7          ...
8          abort
9      }
10     ...
11 }
```

Listing 4.2: Periodic scheduling example with task constraints

4.2 Conditional Computation Offloading

As much of embedded systems hardware is resource-constrained we introduce a mechanism to implement conditional computational offloading. This feature may

offload a task specified within a dedicated code block to fog or cloud computing resources. Implementing computational offloading at the language level addresses several problems. Firstly, programmers do not need to concern themselves with many implementation details. Triton's computational offloading enables both location and failure transparency such that the behaviour of a Triton program is dependable, regardless of whether or not external computing resources are required, available, or there is a network failure. We define a set of metrics developers can use to determine whether a task should be offloaded to external computing resources. In addition, we propose bidirectional communication channels to allow for continuous updates between the application and any task that it offloads. This could be particularly useful in jobs with a longer computational time that may require continuous updates. In the remainder of this section, we demonstrate the usefulness of the proposed solution through a set of example snippets, along with implementation details.

Depending on the application and use-case, communication channels might be required to stream primitive data and objects back and forth between the application and the tasks it offloads. This feature would be most applicable to applications with tasks that have a longer running time.

4.2.1 Conditional Task Offloading

To enable conditional offloading of tasks, metric tracking was implemented as part of the RMD framework and utility functions were added to the Triton standard library to retrieve this information. A boolean expression in parenthesis can be optionally placed following the 'delegate' or 'async' keywords to indicate whether the task should be offloaded. In Listing 4.3, a synchronous (blocking) task is defined, with the offloading condition requiring CPU utilization above 90%. This example serves to show the ease of use of the proposed solution, as the location of task execution is completely

transparent to the developer. The same can be said about the asynchronous example shown in Listing 4.4, however, its offloading condition is dependent on an input value. This is useful when resources are constrained, and the algorithmic complexity does scale well with its input value.

```
1 val a = 100
2 val b = 1000
3
4 val result = delegate (cpuLoad() > 90) {
5     a * b
6 }
7
8 println("Result: " + result)
```

Listing 4.3: Blocking Delegate with Conditional Offloading

```
1 val n = ...
2
3 async (n > 12) {
4     compute(n) // high running time complexity
5 } callback {
6     println("Result: " + it)
7 }
```

Listing 4.4: Asynchronous Delegate with Conditional Offloading

Listing 4.5 is an approximate representation of the compiler output from Listing 4.3. The compiler will generate a synthetic method 'maindel0' to implement the task. Conditional logic is generated by the compiler, whereby RMD will offload the task if the condition is satisfied. Otherwise, the synthetic method will be invoked locally.

When offloading a task, the callsite information and function arguments are passed to RMD. The callsite information tells RMD how to find the method to invoke and includes its name, declaring class, and method descriptor (argument and return types).

```
DelegateInfo main$del$0 = DelegateInfo.of(
    App.class, // containing class
    "main$0$del$0", // name of method
    "(II)I" // method descriptor
)

int a = 100
int b = 1000
int result

if (cpuLoad() > 90 && Rmd.validConfig()) {
    result = ((Number) Rmd.invokeDelegate(
        main$del$0, // callsite information
        new Object[] {a, b} // wrap arguments
    )).intValue() // unbox result to int
} else {
    result = main$del$0(a, b) // local exec
}

println("Result: " + result)
```

Listing 4.5: Approximate Compiler Output from Listing 4.3

To further increase ease of use, we introduce the metric tracking capability to RMD and Triton. Table 4.1 defines the set of metrics that can be used in an offloading

Table 4.1: Offloading Metrics

Metric	**API Call**	**Info**	**Example Condition**
Cpu Load	cpuLoad()	range [0, 1]	cpuLoad() > 0.8
Server Load	serverLoad()	range [0, 1]	serverLoad() < 0.8
Network Latency	rmdLatency()	in milliseconds	rmdLatency() < 25
Outstanding Jobs	rmdJobCount()		rmdJobCount() < 10
Job completion time	rmdJobTime()	in milliseconds	rmdJobTime() < 250

condition with examples. For simplicity reasons, the compiler inserts a reference to the job delegate info when the programmer requires task-specific metrics. This occurs when job completion time is the required metric. This is required as the programmer does not have a handle to the delegate info.

The utility of these metrics is application specific with each metric being advantageous in a variety of scenarios. For instance, server load, job completion time and number of outstanding jobs may all be useful to prevent high load on external computing servers. Network latency will be an advantageous metric for jobs with less required compute time as latency spikes could cause the compute time to be greater than local execution. This metric is measured as the round-trip communication time, in milliseconds. The job completion time metric excludes latency measurements and only measures the amount of compute time on external computing resources. Latency spikes are more likely to arise when utilizing cloud resources instead of hardware that is closer to the edge of the network. Finally, when operating under high CPU load it is advantageous to offload tasks to prevent the system from overloading which could lead to increased compute time and deadline misses. CPU and server load are a measure of the systems total utilization capacity measured between zero and one, with zero being no utilization and one meaning full utilization.

4.2.2 Communication Channels

Communication channels allow developers to program tasks that can remain in constant communication with the master application. We introduce the channel type to accomplish this. Our implementation of channels is generic, meaning that they accept a type parameter. This allows for static, compile-time type checking. The '<-' operator was introduced to facilitate the read and write operations. The operator performs write operations when used as a binary operator, with the channel on the left and the data on the right. Read operations are performed when the operator is used as a unary operator with the channel on the right. Listing 4.6 shows a basic example of a String channel. This indicates to the compiler that a channel's read() and write() methods should both return and accept a String, respectively. The expected output 'Hello, World!' is to be computed on an external computing resource since no offloading condition is provided. The read() method will block until the channel receives a String. Finally, the result is returned to the master application, the callback is invoked and the string is displayed.

```
channel<String> ch

async {
    "Hello, " + <- ch
} callback {
    println(it)
}

ch <- "World!"
```

Listing 4.6: Asynchronous Delegate with Communication Channel

4.3 Grammar

Triton's grammar includes many non-domain-specific features that are non-standard in other programming languages or their development kits. For example, if statements can be used as expressions. In addition, we provide 'when' expressions as a replacement for switch statements. Type inference is also supported for variables defined with 'var' or 'val,' instead of specifying a data type. The grammar also provides support for a script engine through dedicated grammar rules that allow many expressions and statements to be executed outside the confines of a traditional function. Triton's full grammar is shown in Appenix A

4.4 Remote Method Delegation

The computational offloading features of Triton are implemented by the RMD platform. RMD is a platform for offloading computational workloads to external computing resources. Its current implementation supports the remote evaluation of methods defined with Java class files. The design of RMD is not specific to Triton, but rather any language that runs on the JVM. Triton utilizes RMD for computational offloading through API calls generated at compile time. At its core, RMD performs four functions: code migration, load balancing, remote procedure calls, data streaming through communication channels. This design decision allows for a single cohesive application to offload tasks to an external job server with minimal steps required. Its design maintains location and failure transparency and provides a security manager to support the evaluation of untrusted code on job servers. Multiple failure modes are provided to support varying application requirements.

It is the main objective of RMD to enable the simplest possible method for developers to offload computational tasks to external computing resources. The novel can be summarized as part of the following features:

1. Can operate standalone without Triton whereby developers can offload method calls through a method reference. This approach can maintain static type checking.

2. Synchronous and Asynchronously offload tasks to external computing resources

3. Track metrics related to computational offloading, including network latency and job execution time

4. Is fully interoperable with any JVM language.

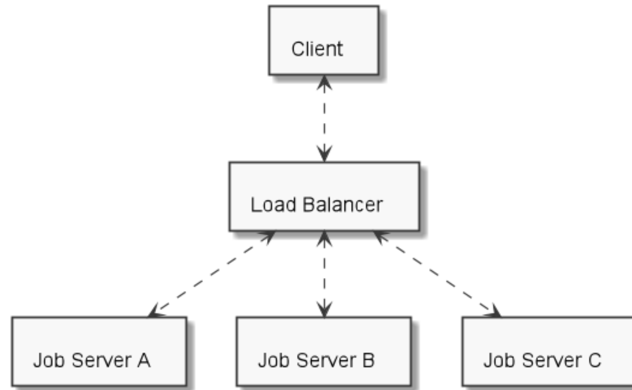

Figure 4.2: 3-Tier RMD Architecture

5. Provides communication channels to support distributed communication of primitive data and objects between an application and the tasks it offloads

4.4.1 Architecture

Architecturally, RMD consists of three separate software packages, the client software, the load balancer, and the job server. In a distributed setting an application can be used in either a 3-tier architecture with a load balancer and one or more job servers as shown in Figure 4.2, or in a 2-tier setup without a balancer. A 2-tier setup can still make use of several job servers because the RMD client software has a load balancer. This comes with the benefit of reduced communication latency but requires prior knowledge of the location of each job server.

When the programmer initiates a request to offload a task, the client module will be responsible for migrating code to external computing resources. This process is described in Section 4.3. If an external load balancer is present, it will also complete this task when required. The load balancer implements a scheduling algorithm to prevent some job servers from becoming overloaded. After migration, an RPC request is made to the remote job server which computes and returns the results.

4.4.2 Code Migration

The code migration process in RMD is *lazy*, meaning that this process is deferred until necessary and only migrates the required code. Figure 4.3 describes the logical flow of this process. The first step involves determining the callsite information of the method to be offloaded. This indicates the location, name, and signature of the method. This step is only required when the platform is used with a language such as Java, as the Triton compiler will provide this information at compile time. If the method has not been previously migrated to the external computing resources, RMD will inspect the bytecode instructions to find any other class files that are required to execute the task. Finally, a migration request including each of the class files is sent to the job server.

RMD relies on ASM, a bytecode manipulation and generation framework [8] which provides the capability to analyze all aspects of a class file. Specifically, we are looking at the following things in each class file:

1. **Inheritance:** parent class, interfaces
2. **Annotations:** of classes, fields, methods, interfaces, and other annotations
3. **Fields:** check the type of each field
4. **Methods:** method signature including arguments and return type
5. **Methods:** check the type of local variables
6. **Methods:** instructions referring to other class files such as retrieving a field

Some jobs may have dependencies that also have their own dependencies. This problem is solved by recursively analyzing each class and ignoring classes that belong to the standard library (don't require migration), and classes that have already been

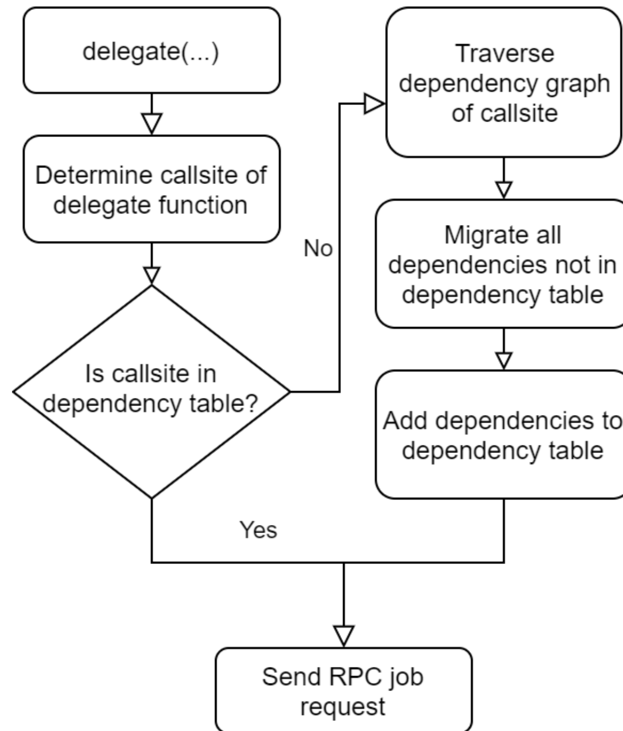

Figure 4.3: Job Delegation Process with RMD

analyzed. The dependency set will be cached, and does not need to be looked up again. Since the client may be in contact with multiple job servers, we use a set to keep track of which dependencies have been migrated to each job server and only send the required dependencies when they are needed to execute a job. The entire migration process is repeated after each subsequent execution of an RMD application. For this reason, versioning issues will not be apparent as each job server will discard old class files.

Figure 4.4 shows an example indicating the relationships between a set of class files in a distributed prime factorization program. Only the classes *PrimeFactorization* and *MathUtility* will be migrated when offloading the *factor* method. The classes belonging to the standard library (such as *Object* and *BigInteger*) are ignored from the migration process because they are guaranteed to exist on the job server. The main class is not marked for migration as the call it delegates belongs to a different class,

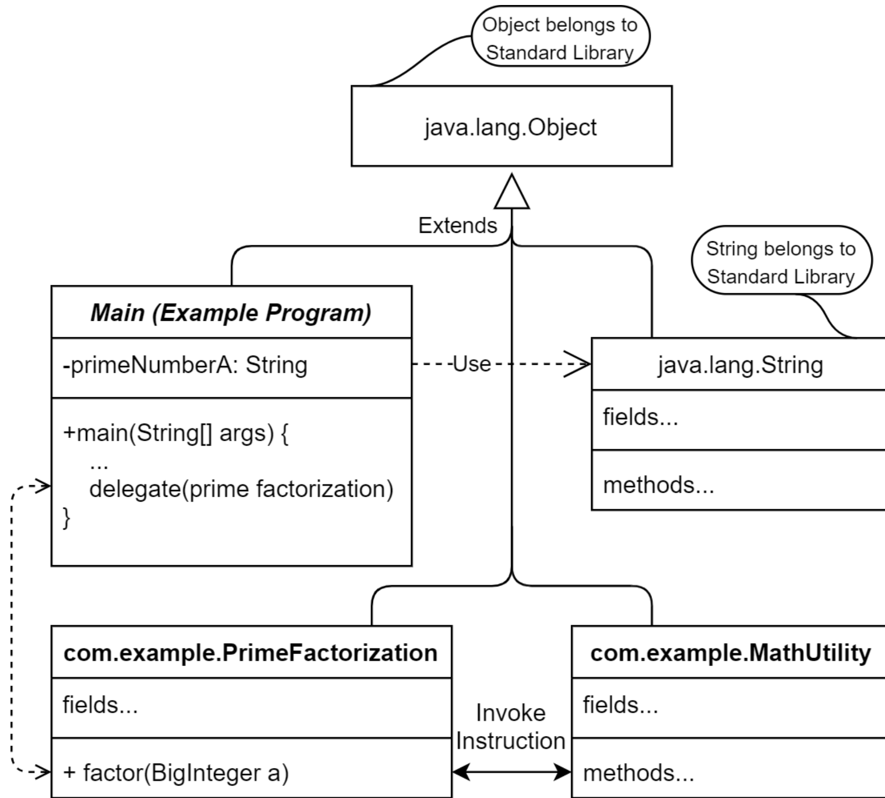

Figure 4.4: Example RMD Program with Dependencies

which also does not depend on the main class. By inspecting all instructions in the *PrimeFactorization* class, invoke instructions are detected that point to a method in the *MathUtility* class. The process is repeated with the *MathUtility* class and no new dependencies are found. The delegate request is mapped to these two dependencies to prevent this process from occurring again. Finally, a migration request will send the two class files to external computing resources, where they will be dynamically loaded. A table will track which dependencies have been migrated to each server.

4.4.3 Communication Channels

As offloaded tasks become more complex with a longer running time, a method of streaming primitive data and objects is required. RMD implements bi-directional

communication channels such that both the application and the tasks it offloads can continuously update each other. A channel object with read() and write() methods is used by the programmer to achieve this. The communication channels must be passed as an input argument to the task that is being offloaded. This is handled by the Triton compiler. The RMD platform will check if any channel objects are passed as input to remote evaluation calls. If channels are used, any channel will be bound to the server which will evaluate the job. This process happens in reverse on the job server, channel objects are bound to the application which made the request. This binding process ensures that the channel's read() and write() methods send data to the correct location.

4.4.4 Security

Since the job server might be responsible for loading and executing untrusted code, a security manager is provided to inhibit malicious actors. This security manager can be used in addition to password-based authentication. The JVM provides a few ways to define security policies. First, the JVM supports security through a policy file to allow or deny permissions. This solution is not acceptable because we need to simultaneously grant permissions to the RMD job server platform whilst denying the same permissions to all untrusted code. To achieve this level of flexibility, we must implement a custom security manager. When an application invokes privileged actions such as network or file system access, the JVM will check the system security manager to determine whether the action is permitted.

The job server has a security policy that will deny all permissions to untrusted code whilst not interfering with any of the permissions required by the job server. The security manager operates on a per-thread basis so that it can allow the job server to communicate with clients while at the same time preventing untrusted code

from accessing the internet or other system resources. The JVM allows applications to specify their own custom security manager at run-time, which is implemented by inheriting from the **java.lang.SecurityManager** class. Every time a permission check is done by the JVM, RMD's security manager checks the calling thread's thread group and compares it with the thread group that jobs belong to. When a new job arrives, it is dispatched onto the thread-pool where each of its threads belongs to the same thread group. This allows the security manager to differentiate between the RMD infrastructure and untrusted code.

4.4.5 Distributed Transparency

Location and failure transparency are some of the key design methodologies in the project. Through location transparency, a developer cannot tell the difference between jobs that are executed locally or by one of many external job servers. Errors caused by application-level exceptions are caught, then a sanitized stack trace back to the client where it can be re-thrown and analyzed by a developer. This creates the appearance that the task has been executed locally.

The default failure mode described in Section 4.4.6 allows tasks to be executed locally under a failure scenario. This could be a result of network failure, server crash, or if no configuration file is provided. The importance of design philosophy allows the behaviour of a program to remain constant, whether or not a task can be offloaded.

4.4.6 Failure Modes

Failures are inevitable in distributed systems. Most of these failures will be a result of network-related issues, including interference in wireless systems, high network traffic, or even malicious denial-of-service (DoS) attacks. Other events outside the control

of the developer including extreme weather events may temporarily shut down data centers, destroy fog resources, or reduce the quality of service. Three failure modes are introduced to maintain deterministic behaviour in programs utilizing computational offloading. The developer may specify their preferred failure mode in the configuration file.

The failure modes can be described as follows: local execution under failure scenarios, an exception-based model, and finally, a retry protocol. Under the exception-based failure protocol, RMD will throw an exception when it encounters a failure. The programmer can catch this exception and handle it however they choose. This protocol should be used when the tasks cannot be executed locally or when the running-time of local execution is too great for results to be useful.

The retry protocol is self-explanatory. Under a failure scenario, the application will continue attempts to contact external computing resources until successful. Synchronous requests will block until the issues can be resolved. This failure mode might be advantageous if local execution of the task is not possible.

4.4.7 Configuration

Configuration of RMD can be accomplished by developers through a configuration file. In the configuration file, developers will define a set of hosts referring to external computing resources available for computational offloading. Due to the distributed transparency, this may include both job servers and load balancers, and the client is not required to differentiate between the two. The RMD client software will distinguish between them as it does not know the difference. Obviously, increased latency will be observed for workloads traveling through an external load balancer. To handle failures in computational offloading, the desired failure mode should be specified in the configuration file. Developers will choose between a retry protocol, an exception-based

Table 4.2: Example Applications and Use-Cases

Industry	**Example Application**	**Note**
Industrial Robotics	Cleaning Robot	May Require LLVM support
	Robotic Assembly Arm	
	Sorting Machine	
IoT	Smart Watch	Android
	Smart Lock	
	Smart Door Bell	
Embedded Systems	Resource Constrained	May Require LLVM support
	Security Camera	
	Virtual Reality	

model, and finally, local job execution.

In the event that no configuration file is provided, a warning message will be displayed. The bahaviour of the program will not be altered in this case. Under the default configuration, RMD operates under the assumption that no external computing resources are available. Instead, requests to offload tasks will be executed locally.

4.5 Applications and Use-Cases

Triton can be applied to many application scenarios and is not limited to the cyber-physical systems domain. Triton may also prove useful for general purpose programming where applications may also be expected to run on poorly performing computer systems. Applications to the fields of industrial robotics, IoT, resource-constrained systems may benefit most from Triton due to its scheduling and computational offloading features. Example applications and use-cases have been defined in Table 4.2. Some of the proposed applications of Triton may require support for native compilation to run. This can be done by providing compiler support for LLVM.

Chapter 5

System Evaluation and Experimental Analysis

In this chapter, we discuss the advantages and performance implications of Triton's features. We discuss the penalties incurred by computational offloading and when to use it. Through two experiments, we evaluate the cost of computational offloading with Triton. Analysis of experimental results shows the usefulness of computational offloading with Triton and indicates desirable use-cases.

5.1 System Evaluation

The proposed solution manages to achieve a higher degree of abstraction because many of the features required by embedded systems that are typically implemented through API calls are now resolved at the language level. It is through this mechanism in which our proposed DSL can reduce boilerplate code to make for an easier and less error-prone development process.

Both the scheduling of real-time tasks and computing their associated constraints

are implemented with a high degree of transparency. This reduces the demand for domain-specific knowledge on developers. The constraint system we have proposed allows for first-class constraint handling. Since constraints are plentiful in a typical embedded system, the way in which they are addressed is important.

To evaluate the proposed DSL, we compare equivalent programs written in our DSL, with ones written in the C language. Shown in Listing 5.1 is a snippet from a program written in our proposed DSL. Its function is to control the speed of a robotic vehicle with two motors using PID controllers and to slow down when the robot exceeds a maximum speed, or to stop when an object is too close.

When comparing to the same program implemented in C or Java (real-time JVM), we observe a considerable reduction in the source code's size. Utilizing Triton, we observed 63 lines of code compared to 134 and 85 for a similar program implemented in C and real-time Java, respectively. Of course, this improvement will not scale with larger programs though it should scale linearly with respect to the number of tasks. Code from our proposed DSL remains much cleaner and is easier to read. The features we provide would normally need to be implemented through API calls, and additional control flow is now handled at the language level. The grammar provides additional readability because it standardizes the way tasks and constraints are defined, providing an expectation of where to look when reading through the code.

```
double objDist = 0

schedule(timeUnit = MILLISECONDS) {
        task(period = 50) {
            setMotorSpeed(leftWheel, SPEED)
            setMotorSpeed(rightWheel, SPEED)
        } constrainedBy {
            abs(leftWheel.getSpeed()) < MAX_SPEED ||
                abs(rightWheel.getSpeed()) < MAX_SPEED
        } constraintViolation {
            skip // speed too high
        } constrainedBy {
            objDist < 25 // 25 cm
        } constraintViolation {
            stop()
            skip // don't crash
        }

        task(period = 100) {
            objDist = ultrasonicSensor.getDistance()
        }

        ...
}
```

Listing 5.1: Robot with PID controllers and object detection

5.2 Experimental Setup and Analysis

This section evaluates conditional computational offloading under two experimental scenarios compared to unconditional offloading and local execution. Experiments were performed on a Raspberry Pi Zero with a local job server handling offloading requests over WiFi.

5.2.1 Offloading by CPU Utilization

To evaluate performance, a synthetic load was created that requires approximately 750 ms of CPU time on the Raspberry Pi Zero. The benchmark was scheduled to run in two simultaneous scheduled period tasks, with a period of 1 second. Given that the Raspberry Pi Zero is a single-core machine, it will not be possible for both tasks to complete before their 1000 ms deadline. In Figure 5.1, we evaluate the program with an offloading condition that requires CPU utilization to surpass 60%. For the most part, task A executes locally in 750 ms, whilst task B is offloaded to a powerful machine and computes in approximately 70 ms. Occasionally, when both task A and B begin at roughly the same time, CPU utilization is perceived to be low, causing neither task to be offloaded. This leads to a spike in required computation time for both tasks.

Figure 5.2 and 5.3 indicate task computing time for unconditional computational offloading and local execution, respectively. Task computational time is highly correlated in both scenarios, although the tasks fail to complete by their deadline when both are executed locally. Under local task execution, each task's average running time has more than doubled from 750 ms to 1550 ms. Under the scenario of unconditional computational offloading, all tasks manage to complete before their scheduled deadlines with an average but inconsistent running time of 165 ms. The added overhead is likely

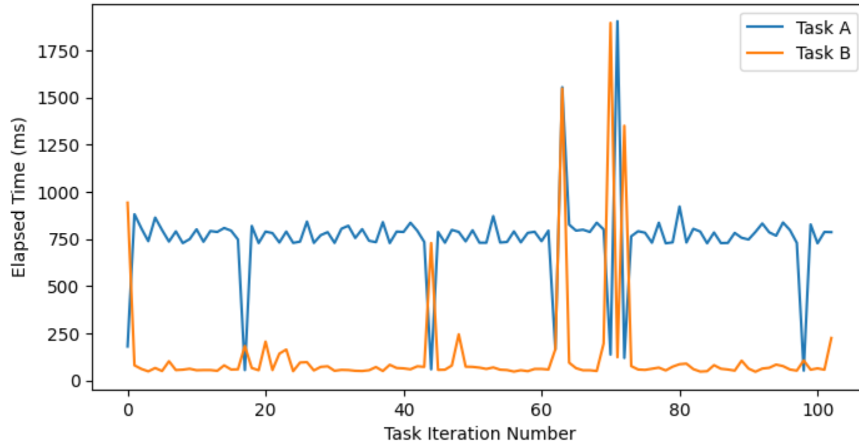

Figure 5.1: Conditional Computational Offloading Performance

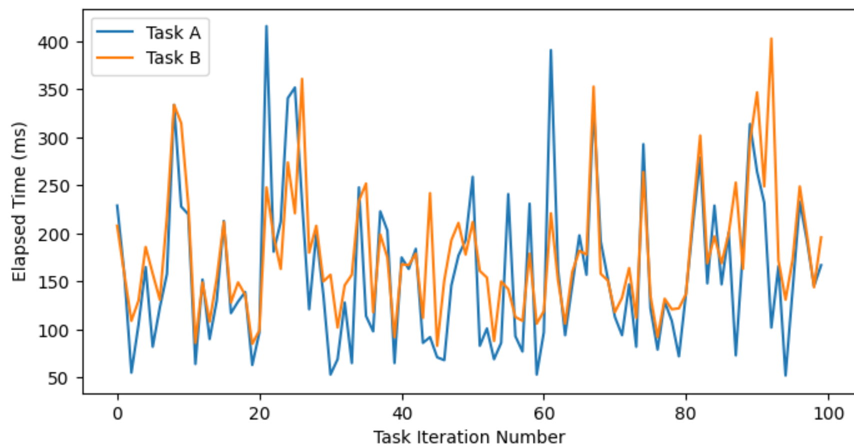

Figure 5.2: Unconditional Computational Offloading Performance

exaggerated due to execution on a single-core processor.

5.2.2 Offloading by Algorithmic Input Value

This experiment evaluates conditional computational offloading of a recursive Fibonacci algorithm based on the input number. Since the recursive Fibonacci algorithm has an exponential running time complexity, its running-time does not scale well as the input value is increased. On resource-constrained hardware like the Raspberry Pi used in this experiment, there is a trade-off point where offloading the computation

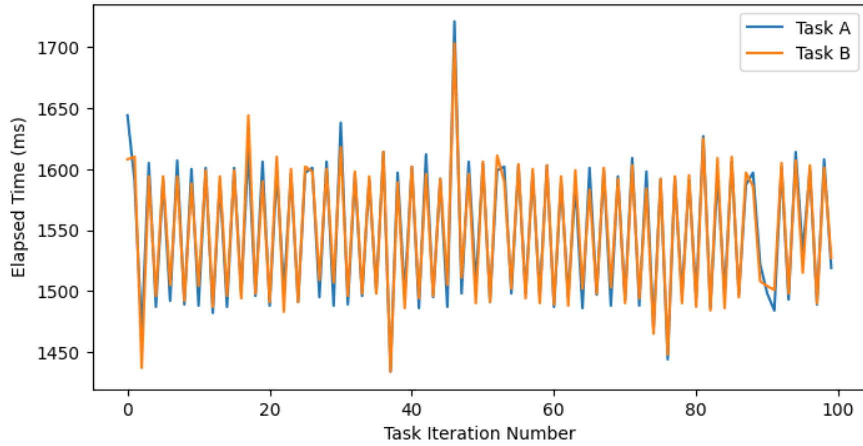

Figure 5.3: Local Task Execution Performance

to external computing resources will improve the computational time required. It is much more efficient for lower Fibonacci numbers to compute them locally due to the added network latency and RPC overhead.

Consider the code in Listing 5.2 to see the implementation of this benchmark. Note that the same code can be used to evaluate both conditional and local execution. Triton will default to local execution if a valid RMD configuration is not found. To implement unconditional computational offloading, the conditional expression may be removed; otherwise, it should always evaluate to true. At $Fib(28)$ and beyond, the computation is offloaded to powerful external computing resources.

```
fun fib(long n): long = if (n <= 1) 1 else fib(n - 1) + fib(n - 2)

fun main(String[] args) {
    for (var i = 0; i < 46; i += 1) {
        val result = delegate (i >= 28) {
            return fib(i)
        }
        println("fib(" + i + ") = " + result)
    }
}
```

Listing 5.2: Offloading Fibonacci's algorithm by Input Value

Table 5.1 shows the required computational time for Fibonacci numbers up to 45. Before $Fib(20)$, the unconditional computational offloading has an average of 16 ms of extra overhead compared to conditional computational offloading. The gap conditional offloading and unconditional offloading continuously shrinks to 0 by $Fib(27)$ where it becomes more efficient to offload the function evaluations. For reference, each Fibonacci number's computational time is recorded when executing locally on the Raspberry Pi. Computational time under the local execution grows at the same rate as when offloaded, however, due to the vastly superior external computing resources the tasks can be completed much faster. $Fib(35)$ computes in 1400 ms when executed locally and in approximately 60 ms when offloaded. At $Fib(45)$ we observe 174 seconds of compute time under local execution and approximately 5.6 of compute time when offloaded. Performance results were upwards of 30 times slower for local execution of high Fibonacci numbers than when offloaded. The results of each trial are averaged over 10 runs with some inconsistency in compute time caused by network latency and operating system scheduling.

Table 5.1: Recursive Fibonacci execution time (ms) under conditional offloading, unconditional offloading, and local execution. The results are averaged over 10 runs.

Fib(n)	Conditional Offloading	Unconditional Offloading	Local Execution
0	0.009 ms	15.512 ms	0.008 ms
1..15	$<$ 0.2 ms	$\approx$ 16 ms	$<$ 0.2 ms
16	0.191 ms	13.360 ms	0.202 ms
17	0.279 ms	28.336 ms	0.276 ms
18	0.425 ms	19.719 ms	0.427 ms
19	0.661 ms	10.999 ms	0.664 ms
20	1.056 ms	15.079 ms	1.047 ms
21	1.700 ms	13.042 ms	1.687 ms
22	2.807 ms	11.076 ms	2.982 ms
23	4.342 ms	12.336 ms	4.350 ms
24	7.039 ms	11.474 ms	7.036 ms
25	11.605 ms	16.230 ms	11.633 ms
26	23.856 ms	11.889 ms	21.165 ms
27	29.589 ms	13.686 ms	31.763 ms
28	27.776 ms	13.483 ms	48.041 ms
29	28.211 ms	12.413 ms	84.331 ms
30	31.527 ms	15.316 ms	124.903 ms
31	25.882 ms	23.237 ms	217.884 ms
32	31.878 ms	24.962 ms	338.626 ms
33	30.134 ms	30.343 ms	557.083 ms
34	48.599 ms	47.340 ms	900.883 ms
35	64.040 ms	60.264 ms	1441.240 ms
36	88.573 ms	89.419 ms	2358.651 ms
37	143.535 ms	137.039 ms	3822.340 ms
38	214.732 ms	207.244 ms	6170.503 ms
39	326.488 ms	321.259 ms	9952.850 ms
40	521.578 ms	520.606 ms	16004.14 ms
41	855.657 ms	814.626 ms	26162.01 ms
42	1317.41 ms	1313.67 ms	42027.27 ms
43	2107.12 ms	2147.82 ms	67472.28 ms
44	3409.99 ms	3574.62 ms	109125.5 ms
45	5568.98 ms	5800.75 ms	174089.5 ms

Chapter 6

Conclusions

With the increasing popularity of IoT, embedded, and edge devices, it is important that they continue to function when cloud and fog computing resources are under stress or unavailable. Of course, computational offloading is useful for maintaining timeliness and meeting computational deadlines, but many such systems exhibit an over-reliance on fog and cloud computing resources. An over-reliance on external computing resources can lead to system failures, reduced or no service. Shifting requirements may present developers with scenarios where resource-constrained hardware is upgraded. This presents developers with two distinct programming paradigms: computational offloading and local execution. Developing and maintaining software that must function in both environments presents unnecessary work. Other concerns are also present in many embedded and/or real-time systems, including scheduling. Traditional scheduling implementations require numerous API calls, and some algorithms are platform dependent. Therefore, we introduce Triton, a DSL with novel language features, to address many of these prevalent problems.

Triton can ease the development of IoT, embedded, and cyber-physical systems by reducing the chance of programming errors and limiting the time developers spend

writing boilerplate code. The simplicity of Triton's grammar is demonstrated in example code snippets. We introduce first-class scheduling support in addition to computational offloading with communication channels to Triton. Dedicated grammar rules support task scheduling with constraint management at the language level through dedicated code blocks. Tasks can be scheduled periodically whilst subjected to various constraints, including sensor inputs. Two new keywords, *abort* and *skip*, were defined to either abort or skip a task's execution if its constraints are violated. Both synchronous and asynchronous computational offloading is introduced at the grammar level and implemented by the RMD platform which includes code migration, load balancing, remote evaluation, and security features. Computational offload in Triton is conditional, depending on developer specified criteria, such as CPU utilization, network latency, number of outstanding jobs, etc. This advantage can increase computational throughput in scenarios where offloading is not desired. Such scenarios may include tasks with algorithms where computational time scales with input values. Bidirectional communication channels were introduced to enable an application to maintain constant communications with the tasks it offloads.

Implementation details are described with system evaluations in which we compare Triton with traditional embedded development techniques. We show that the proposed DSL achieves a high degree of abstraction, is easy to read and understand, reduces boilerplate code, and can eliminate entire classes of errors. We demonstrate the advantage of using Triton for computational offloading through several experiments. Experimental results show increased computational throughput on resource-constrained hardware by offloading tasks to external computing resources. This is accomplished without total reliance on external hardware and maintains the same program behaviour if these servers go offline or network failure occurs.

6.1 Future Works

A few of Triton's issues may present concerns for the real-world viability of the language. We aim to solve several of Triton's present issues in the future. Firstly, the Triton compiler only provides support for the JVM target at this time. This limitation could be a problem for embedded systems developers, especially for real-time systems. Therefore, we plan to explore the proposed DSL implementation on other target platforms such as LLVM. This will include real-time operating systems with support for real-time scheduling algorithms that can be specified by a compiler option. If this happens, the RMD platform will need enhancements to run native code. The lack of development environment support may present another hurdle for developers. Reliance on debuggers, inspections, syntax, and semantic checking is an important factor in developer productivity. Addressing these problems will increase Triton's viability for many projects.

The implementation of conditional computational offloading may depend on several factors that may be difficult for a human to optimize the decision-making strategy when combined. We aim to explore the use of optimization algorithms to maximize computational throughput on resource-constrained systems to address this.

Enhancements to the RMD framework should be introduced to increase security and support new features. For instance, the RMD job server lacks support for request rate limiting. This could pose a security vulnerability if job servers are permitted to run code from public users. Even with password authentication, a malfunctioning device could inadvertently cause a denial or reduced service. Other enhancements may include the addition of other communication protocols such as Bluetooth and Zigbee. This capability will extend the usefulness of computational offloading to more devices.

Appendix A

Antlr4 Triton Grammar

Listing A.1: Antlr4 Grammar

```
grammar Triton;

file
    : NL* (packageDef NL*)? imp* NL* topLevelStatement* NL* EOF
    ;

script
    : NL* (packageDef NL*)? imp* NL* (scriptStatement semi)*
        (scriptStatement semi?)? NL* EOF
    ;

scriptStatement
    : statement
    | functionDef
    | varDef
    ;
```

```
16
17 topLevelStatement
18     : (
19     functionDef
20     ) semi?
21     | varDef semi
22     | schedule semi
23     | clazz semi
24     ;
25
26 statement
27     : block
28     | whileStatement
29     | forStatement
30     | expression
31     | async
32     | varDef
33     | channel
34     | returnStatement
35     | schedule
36     | SEMICOLON
37     ;
38
39 expression
40     : LPAREN NL* wrapped=expression NL* RPAREN
41     | literal
42     | preceeding=expression NL* DOT NL* id=IDENTIFIER
```

```
43	    | name=fqn
44	    | preceeding=expression NL* DOT NL* call=functionCall
45	    | call=functionCall
46	    | ifStatement
47	    | whenExpr
48	    | delegate
49	    | newStatement
50	    | listDef
51	    | preceeding=expression NL* DOT NL* assignment
52	    | assignment
53	    | typeCast
54	    | expression indices (NL* ASSIGN NL* assign=expression)?
55	    | lhs=expression NL* (RANGE) NL* rhs=expression
56	    | (PLUS | MINUS | NOT | CHAN) NL* unaryOperand=expression
57	    | lhs=expression NL* (POW) NL* rhs=expression
58	    | lhs=expression NL* (CHAN) NL* rhs = expression
59	    | lhs=expression NL* (MULT | DIV | MOD) NL* rhs=expression
60	    | lhs=expression NL* (PLUS | MINUS) NL* rhs=expression
61	    | lhs=expression NL* (GT | LT | GTE | LTE | EQUALS | NOT_EQ) NL*
	      rhs=expression
62	    | lhs=expression NL* (AND | OR) NL* rhs=expression
63	    ;
64
65	schedule
66	    : SCHEDULE NL* LBR NL* (task NL*)* RBR
67	    ;
68
```

```
69  task
70      : TASK NL* taskParams NL* block NL* constraint*
71      ;
72
73  taskParams
74      : LPAREN ((NL* PERIOD NL* ASSIGN)? NL* period=INT) NL* RPAREN
75      ;
76
77  constraint
78      : CONSTRAINT NL* condition=block (NL* CONSTRAINT_VIOLATION NL*
            violation=block)?
79      ;
80
81  indices
82      :   (NL* '[' NL* expression NL* ']')+
83      ;
84
85  clazz
86      : (modifierList NL*)? CLASS NL* IDENTIFIER (NL* shortConstructor)?
            (NL* inheritance)?
87      ;
88
89  shortConstructor
90      : '(' varDefList ')'
91      ;
92
93  inheritance
```

```
94      : ‘:’ (NL* classInheritance NL* ’,’) NL* interfaceInheritance
95      | ‘:’ NL* classInheritance
96      | ‘:’ NL* interfaceInheritance
97      ;
98
99  classInheritance
100     : fqn NL* ‘(’ NL* (expressionList NL*)? ‘)’
101     ;
102
103 interfaceInheritance
104     : fqn (NL* ‘,’ NL* fqn)*
105     ;
106
107 /** RMD **/
108
109 delegate
110     : DELEGATE (NL* LPAREN condition=expression RPAREN)? NL* body=block;
111
112 async
113     : ASYNC (NL* LPAREN condition=expression RPAREN)? NL* body=block (NL*
          CALLBACK NL* cb=block)?
114     ;
115
116 channel
117     : (modifierList NL*)? ‘channel’ NL* (‘<’ NL* type NL* ‘>’ NL*)?
          IDENTIFIER
118     ;
```

```
119
120 /** RMD **/
121
122 assignment
123     :   <assoc=right>
124         IDENTIFIER NL*
125         (   ASSIGN
126         |   PLUS_EQ
127         |   MINUS_EQ
128         |   MULT_EQ
129         |   DIV_EQ
130         |   MOD_EQ
131         |   POW_EQ
132         )
133         NL* val=expression
134     ;
135
136 typeArguments
137     : '<' NL* (typeArgument (NL* ',' typeArgument)* NL*)? '>'
138     ;
139
140 typeArgument
141     : fqn typeArguments?
142     | arrayType
143     ;
144
145 listDef
```

```
146     : (typeArguments NL*)? ‘[’ NL* (expressionList NL*)? ‘]’
147     ;
148 
149 typeCast
150     : ‘(’ NL* type NL* ‘)’ NL* expression
151     ;
152 
153 varDef
154     : (modifierList NL*)? (type | VAR | VAL) NL* IDENTIFIER (NL* ASSIGN
            NL* expression)?
155     ;
156 
157 varDefList
158     : varDef (NL* ‘,’ varDef)*
159     ;
160 
161 functionCall
162     : IDENTIFIER NL* LPAREN (NL* expression (NL* COMMA NL* expression)*)?
            NL* RPAREN
163     ;
164 
165 functionDef
166     : (modifierList NL*)? FUN NL* IDENTIFIER NL* LPAREN functionParamDefs?
            RPAREN NL* (‘:’ NL* (VOID_T | type) NL*)?
167     (block? | (‘=’ NL* expression))
168     ;
169 
```

```
170 functionParamDefs
171     : functionParam (NL* COMMA NL* functionParam)*
172     ;
173
174 functionParam
175     : (modifierList NL*)? type NL* IDENTIFIER
176     ;
177
178 type
179     : primitiveType
180     | fqn typeArguments?
181     | arrayType
182     ;
183
184 arrayType
185     : (primitiveType | fqn) (NL* '[' NL* ']')+
186     ;
187
188 primitiveType
189     : INT_T
190     | LONG_T
191     | BOOL_T
192     | BYTE_T
193     | FLOAT_T
194     | DOUBLE_T
195     ;
196
```

```
197 packageDef
198     : PACKAGE fqn semi
199     ;
200
201 imp
202     : IMP fqn semi
203     ;
204
205 block
206     : LBR NL* (statement semi)* (statement semi?)? NL* RBR
207     ;
208
209 ifStatement
210     : IF NL* LPAREN condition=expression RPAREN NL* body=statement
          SEMICOLON?
211     (NL* ELSE NL* else_=statement)?
212     ;
213
214 whenExpr
215     : WHEN NL* (LPAREN NL* expression NL* RPAREN NL*)? '{' NL* (whenCase
          semi)* (whenElse semi?)? '}'
216     ;
217
218 whenCase
219     : whenCondition NL* '->' NL* (expression | block)
220     ;
221
```

```
222 whenCondition
223     : expression
224     | ('is' | '!is') type
225     ;
226
227 whenElse
228     :   ELSE NL* '->' NL* (expression | block)
229     ;
230
231 whileStatement
232     : WHILE NL* LPAREN condition=expression RPAREN NL* body=statement
233     | DO NL* body=statement NL* WHILE NL* LPAREN condition=expression
          RPAREN
234     ;
235
236 returnStatement
237     : RETURN expression?
238     ;
239
240 newStatement
241     : NEW NL* fqn NL* LPAREN (expression (NL* COMMA NL* expression)*)? NL*
          RPAREN
242     | array=NEW NL* (fqn | primitiveType) NL* ('[' NL* expression NL* ']')+
243     ;
244
245 forStatement
```

```
246     : FOR NL* LPAREN NL* forControl NL* RPAREN NL* ((statement semi?) |
           SEMICOLON)
247     | FOR NL* statement
248     ;
249
250 forControl
251     : (modifierList NL*)? (type | VAR | VAL) NL* IDENTIFIER NL* COLON NL*
           expression
252     | ((varDef | init=expression) NL*)? SEMICOLON NL*
           (condition=expression NL*)? SEMICOLON (NL* expressionList)?
253     ;
254
255 localVariable
256     : type localVarDefList
257     ;
258
259 localVarDefList
260     : localVarDef (COMMA localVarDef)*
261     ;
262
263 localVarDef
264     : IDENTIFIER ASSIGN expression
265     ;
266
267 expressionList
268     : expression (NL* COMMA NL* expression)*
269     ;
```

```
270
271 literal
272     : number
273     | bool
274     | string
275     | NULL
276     ;
277
278 bool
279     : TRUE | FALSE
280     ;
281
282 number
283     : INT | HEX | FLOAT
284     ;
285
286 string
287     : StringLiteral
288     ;
289
290 fqn
291     : IDENTIFIER (DOT IDENTIFIER)*
292     ;
293
294 modifierList
295     : modifier+
296     ;
```

```
297
298 modifier
299     : visibilityModifier
300     | 'final'
301     ;
302
303 visibilityModifier
304     : PUBLIC
305     | PRIVATE
306     | PROTECT
307     ;
308
309
310 // tokens
311
312
313 StringLiteral
314     :   '"' StringCharacters? '"'
315     ;
316 fragment
317 StringCharacters
318     :   StringCharacter+
319     ;
320 fragment
321 StringCharacter
322     :   ~["\\]
323     |   EscapeSequence
```

```
324        ;
325    //  3 .10.6 Escape Sequences for Character and String Literals
326    fragment
327    EscapeSequence
328        :   '\\' [btnfr"'\\]
329        |   OctalEscape
330        |   UnicodeEscape
331        ;
332
333    fragment
334    OctalEscape
335        :   '\\' OctalDigit
336        |   '\\' OctalDigit OctalDigit
337        |   '\\' ZeroToThree OctalDigit OctalDigit
338        ;
339    fragment
340    OctalDigit
341        :   [0-7]
342        ;
343    fragment
344    UnicodeEscape
345        :   '\\' 'u' HexDigit HexDigit HexDigit HexDigit
346        ;
347
348    fragment
349    ZeroToThree
350        :   [0-3]
```

```
351     ;
352
353 IF      : 'if';
354 IS      : 'is';
355 DO      : 'do';
356 NEW     : 'new';
357 FUN     : 'fun';
358 VAR     : 'var';
359 VAL     : 'val';
360 FOR     : 'for';
361 IMP     : 'import';
362 INT_T   : 'int';
363 TASK    : 'task';
364 LONG_T  : 'long';
365 BYTE_T  : 'byte';
366 NULL    : 'null';
367 TRUE    : 'true';
368 ELSE    : 'else';
369 WHEN    : 'when';
370 FALSE   : 'false';
371 CLASS   : 'class';
372 WHILE   : 'while';
373 FLOAT_T : 'float';
374 NATIVE  : 'native';
375 DOUBLE_T: 'double';
376 PERIOD  : 'period';
377 RETURN  : 'return';
```

```
378 PUBLIC : 'public';
379 PRIVATE : 'private';
380 PACKAGE : 'package';
381 PROTECT : 'protected';
382 VOID_T : 'void';
383 BOOL_T : 'boolean';
384 LPAREN : '(';
385 RPAREN : ')';
386
387 SCHEDULE : 'schedule';
388 CONSTRAINT : 'constrainedBy';
389 CONSTRAINT_VIOLATION : 'constraintViolation';
390
391 // RMD
392 DELEGATE : 'delegate';
393 ASYNC : 'async';
394 CALLBACK : 'callback';
395
396
397 LBR    : '{';
398 RBR    : '}';
399 RANGE  : '..';
400
401
402 IDENTIFIER : [a-zA-Z_][a-zA-Z_0-9]*;
403
404 WS  : [\u0020\u0009\u000C] -> skip;
```

```
405
406 NL: ‘\u000A’ | ‘\u000D’ ‘\u000A’ ;
407
408 semi: NL+ | SEMICOLON | SEMICOLON NL+;
409
410 COMMENT :  ‘/*’ .*? ‘*/’ -> channel(HIDDEN);
411
412 LINE_COMMENT : ‘//’ ~[\r\n]* -> channel(HIDDEN);
413
414 // arithmetic
415
416 PLUS   : ‘+’;
417 MINUS  : ‘-’;
418 MULT   : ‘*’;
419 DIV    : ‘/’;
420 MOD    : ‘%’;
421 POW    : ‘*’;
422
423 CHAN   : ‘<-’;
424 ASSIGN : ‘=’;
425 PLUS_EQ : ‘+=’;
426 MINUS_EQ: ‘-=’;
427 MULT_EQ : ‘*=’;
428 DIV_EQ : ‘/=’;
429 MOD_EQ : ‘%=’;
430 POW_EQ : ‘**=’;
431
```

```
432 // logical
433
434 EQUALS : '==';
435 NOT_EQ : '!=';
436 AND    : '&&';
437 OR     : '||';
438 NOT    : '!';
439 GT     : '>';
440 GTE    : '>=';
441 LT     : '<';
442 LTE    : '<=';
443
444 COLON  : ':';
445 DOT    : '.';
446 COMMA  : ',';
447 SEMICOLON : ';';
448
449 INT
450     : Digit+
451     ;
452
453 HEX
454     : '0' [xX] HexDigit+
455     ;
456
457 FLOAT
458     : Digit+ '.' Digit* ExponentPart? [fF]?
```

```
459     | '.' Digit+ ExponentPart? [fF]?
460     | Digit+ ExponentPart [fF]?
461     | Digit+ [fF]
462     ;
463
464 fragment
465 ExponentPart
466     : [eE] [+-]? Digit+
467     ;
468
469 fragment
470 HexExponentPart
471     : [pP] [+-]? Digit+
472     ;
473
474 fragment
475 Digit
476     : [0-9]
477     ;
478
479 fragment
480 HexDigit
481     : [0-9a-fA-F]
482     ;
```

Appendix B

Triton Quick Reference Sheet

Hello, World program

```
fun main(String[] args) {
    println("Hello, World!")
}
```

Function declarations

```
fun add(int a, int b): int {
    return a + b
}
fun sub(int a, int b) = a - b
fun mult(int a, int b): int = a * b
```

For Loops

```
for (var i = 0; i < 10; i++) {
    println(i)
}
for (var a : array) {
    println(a) // for each
}
for {
    println("Infinite loop")
}
for println("Infinte Loop")
```

While Loop

```
while (i < 100) {
    println(i)
    i += 2
}
```

If statements

```
// use as expression
```

```
int value = if (x > 100) x else -x
if (x > 5 && y < 10) {
    println("True...")
} else {
    println("False...")
}
```

When statements

```
var a = when (x) {
    10 -> "a"
    20 -> "b"
    else -> "c"
}
```

```
var a = when {
    x == 10 -> "a"
    x == 20 -> "b"
    else -> "c"
}
```

Variable Declarations

```
int a = 100
var b = 200 // inferred type int
val c = 300 // immutable
int d = 10, e = 20
```

Arrays and Objects

```
int[] ia = new int[10]
int[][] ia2d = new int[10][100]
val lst = new LinkedList()
```

Synchronous Offloading

```
int a = ...
int b = ...
int c = delegate {
    a * b
}
// offload when c > 1000
int d = delegate (c > 1000) {
    compute(c)
}
```

Asynchronous Offloading

```
delegate {
    a * b
} callback {
    println(it)
}
// offload when a > 100
delegate (a > 100) {
    compute(a)
} callback {
    println(it)
}
```

String operations

```
println("a" + "b") // ab
// add variables
println(a + a)
println("$a $b")
```

Task Scheduling

```
schedule {
    task (period=1000) {
        println("Hi")
    }
    task (period=100) {
        println("Test")
    } constrainedBy {
        a < 10
    } constraintViolation {
        println("oh no")
    }
}
```

Builtin Functions

```
// printing
println()
println(x)
print(x)
// read from stdin
readline()
readInt()
readBoolean()
readFloat()
readDouble()
// time
time() // time in ms
nanoTime() // time in ns
// sorting
sort(list)
// RMD
cpuLoad() // [0, 1]
serverLoad() // [0, 1]
rmdLatency() // in ms
rmdJobCount()
// only from ctx of offload
    condition
rmdJobTime() // in ms
```

RMD Configuration File

```
config {
    hosts: {
        "localhost"
        "example.com"
    }
    errorStrategy: "RETRY"
    port: 5050
}
```

Bibliography

[1] The go programming language specification. `https://golang.org/ref/spec`. Accessed: 2021-02-02.

[2] Kotlin channels. `https://kotlinlang.org/docs/reference/coroutines/channels.html`. Accessed: 2021-02-02.

[3] Quasar library overview. `https://docs.paralleluniverse.co/quasar/`. Accessed: 2021-02-02.

[4] Ahn, S., Gorlatova, M., and Chiang, M. Leveraging fog and cloud computing for efficient computational offloading. In *2017 IEEE MIT Undergraduate Research Technology Conference (URTC)* (2017), pp. 1–4.

[5] Atzori, L., Iera, A., and Morabito, G. The internet of things: A survey. *Computer Networks 54*, 15 (2010), 2787–2805.

[6] Barry, R. *FreeRTOS reference manual: API functions and configuration options*. Real Time Engineers Limited, 2009.

[7] Bonomi, F., Milito, R., Zhu, J., and Addepalli, S. Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing* (New York, NY, USA, 2012), MCC '12, Association for Computing Machinery, p. 13–16.

[8] Bruneton, E. *ASM 4.0 A Java bytecode engineering library*, 9 2011.

[9] Cramer, T., Friedman, R., Miller, T., Seberger, D., Wilson, R., and Wolczko, M. Compiling java just in time. *IEEE Micro 17*, 3 (1997), 36–43.

[10] Feiler, P. H., Gluch, D. P., and Hudak, J. J. The architecture analysis & design language (aadl): An introduction. Tech. rep., Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst, 2006.

[11] Garcia Lopez, P., Montresor, A., Epema, D., Datta, A., Higashino, T., Iamnitchi, A., Barcellos, M., Felber, P., and Riviere, E. Edge-centric computing: Vision and challenges, 2015.

[12] Grogan, M. Jsr-223 scripting for the java tm platform. sun microsystems. *Inc., Santa Clara, USA, final edn* (2006).

[13] Grune, D., Van Reeuwijk, K., Bal, H. E., Jacobs, C. J., and Langendoen, K. *Modern compiler design*. Springer Science & Business Media, 2012.

[14] Günther, S., and Cleenewerck, T. Design principles for internal domain-specific languages: A pattern catalog illustrated by ruby. In *Proceedings of the 17th Conference on Pattern Languages of Programs* (New York, NY, USA, 2010), PLOP '10, ACM, pp. 3:1–3:35.

[15] Hammond, K., and Michaelson, G. Hume: A domain-specific language for real-time embedded systems. vol. 2830, pp. 37–56.

[16] Hickey, P. C., Pike, L., Elliott, T., Bielman, J., and Launchbury, J. Building embedded systems with embedded dsls. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming* (2014), pp. 3–9.

[17] Ivanova, V., Sedov, B., Sheynin, Y., and Syschikov, A. Domain-specific languages for embedded systems portable software development. In *Proceedings of 16th Conference of Open Innovations Association FRUCT* (2014), IEEE, pp. 24–30.

[18] Kahani, N. Automodel: a domain-specific language for automatic modeling of real-time embedded systems. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)* (2018), IEEE, pp. 515–517.

[19] Klikovits, S., Linard, A., and Buchs, D. Crest-a dsl for reactive cyber-physical systems. In *International Conference on System Analysis and Modeling* (2018), Springer, pp. 29–45.

[20] Levine, J. R., Mason, J., Levine, J. R., Mason, T., Brown, D., Levine, J. R., and Levine, P. *Lex & yacc.* " O'Reilly Media, Inc.", 1992.

[21] Margolis, J., Oh, T. T., Jadhav, S., Kim, Y. H., and Kim, J. N. An in-depth analysis of the mirai botnet. In *2017 International Conference on Software Security and Assurance (ICSSA)* (2017), pp. 6–12.

[22] Moreira, T. G., Wehrmeister, M. A., Pereira, C. E., Petin, J.-F., and Levrat, E. Automatic code generation for embedded systems: From uml specifications to vhdl code. In *2010 8th IEEE International Conference on Industrial Informatics* (2010), IEEE, pp. 1085–1090.

[23] Parr, T. J., and Quong, R. W. Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience 25*, 7 (1995), 789–810.

[24] Pradhan, S. M., Dubey, A., Gokhale, A., and Lehofer, M. Chariot: A domain specific language for extensible cyber-physical systems. In *Proceedings of*

the *Workshop on Domain-Specific Modeling* (New York, NY, USA, 2015), DSM 2015, Association for Computing Machinery, p. 9–16.

[25] Rumbaugh, J., Jacobson, I., and Booch, G. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[26] Santos, A., Cardoso, J., Diniz, P., Ferreira, D., and Petrov, Z. A dsl for specifying run-time adaptations for embedded systems: an application to vehicle stereo navigation. *The Journal of Supercomputing 70* (12 2014).

[27] Sifakis, J. A vision for computer science—the system perspective. *Central European Journal of Computer Science 1*, ARTICLE (2011), 108–116.

[28] Stevenson, A., and MacDonald, S. Smart proxies in java rmi with dynamic aspect-oriented programming. In *2008 IEEE International Symposium on Parallel and Distributed Processing* (2008), IEEE, pp. 1–6.

[29] Wilbur, S. R., and Bacarisse, B. Building distributed systems with remote procedure call. *Software Engineering Journal 2* (1987), 148–159.

[30] Wood, B., and Azim, A. Triton: a domain specific language for cyber-physical systems. In *IEEE International Conference on Industrial Technology* (2021).

[31] Wood, B., Watling, B., Winn, Z., Messiha, D., Mahmoud, Q. H., and Azim, A. Remote method delegation: a platform for grid computing. *Journal of Grid Computing 18*, 4 (2020), 711–725.

[32] Yi, S., Li, C., and Li, Q. A survey of fog computing: Concepts, applications and issues. In *Proceedings of the 2015 Workshop on Mobile Big Data* (New York, NY, USA, 2015), Mobidata '15, Association for Computing Machinery, p. 37–42.