ProcessJ: The JVMCSP Code Generator

Oswaldo Benjamin Cisneros Merino
benjcisneros@gmail.com

PROCESSJ: THE JVMCSP CODE GENERATOR

by

Benjamin Cisneros Merino

Bachelor of Science (B.Sc.)
University of Nevada, Las Vegas
2017

A thesis submitted in partial fulfillment of
the requirements for the

Master of Science in Computer Science

Department of Computer Science
Howard R. Hughes College of Engineering
The Graduate College

University of Nevada, Las Vegas
May 2019

Thesis Approval

The Graduate College
The University of Nevada, Las Vegas

May 16, 2019

This thesis prepared by

Benjamin Cisneros Merino

entitled

PROCESSJ: the JVMCSP Code Generator

is approved in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science
Department of Computer Science

Abstract

We as a society have achieved greatness because we work together. There is power in numbers. However, when it comes to programming we have not been able to achieve the same level of symbiosis. This is because concurrent programming has been stigmatized as an advance and abstract subject allegedly harder than sequential programming. Additionally, traditional approaches to solving concurrent problems using sequential programming become unnecessarily difficult because most of what newcomers are taught when it comes to concurrent programming (e.g., message passing and threads), while being technically correct, is completely irrelevant to the problems at hand. Rather than examining the preconceived notions of the problem, we stubbornly try to fix it using thread-and-lock models or non-shared memory and message passing models, making reasoning about the concurrent behavior of the problem extremely complicated if at all even possible.

Exploiting threads effectively depends on the concurrency model supported by the programming language being used. What is also needed is fine-grained parallelism without the explicit use of locks and without asynchronicity so programs can be made easy to reason about. I believe that ProcessJ can be the programming language that provides a bridge from todays languages to tomorrows concurrent programs. This thesis introduces ProcessJ, a new process-oriented language with Java-like syntax and CSP-based communication that uses synchronous channels. ProcessJ is cooperatively scheduled, runs on the Java Virtual Machine (JVM), and allows hundreds of millions of concurrent processes on a single core. Next, I describe its implementation and features. Following this, I explain the translation scheme of ProcessJ source code to Java, and how the generated code is used to create processes that correctly cooperate in scheduling without using the `Thread` or `Runnable` Java classes.

Acknowledgements

"I would like to express my deepest appreciation to my uncles *Connie* and *Randall*, and *Kevin* and *Robert*. Thank you for giving me a home and strength in moments when I thought I could not go on. I would not have made it this far without your help.

My sincere appreciation also goes to my advisor, *Señor Dr. Pedersen*, who has been always helping me and encouraging me through out the master program. For challenging me and making me realize that I absolutely love compilers and programming languages (even at times when I want to pull my hair). For his patience, motivation, enthusiasm, immense knowledge, and guidance in writing this thesis (and for the **twenty** tacos he promised me).

Besides my advisor, I would like to thank the rest of my thesis committee: *Dr. Taghva*, *Dr. Gewali*, and *Dr. Regentova*.

I am thankful to everyone at the Graduate College; in particular to *Janine*, *Kara*, *Brianne* (my favorite person), *Leslie*, *Maulik* and *Payam* (for being the only ones whom I could talk to about football – the real football), and *Jennifer* (the GA who was favorited more than me, who taught me all that I know, and who convinced Trey not to give me a tour).

At last but not least, the most important people in my life, my immediate family; especially *my mum* for supporting me spiritually throughout my academic and non-academic life – love you mum!"

BENJAMIN CISNEROS MERINO

University of Nevada, Las Vegas

May 2019

Table of Contents

List of Tables

List of Figures

List of Listings

List of Grammars

Chapter 1

Introduction

Since the 1970's transistors in processors have been doubling roughly every 2 years. This is known as Moore's law [91] which is not an actual law but an axiom based on observation; an axiom which is now reaching its limit. The size of transistors are currently 7nm and the next step, 5nm and below, posses a problem because of quantum physics. At the 5nm point and below a phenomenon known as quantum tunneling occurs. This means that the electrons in the transistors will jump from transistor to transistor making it impossible for them to have reliable on/off states.

Furthermore, Dennard Scaling [43, 48] has already come to an end. This is the principle that power needed to run transistors stays the same within a given area even as the number of transistors increases. The amount of energy required to run chips coupled with the close proximity of each transistor creates a huge heat concern. Without proper cooling, such as liquid nitrogen, the increase in heat threatens thermal runaway which would cause the chip to fail. This limits the potential for increasing clock speed past 5GHz without specialized cooling systems. We need new avenues to improve CPU performance and one such avenue is software based. As chip architecture has improved, software's ability to fully utilize those advancements has fallen behind. The 2019 line of consumer level CPU's come with 8 cores and 16 threads which is perfectly suited for concurrent and parallel programming. Meanwhile, programming these architectures to make good use of multiple cores available is difficult and prone to errors.

Concurrency is frequently misunderstood and mistaken for parallelism. However, concurrency and parallelism are not the same thing. An obvious example is when people say they are good at multitasking when they usually mean they are good at concurrency. Concurrency implies the scheduling of independent pieces of code to be executed in an interleave manner. Parallelism, on the other hand, implies executing 2 or more pieces of code at the same time on separate execution units. The latter can be achieved by using multi-core processors or multiple computers, whereas the former can occur in a time-shared manner on a single-core processor. An analogy of ordering a latte at a coffee shop can be used to explain the difference between them (Figure 1.1). Serving the orders requires multiple tasks such as taking an order, grinding the coffee beans, preparing the milk, steaming the milk, combining the steamed milk with the shot of espresso, and serving the latte. I can then switch back and forth between tasks by when they need to be done for serving multiple customers. While serving a latte, I can achieve parallelism if I ask someone to help me with these tasks. I can prepare latte for customer A, while somebody else can serve customer B. Another example is talking on the phone while driving. I am physically doing both tasks at the same time and is therefore parallelism; however, texting and driving would be concurrency because I have to look away from the road to text for a moment and then switch back to driving by looking at the road repeatedly in order to avoid crashing – don't text and drive.

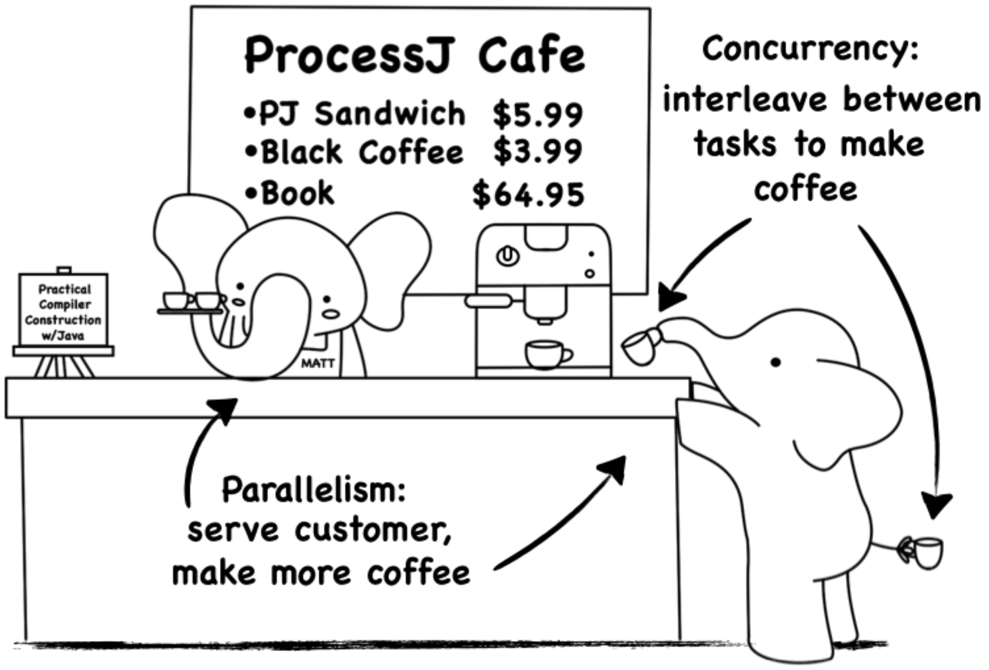
ProcessJ Cafe
•PJ Sandwich $5.99
•Black Coffee $3.99
•Book $64.95
Concurrency: interleave between tasks to make coffee
Practical Compiler Construction w/Java
MATT
Parallelism: serve customer, make more coffee

Figure 1.1: An example of concurrency and parallelism.

Unfortunately, parallelism and concurrency are notoriously difficult programming challenges. Examples of what can go wrong include deadlocks, livelocks, data race hazards, and process starvation. Despite using conventional techniques such as locks, mutexes, and semaphores, along with various design patterns to avoid these problems, programmers often resort to all kinds of ugly hacks because of their sequential programming assumptions and intuition. The result is typically an excessively complicated, restricted, and unreliable program prone to deadlocks and unpredictable race hazards; not to mention unexplained crashes, undefined behavior, low performance, and faulty results. In addition, while programs may appear to be fine during testing, they might deadlock or livelock when scheduled differently or scaled up. Debugging such programs is hard due to the possibly rare race conditions and non-determinism. Hence, it becomes almost impossible to reason about or prove properties of lock-based programs.

Despite all these difficulties, the full utilization of computing resources requires parallelism and concurrency. I believe in a model that is more appropriate and safer, where no locks, mutexes, semaphores, etc. are required. A model for dealing with parallelism and concurrency that has a strong mathematical foundation, verification and formal model checking, and is highly efficient and easy to learn. CSP-based languages – known as process-oriented – fulfill this need. Communication Sequential Processes (CSP) [63] is a process algebra for modeling complex interactions between concurrent components. Further still, it enables us to specify and verify the behaviors of the components involved at each program stage. We can therefore prove things about the code we write even without running it – we can actually check the behavior of our code using a model checker – allowing it to be used with confidence.

Even though languages like occam [69]/occam-π [23, 108] exists and provide the standard for CSP implementation, their development has stalled. For example, occam-π only supports linux OS 32-bit architecture. This is where we come in. ProcessJ is a new process-oriented programming language featuring CSP-based communication semantics using synchronous channels, a Java-like syntax, and a multi-threaded JVM-based runtime with a cooperative scheduler. The reason for the development of ProcessJ is mainly to promote process-oriented programming. I believe that process orientation is the best way to write reliable fine grained and coarse grained parallel code; especially when dealing with multi-core architectures, which are becoming increasingly popular. In this thesis I report on the development of ProcessJ's runtime system and code generation. I explain the translation scheme of ProcessJ source code to Java, and illustrate how the compiler generates code – by ways of producing Java source that after compilation is rewritten to integrate yield and resume points – for processes that correctly cooperate in scheduling on the JVM.

1.1 Motivation

The motivation for this thesis came about from the substantial amount of work that has been done to the ProcessJ programming language and its compiler since 2009. At the time of writing, ProcessJ provides a functional implementation. However, the code generation of the ProcessJ compiler is currently coupled with a number of bugs that required fixing. The work presented within this thesis contributes in the rewriting and improvement of the ProcessJ compiler. I re-engineered an existing experimental version of the ProcessJ compiler. In particular, I focused on the usability of the front end (the CLI), and I reimplemented the code generator (the back end) as it was faulty and somewhat poorly put together.

The groundwork for this thesis is divided into two parts. First, since many programming languages have built-in tools for parsing command line arguments, I believe ProcessJ should be no different. The idea behind ProcessJ's command line interpreter is to make the process of writing command line options effortless while allowing rapid customization when needed. Second, fixing and improving the generated Java code was the most important step of this thesis. It required rewriting the visitor pattern implementation and defining a proper walking procedure for traversing the syntax tree. This makes it possible for the compiler to correctly convert the syntax tree into the target code after pushing the intermediate data generated by a visitor object into a template that translates ProcessJ source code to Java code.

Figure 1.2 illustrates the ProcessJ system. The script file called `pjc` takes the arguments specified on the command line and hands them over to the command line interpreter (CLI). The command line interpreter portion of the compiler has four main responsibilities: parse the command line value, validate input, determine the source file, and hand over control to the ProcessJ compiler. The ProcessJ compiler reads `.pj` files and produces `.java` files which are then compiled with a standard Java compiler to produce class files. The compiler is divided into two parts: the front end and back end. The front end analyzes a ProcessJ source program and creates an intermediate Java representation, from which the back end generates a Java source file using the **StringTemplate** library [77]. From this Java source file, the Java compiler generates class files. The class files are further compiled and then instrumented by the **Instrumenter** program, which uses the **ASM** [38] tool to create code that can work with a cooperative scheduler. The rewritten class files are then packaged with the ProcessJ runtime elements to form an executable `.jar`. The script file called `pj` runs this executable on the JVM as long as the libraries are installed in the correct locations.

1.2 Thesis Outline

The organization of this thesis is as follows: Chapter 2 presents a detailed introduction to the background concepts necessary to understand process-oriented programming in ProcessJ, including the CSP model of parallel computing. Chapter 3 provides a brief description of the various approaches to concurrency and parallel programming. Chapter 4 gives a comprehensive overview of the command line interpreter and an example that demonstrates how it works. In Chapter 5, I present the background work done for the ProcessJ compiler, that is, the implementation details of the runtime components and code generator. Chapter 6 presents two test programs. This is followed by a conclusion in Chapter 7. Finally, the possibilities of improving and extending the compiler are presented in Chapter 8.

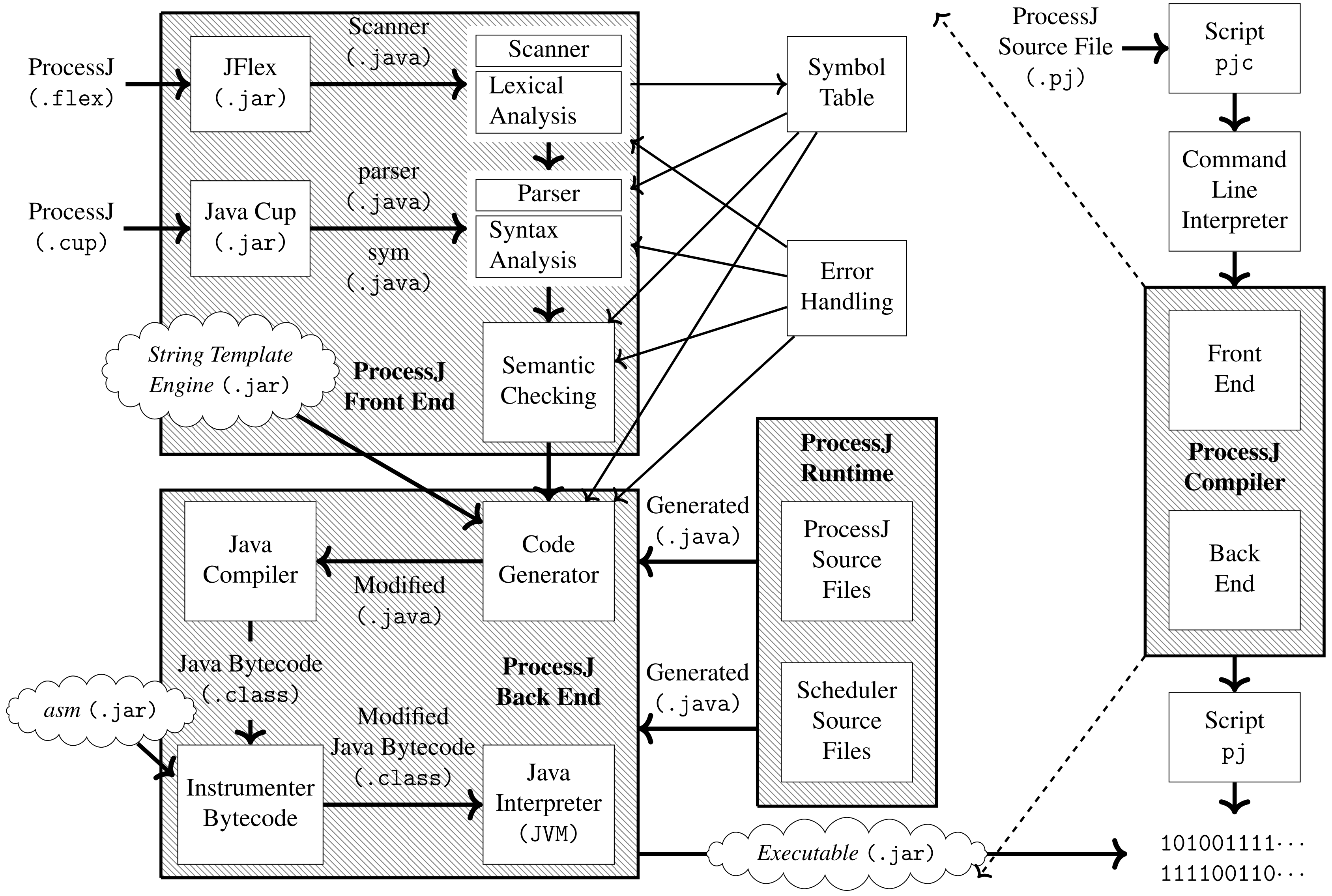

Figure 1.2: Parts of the ProcessJ compiler.

1.3 ProcessJ File Structure

Because at UNLV we believe engaging Reproducible Research is key to cummulative science [45, 44], ProcessJ's source code is publicly hosted on GitHub[1] and is available for anyone to contribute or download it. It will also be available in the near future as a Docker Container. The file structure of the ProcessJ compiler is shown in Figure 1.3. Note that as we continue to improve the language, these files or directories may change.

```
processj/+---- pjc                                    -- install script
         +---- pj                                     -- run script
         +---- src/+---- PJMain.java                  -- command line options
                   +---- ProcesJc.java                -- processj compiler
                   +---- ast/                         -- parse tree node hierarchy
                   +---- cli/                         -- command line interpreter
                   +---- codegeneratorjava/           -- java code generator tool
                   +---- instrument/                  -- java byte code tool
                   +---- library/
                   +---- namechecker/
                   +---- parallel_usage_check/
                   +---- parser/
                   +---- printers/
                   +---- processj/
                   +---- reachability/
                   +---- scanner/
                   +---- typechecker/
                   +---- utilities/
                   +---- yield/
         +---- lib/+---- JVM/                         -- library files
         +---- resources/+---- jars/                  -- processj components
                         +---- properties/            -- configurable parameters
                         +---- stringtemplates/       -- processj template engine
         +---- include/+---- JVM/                     -- include files
```

Figure 1.3: The ProcessJ compiler file structure.

[1]The project's home is at https://github.com/mattunlv

Chapter 2

Background

Most real world situations occur concurrently, and yet the computer tools that we have available today are not very good at expressing concurrency. If we look around, what we are going to see is a complex world where many things are happening concurrently at any given time – from traffic control, to banking and airline tickets, to even trivial activities that take place in our physical world. For example, we expect to do more than one activity at a time when doing our laundry. This simple task can be broken down into a set of simpler tasks that can be ran concurrently; we take the clothes out of the washer, insert them into the dryer, load the washer with more clothes, and run both simultaneously. Naturally, we describe the behavior of real-world objects through their interactions with other objects: take the output of one program (clean-wet clothes from the washer) and send it as the input to another program (clean-wet clothes to the dyer).

Concurrency provides a natural decomposition of real world applications. It is a way to write software code that can clearly express real world situations. With concurrency, we structure software in such a way that we can write code that is easy to use, easy to understand, and, most importantly, easy to reason about. Sequential processing alone, on the other hand, does not model the behavior of our physical world accurately. To achieve that kind of behavior, programmers often rely on locks, mutexes, and semaphores to coordinate the activities of objects produced by the sequential components in a concurrent program. Naturally, this almost always requires various programming techniques and development to correctly coordinate access to shared mutable data. Therefore, when writing parallel applications using conventional procedural programming languages (with object oriented features), it is common for programmers to step back and devise an entirely new algorithm to avoid data race hazards, as well as unplanned non-determinism.

Unfortunately, many still believe that our world can be described in terms of *abstraction*, *encapsulation*, *inheritance*, and *polymorphism* – the four pillars of Object-Oriented Programming (OOP). The reason for this widespread belief is the naïve idea that objects can help deal with many different kinds of systems that have complex behaviors. However, there are limits to the amount of complexity that programmers can handle using OOP; one, of course, being concurrency (multi-threading). In OOP, an object that exists in the real world is modeled by its fields and methods, and the interaction between them [67]. Programmers therefore think of a program as a collection of interacting objects. Objects, however, provide an artificial sense of security. That is to say, since objects are *passive*, they are not in charge of executing their methods in their own thread of control. As a result, a thread can leave the state of an object inconsistent [75]. This minor issue leads to larger problems: data race hazards, deadlocks, livelock, and starvation. In addition, it is difficult to visualize the control flow of complex systems using OOP features. This is most notably true for systems with a lot of inheritance that use abstract interfaces and have no strong typing. Naturally, not all systems can be modeled accurately by (passive) objects, and when they can they do not behave like the real world counterparts they represent.

Process-oriented programming (POP) is a programming paradigm that aims to simplify concurrency and parallelism based on Hoare's Communication Sequential Process Calculus [63]. POP is far more similar to the way the real world works. It promotes the use of independent processes (objects) that interact with each other in regular or chaotic patterns, and at all levels of scale. Contrary to OOP's view of the world, in POP, a concurrent system is modeled by the interaction between individual processes. These processes interact with one another by communicating through synchronized channels, or through multi-way synchronization points. This means that programmers can define larger and larger concurrent systems without worrying about undesirable nondeterminism and/or unforeseen side effects. While concurrency in various programming languages is made difficult due to the minute subtleties required to implement correct access to shared data, ProcessJ promotes a different approach in which shared values are passed around along channels – in fact, these variables are never shared by different threads of execution. I begin this chapter with an introduction to Communication Sequential Processes (CSP) before explaining the design and development of PocessJ using CSP-like constructs.

2.1 Communicating Sequential Processes (CSP)

Communicating Sequential Processes (CSP) [12, 41, 63, 64, 88] is a formal process algebra introduced by Hoare in 1978 to describe concurrent systems using processes[1] and events. It is a paradigm for expressing concurrency on the basis of message passing without sharing data (memory variables), and for solving the problem of coordinating synchronous (rendezvous) communication between processes. The idea behind CSP, in particular, is that a program consists of multiple processes that communicate with each other, each process represents the execution of a sequential program, and the simplest form of interaction is done by communicating, or passing, messages synchronously along channels.

While processes are composable (e.g., to create a network of processes, they combine and connect to each other), they perform internal activities in isolation. This means that other processes cannot see these activities unless they interact (engage in events) with the outside world. A process must therefore place (write) some data in the message that it wishes to send for other processes to see (read) and assign the data in the message to a local variable. A receiving process may choose to ignore the data in the message and only be concerned with the fact that a message has arrived. In a situation such as this, the message received can be regarded as a 'signal' indicating that communication was established. Such behavior can be used to simulate phone calls using processes to represent telephones with recording machines. For example, for a phone conversation to take place, both the caller (sender) and the callee (receiver) must be present because, otherwise, no one would pick up the phone. When a phone call is made, the sender waits until the receiver accepts the call. If the receiving process accepts the call, then communication is established. As a result, any message placed in the 'phone line' will be received. However, if instead the recording machine answers the phone call, depending on whether the machine is able to take a message, the message may or may not be ignored.

Given that the behavior of a process is described through its interaction with other processes, the outside world can neither see the process's data or execute its algorithms. It is of note that,

[1]Note, in contrast to the term used in operating systems, the term *process* in CSP is used to describe an entity that is part of a complex system. Such a complex system consist of less complex components, which in term consist of simple components that operate sequentially inside the entire system.

although values can be handed off to other processes, the algorithms are executed by the process itself in its own thread of control. Unlike objects in OOP whose (protected or private) data is at the mercy of any thread that sees it [21], a process can change its values only when it wishes to do so.

CSP is built on two basic primitives: *processes* and *events*. A process is an ordered sequence of operations which represent a behavior in a system. For example, the simplest form of behavior is stop – the process does not engage in any event and will not terminate – and skip – the process does not engage in any event but will terminate. Furthermore, since there is no notion of data (which is the idea that when a process gives a value to another process, it no longer has it), a process is abstractly described by the event or sequence of events that it produces. Events, on the other hand, are the way in which processes communicate or interact, and they may occur instantaneously when executed by the system. Each process sequentially produces events that are atomic, synchronous, and instantaneous, and that can involve other processes. Naturally, if an event involves multiple processes, the processes will block until they are all engaged in the event. These processes will then carry out such an event simultaneously.

CSP processes are developed from events and other processes using a number of operators. In order to understand CSP, let us briefly look at the notation used to describe and analyze systems consisting of processes. In particular, let us look at the alphabet and processes, and communication.

2.1.1 Alphabet and Processes

In CSP, a process is the behavior of an object described with its alphabet. Each process has an alphabet defined as a set of names of all events that may be relevant to the process itself. This is denoted by $\alpha \mathrm{P} = \{\ event_1,\ event_2,\ event_3,\ \ldots,\ event_n\ \}$, where αP describes all the possible interactions with process P[2]. For example, both an ATM[3] and a PERSON[3] can be modeled as processes with a *card* event and a *money* event. These processes can therefore have an alphabet similar to this:

$$\alpha \mathrm{ATM} = \{\ card,\ money\ \}$$
$$\alpha \mathrm{PERSON} = \{\ card,\ money\ \}$$

[2]The letters P, Q, R are reserved for arbitrary processes.

[3]Words written in all-upper case letters defined specific processes.

Where *card* and *money* represent a set of **atomic** interactions of the system we are trying to model; *card* represents the action of inserting a card in an ATM card reader, and *money* represents the action of taking money from the ATM's dispenser. While these events may be atomic names (e.g., *card* and *money*), they may also be **compound names** (e.g., *dispenser.open*, *dispenser.close*) or **input/output** events (e.g., *mouse?xy*, *screen!bitmap*). Indeed, a process's alphabet is simply the set of actions it performs, we use the alphabet to create a system that can be modeled through a chain of events. We can then verify and prove deadlock and livelock freedom based on the basic assumptions made in said system using tools such as the Failures-Divergences-Refinement model-checker (FDR) [14] together with the concept of *refinement*.

Although we normally specify an alphabet for each process (referred to as αP in the above-mentioned process P), I will leave this notation implicit and simply refer to the process by its name for the rest of this section.

Prefix Operator ($\rightarrow$)

The prefix operator, denoted by $a \rightarrow$ P, is used to engage a process in an event. For example, the process P $= a \rightarrow$ skip performs event a then behaves as skip. In other words, the process engages in event a and then ends (it does nothing but terminates successfully). Naturally, ending a definition of a process with the definition of another process enable us to create processes that only engage in a finite sequence of events. To create an endless sequence of events in CSP, we use recursion for repeated tasks. For example, the process P $= a \rightarrow$ P describes a *repetitive* behavior for process P. Therefore, by having processes recurse on themselves, we can create processes that communicate forever.

External Choice ($\Box$)

The external choice operator, denoted by R $\Box$ Q, is used to execute a choice between R or Q. For example, the process P $= (a \rightarrow \text{R}) \Box (b \rightarrow \text{Q})$ chooses one event. This means that if only a is available then P behaves as $a \rightarrow$ R, or if only b is available then P behaves as $b \rightarrow$ Q. However, if both events are available, then one is arbitrarily chosen depending on the actions offered by the environment. As another example, let us consider the following process: P $= (a \rightarrow \text{R}) \Box (a \rightarrow \text{Q})$. Notice that both events start with an a, which can lead to a non-deterministic choice. CSP typically allows the environment to choose which event to take. However, in a situation like this, it will allow P to make the decision with no influence from the outside world whatsoever. Naturally, then, it should be cleared that P will consider this choice internally.

Internal Choice ($\sqcap$)

The internal choice operator, denoted by R $\sqcap$ Q, is used to execute an arbitrary choice between R or Q. For example, the process $\text{P} = (a \rightarrow \text{R}) \sqcap (b \rightarrow \text{Q})$ behaves as either $a \rightarrow \text{R}$ or $b \rightarrow \text{Q}$. However, the choice is decided by P itself internally without considering the external environment. In addition, this process may deadlock if the environment only provides an a when it decides to do a b instead. It should be pointed out that for the above-mentioned process $\text{P} = (a \rightarrow \text{R}) \square (a \rightarrow \text{Q})$, where the choice is made non-deterministically, this is the same process as $\text{P} = (a \rightarrow \text{R}) \sqcap (a \rightarrow \text{Q})$. Therefore, the choice between the two processes is entirely arbitrary and different from that of the $\square$ operator. That is to say, CSP will give either $a \rightarrow \text{R}$ or $a \rightarrow \text{Q}$, but without providing the external environment with a way for choosing which.

Parallel Composition ($\|$)

The parallel composition, denoted by R $\|$ Q, describes a process that behaves like a system in which R and Q are allowed to proceed concurrently and independently. However, they must agree to synchronize on all the common events. In other words, processes interact by handshake communication. For example, consider the following three processes:

$$\text{R} = a_1 \rightarrow b \rightarrow \mathsf{skip}$$
$$\text{Q} = a_2 \rightarrow b \rightarrow \mathsf{skip}$$
$$\text{P} = \text{R} \parallel \text{Q}$$

Process P behaves as either $a_1 \rightarrow a_2 \rightarrow b \rightarrow \mathsf{skip}$ or $a_2 \rightarrow a_1 \rightarrow b \rightarrow \mathsf{skip}$. Notice the interleaving on a_1 and a_2, and synchronization on b. This leads to the situation where R can engage in a_1 without synchronizing with Q. Similarly for Q on event a_2. It should therefore be cleared that the two possible execution traces[4] for P is the set $\{< a_1, a_2, b, \checkmark >, < a_2, a_1, b, \checkmark >\}$, where events a_1 and a_2 can occur in any order. In spite of that, both processes must synchronize

[4]A *trace* represents the sequence of all events a process may perform.

on the shared event (namely b if $b \in \alpha\mathrm{P}$) in order to make further progress, after which P should end. Note that the special event ✓ is used to indicate termination.

As another example, consider a slightly modified version of the processes described above:

$$\mathrm{R} = a \rightarrow b \rightarrow \mathsf{skip}$$
$$\mathrm{Q} = b \rightarrow a \rightarrow \mathsf{skip}$$
$$\mathrm{P} = \mathrm{R} \parallel \mathrm{Q}$$

If we assume that the only common event shared by R and Q is b (that is, $b \in \alpha\mathrm{R}$ and $b \in \alpha\mathrm{Q}$), then process P behaves as $a \rightarrow b \rightarrow a \rightarrow \mathsf{skip}$. In contrast with our first example, in order for R to interact with Q, they must both agree on an event, and on the condition that R must initially engage in event a before both processes can engage in event b. Once R does so, to make further progress, it must synchronize with Q on b, after which Q can engage in a. This means that P has a traces set $\{< a, b, \checkmark >, < b, a, \checkmark >\}$, where events a and b must occur in that order.

Sequential Composition (;)

The sequential composition, denoted by P ; Q, is used to execute P and Q sequentially. For example, the process P ; Q behaves as P first until P terminates, then behaves as Q until Q terminates, and finally behaves like skip (which again represents a successful termination) when the behavior of the process is required to continue. Naturally, if P never terminates then neither can Q. It should be mentioned that, unlike parallel composition, sequential composition operates only on processes and not events; therefore, both P and Q must also be sequential processes for the ; operator to work.

2.1.2 Communication

A process will often want to transmit data to another process; fortunately, CSP enables events (that conceptually consists of channel names) to carry zero or more data components. Communication in CSP is an event described by a pair $c.v$, with c denoting the name of a *channel* and v denoting the *component* sent across this channel by the outputting process, where each pair of $c.v$ represents a different event. A channel is a special type of event that is (usually) part of the alphabet of two processes: the sender and the receiver. Naturally, to enable two processes to communicate over a channel, both processes (whose alphabets include the channel) must first engage in it. Furthermore, we define the set of data which a process can communicate on a channel (say) c as $\{ c.v \mid v \in Values \}$ for an event that inputs and outputs values of type *Values*. We can then use input ($c?x$) and output ($c!v$) operations to allow communication between two processes. These input and output operations, however, are simply a shorthand resulting from the following identities:

$$c!42 \rightarrow \mathsf{skip} \equiv c.42 \rightarrow \mathsf{skip}$$
$$c?x \rightarrow \mathsf{skip} \equiv (c.42 \,\square\, \square_{v \in V \setminus \{42\}}\, c.v) \rightarrow \mathsf{skip}$$

Consider the following three processes:

$$\mathrm{R} = c!42 \rightarrow \mathsf{skip}$$
$$\mathrm{Q} = c?x \rightarrow \mathsf{skip}$$
$$\mathrm{P} = \mathrm{R} \parallel \mathrm{Q}$$

The above example shows that the resulting combined process P outputs the value 42 from R, which is supplied via channel c to Q and then bound to (some variable) x. In order words, if R engages in $c!42$ and Q engages in $c?x$, then communication will be established and the value 42 will be passed along the channel c. It should be mentioned that the channel used in the above example assumes *rendezvous behavior*. This means that the sender and receiver will block any operation on the channel until the message is transmitted. Therefore, data can be safely exchanged between R and Q.

While the syntax of CSP (of which there is much more to say) cleanly reflects an intuitive way to describe the behavior of objects in terms of events and other processes, I will end the CSP crash course as I have covered the concepts necessary to understand the process-oriented concurrency primitives (mainly process, channels, barriers, and alternations) with their respective CSP semantics used in ProcessJ.

2.2 Communicating Sequential Processes in ProcessJ

ProcessJ [79, 80, 81, 82, 94, 95, 98, 99] has been under development at the University of Nevada, Las Vegas (UNLV) since 2009. It is intended both as a programming tool for research in concurrent and parallel programming, and as a programming language used for the introduction of CSP to UNLV Computer Science students.

ProcessJ is based on CSP process algebra [63, 64] and the π-calculus [71, 70]. It is a general purpose process-oriented programming language; semantically it is close to occam-π, but with a syntax similar to Java (without objects) and with added constructs for CSP-based communication using synchronous typed channels. It has a dynamic concurrency model built into its core design, which is deterministic by default. Such model can be expressed explicitly using the 'par' construct for parallel composition. However, a special feature called 'alternation' (e.g, a *choice* to choose from among many other alternatives) can be used to introduce non-determinism to represent parallel computations. As a result, non-deterministic systems can be built when necessary, regardless of the external context within which they are executed, simply by making an external choice available through alternations.

ProcessJ supports all the basic types (see Appendix A) and control flow structures of Java; for example, a for-loop statement in ProcessJ is the same as in Java. In addition, while the new process-oriented concurrency primitives – process, channels, barriers, and alternations – have a familiar Java syntax feel, their semantics directly reflect those of occam/occam-π: based on the CSP concept of concurrent process synchronization, along with features that allow dynamic process creation and process mobility. This simply means the following: that 1) with these relatively simple primitives, ProcessJ allows programmers to write quite powerful parallel and distributed programs that are easy to reason about with respect to the interactions between concurrent components; and 2) a formal model-checker tool like FDR [14] can be used to statically prove vital properties of such programs; typically, this is used to ensure the absence of deadlock, or refinement of a specification.

As a process-oriented language, ProcessJ is composed of processes, each of which is executed in its own context. This means that every process has its own private state in which no other process, except the owner, has access to it. Since there is nothing to share, there is nothing to fight for. Having no mutable shared data between processes means there is no need for locking mechanisms. As a consequence, ProcessJ programs cannot have race conditions. In addition, there is no need for complex logic either. Having message passing for communication among processes, inherently, avoids the risks of race hazards (even when processes are forming dynamically). This allows the ability to create complex networks of communicating processes, which can be used to model non-trivial real world applications.

2.2.1 Scheduler

Like any other CSP-based programming language, ProcessJ uses cooperative scheduling; that is, Java threads are not used as a unit of concurrency. Recall that in a cooperative scheduler (also known as non-preemptive), a process does not stop until it decides to do so voluntarily [66]. To enable all parts of a system to progress fairly, our process model must support and correctly implement *explicit yielding* [82, 94]. Since our scheduler itself cannot decide when a process should yield, the process must explicitly give up the CPU for other processes to get run. Such yielding must happen at synchronizations points like channel communication, barrier synchronizations, alts, timer timeouts, and par blocks. This is because processes that either receive or deliver information may block. When this happens, our runtime scheduler is left with no alternative but to schedule other processes, including those whose actions (which again receive or deliver information) will unblock the first set of processes.

ProcessJ's scheduler is extraordinarily simple. Whether processes are executed by a single or multi-threaded scheduler, it does the basic operations a cooperative scheduler requires: take a process out of the queue for as long as it has processes. If the process is ready to run, let it run until it either terminates or yields. When the process yields or is not-ready to run, put it back in the queue. Repeat the same steps until the queue has no processes. A pseudo-code snippet for such a scheduler can be seen in Figure 2.1.

2.2.2 Process Mobility and Resumption

While processes can yield mid-execution and then resume, be disconnected from their local environment and then be reconnected to it, or be moved (by communication over a channel) to some environment, their state must be saved and restored. In ProcessJ, a mobile process is a process that stops running when it is explicitly suspended; this can be done with the `suspend` statement. When a process is in its suspended state (e.g., it is not executing), it becomes a piece of data that can be transmitted to another process over a channel. The suspended process can be resumed only by invoking it and giving it the proper parameters it needs. Although we can 'pause' a running process at any time, and resume it later without having to start it all over again, execution must continue on the line indicated after the statement that suspended the process and with the process's

```
Queue<Process> runQueue;
. . .
// enqueue one or more processes to run
. . .
while (!runQueue.isEmpty()) {
    Process p = runQueue.dequeue();
    if (p.ready())
        p.run();
        if (!p.terminated())
            runQueue.enqueue(p);
        else
            p.finalize();
    else
        runQueue.enqueue(p);
}
```

Figure 2.1: The pseudo-code for ProcessJ's cooperative scheduler.

local state unchanged.

Initially, in [82], one of the earliest experimental versions of ProcessJ, we used an **activation record**[5] as a means of preserving a process's state before it yielded. Every process invocation was linked to an activation record created on the fly and then stored, where each activation record contained information needed to manage a process; typically, this included the state of the parameters and local variables, and the links to other activation records. While this approach facilitated access to locals, as well as parameters – including variables from other activation records, it had performance implications when creating and maintaining activation records for the called process. For example, when a process invoked a procedure, and the callee (the called process) engaged in a synchronization event, the callee had to yield. This required saving all locals and parameters in two separate activation records (one for the caller and another for the callee) before the called process yielded.

[5]An array of Java *Objects* that hold values of locals and parameters while a process was in its suspended state.

```
public void foo(chan<int>.read in) {
    int d = in.read(); // synchronization event
    println("read: " + d); // procedure invocation
    ...
}

public void bar(chan<int>.read in) {
    ...
    foo(in); // process invocation
}

public void main(string[] args) {
    chan<int> c;
    bar(c.read);
    ...
}
```

Listing 2.1: Example of a process calling procedure.

In Listing 2.1, the *read*() in `foo` will yield, which means that `bar` must also yield. When `bar` is rescheduled to run, the flow of control will have to make it through `bar`, continue through `foo`, re-attempt the read in `foo`, and, finally, call `println`. Naturally, this demanded a high amount of bookkeeping. In particular when re-establishing variables from an activation record required placing the values of all locals and parameters into an *Object* array, and then placing this array in the activation record stack. As a consequence, this approach was later discarded in favor of a simpler approach using auto-generated code.

Currently, mobile processes are being implemented as a Java class [81, 99] in ProcessJ. Each mobile process becomes a class that extends the ProcessJ class `PJProcess`. This makes possible the mobility of processes to other computational environments when they change state (e.g., when processes suspend or resume). We approach the rewriting, that is, the way a process is represented by the abstract `PJProcess` class, in the following way: the ProcessJ compiler rewrites all local variables, including parameters, which form part of a procedure as fields. Replacing theses local and formal variables with fields is an easy way to maintain and preserve state when a process resumes execution. For example, when a process becomes active again (the process is restarted), all fields (originally locals or parameters) contain the same values as they did before. Naturally, we no longer have to worry about losing or correctly updating values because a process always carries its own data around. A code snippet of a rewrite example can be seen in Figure 2.2.

ProcessJ Code

```
public void foo(chan<int>.read r) {
    int d;
    d = r.read();
    println("read: " + d);
}
```

$\Rightarrow$

Java Code

```
public class foo extends PJProcess {
    PJChannel<Integer> pd$r1; // original r
    int ld$d1; // original d

    public foo(PJChannel<Integer> pd$r1) {
        this.pd$r1 = pd$r1;
    }

    @Override
    public synchronized void run() {
        switch (runLabel) {
        case 0: break;
        case 1: resume(1); break;
        case 2: resume(2); break;
        default: // runtime error
    }
    ...
}
```

Figure 2.2: A code snippet of a rewrite example for a mobile process `foo` in ProcessJ.

Note how a ProcessJ procedure (left) is translated into a Java class (right) that holds the data being transferred, namely, the locals and parameters of the procedure. Since a process is represented as an **object** created from the `PJProces` class, no complex trickery is needed to keep track of its variables when it yields or when it is woken up by the scheduler. When a write operation occurs, for example, data is retrieved directly from the process and then placed in the channel. Similarly for a read operation, when data is read from the reading end of a channel, it is directly stored in the process. Additionally, we no longer have to worry about activation records. A call to a procedure, whether it yields or not, is treated as a concurrent process wrapped in a par-block. This means that when a procedure is invoked, the caller yields. That is to say, the caller remains in a not ready state and waits until the called procedure (which is now a process) is finished. Once the called procedure is finished running, the scheduler will reschedule the caller so that it can run again. The caller will then continue execution on the line following the yield instruction.

2.2.3 Building Processes as a Network of Processes

In this section, an example of an ‘integrator’ process is used to demonstrate a complex network of independent objects all acting and interacting with each other. In ProcessJ, it is easy to write code that runs a number of processes concurrently. Consider three simple processes: `delta` which reads input from an input channel and writes the read value to two output channels (Listing 2.2); `plus` which reads input from two input channels, adds the read values together and writes the sum to the output channel (Listing 2.3); and `prefix` which takes (as a parameters) an initial value, an input and an output channel (Listing 2.4).

```
public void delta(chan<int>.read in,
                  chan<int>.write out1,
                  chan<int>.write out2) {
  while (true) {
    int x;
    x = in.read();   // read input from Plus
    par {
      out1.write(x); // write to output stream
      out2.write(x); // write to Prefix
    }
  }
}
```

Listing 2.2: The `delta` process.

```
public void plus(chan<int>.read in1,
                 chan<int>.read in2,
                 chan<int>.write out) {
  while (true) {
    int x1, x2, sum;
    par {
      x1 = in1.read(); // read from input stream
      x2 = in2.read(); // read from Prefix
    }
    sum = x1 + x2; // increment current-sum
    out.write(sum); // write to Delta
  }
}
```

Listing 2.3: The `plus` process.

```
public void prefix(int initVal,
                   chan<int>.read in,
                   chan<int>.write out) {
  out.write(initVal);
  while (true) {
    int x;
    x = in.read(); // read from Delta
    out.write(x); // write to Plus
  }
}
```

Listing 2.4: The `prefix` process.

The process network in Figure 2.3 shows how these three processes can be combined to form the `integrate` process, a process composed of the previous three using the `par` construct. Note that `a`, `b`, and `c` are internal channels to the `integrate` process, and `in` and `out` are input and output channels to the environment (Listing 2.5).

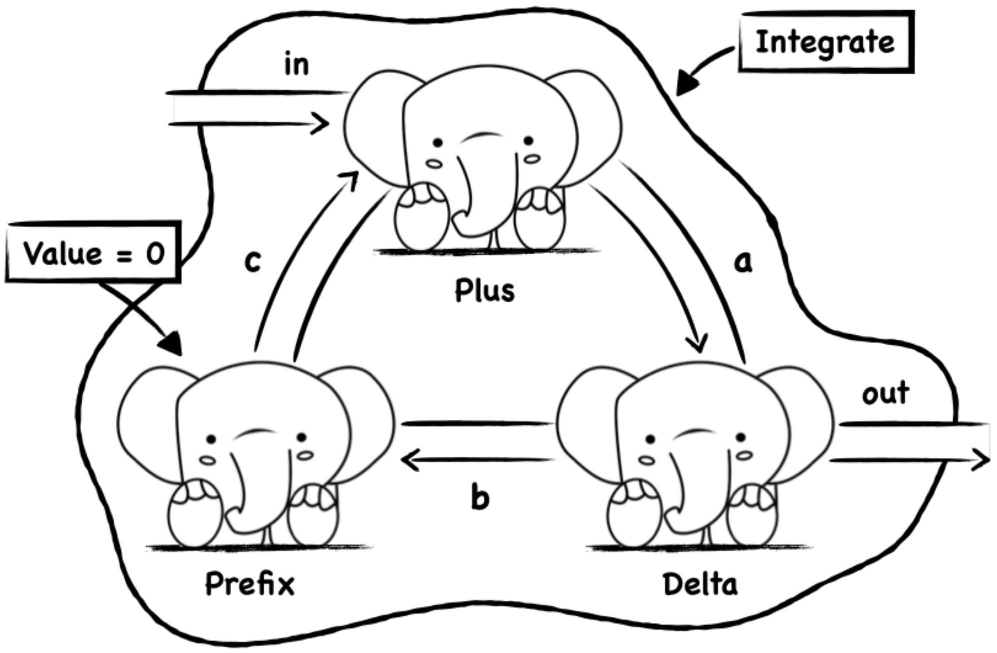

Figure 2.3: An implementation of an *integrator* process in ProcessJ.

```
public void integrate(chan<int>.read in,
                      chan<int>.write out) {
  chan<int> a,b,c;
  par {
    plus(in, c.read, a.write);
    prefix(0, b.read, c.write);
    delta(a.read, out, b.write);
  }
}
```

Listing 2.5: The `integrate` process.

Additionally, we can write a `producer` process (Listing 2.6) to produce numbers on the `in` channel, and a `consumer` process (Listing 2.7) to read values from the `out` channel, and then use them to create another (layer of) parallel composition of processes. Since we use channels for message passing, the `producer` and the `consumer` do not have to know about each other. Naturally, messages can be delivered in the order in which they are sent, making the system much easier to reason about.

```
public void producer(chan<int>.write out) {
  int x = 0;
  while (true) {
    out.write(x);
    x++;
  }
}

```

Listing 2.6: The `producer` process.

```
public void consumer(chan<int>.read in) {
  while (true) {
    int x;
    x = in.read();
    println(x);
  }
}

```

Listing 2.7: The `consumer` process.

When the main program (Listing 2.8) is run, the *producer*(), *consumer*, and *integrate*() are executed concurrently as processes inside the par-block; that is, they all run at the same time.

```
...
public void main(string args[]) {
  chan<int> in, out;
  par {
    producer(in.write);
    consumer(out.read);
    integrate(in.read, out.write);
  }
}
...
```

Listing 2.8: The `producer`, `consumer`, and `integrate` processes.

We should emphasize that the opposite of a ‘par’ (parallel) block is a ‘seq’ (sequential) block. We are familiar with sequential blocks from many programming languages. In Java, ignoring threading for now, all blocks are sequential. Recall, a sequential block consist of a number of statements within a set of { }. These statements are normally executed sequentially from top to bottom in the order that the code appears. This is true in ProcessJ as well. However, a sequential block may be prefixed with the `set` keyword.

```
...
public void main(string args[]) {
  chan<int> in, out;
  seq {
    par {
      producer(in.write);
      consumer(out.read);
      integrate(in.read, out.write);
    }
  }
}
...
```

Listing 2.9: Executing a sequence of statements in ProcessJ.

The program in Listing 2.9 is equivalent to the one in Listing 2.8. This programming style is an inherited option from the classical occam language, a language from the 80’s. Even though this program (Listing 2.9) is equally valid, the preferred way to execute sequential statements in ProcessJ is by using the sequence structure built into the language, namely the semicolon.

Determinism in ProcessJ

The above scenario shows how the `par` construct can be used to create a (complex) system from fine-grained components that do not retained any information. Further still, the outputs of this system depend only on the inputs it receives, regardless of the features of the runtime environment in which the system operates. In other words, whether the distribution of processes in this system runs on a single-core or multi-core machine, it is independent from the scheduling policies of the runtime environment (the JVM in this case). Consequently, the model of computation in this system is inherently deterministic.

Non-determinism in ProcessJ

While the above parallel system is deterministic, it runs forever. None of the processes terminate – clearly, due to their while-loop condition. In order to gracefully terminate the entire network, we need to introduce a channel on which we can send a 'kill' signal that will terminate the network of communicating processes; graceful termination, however, is not always a simple task. To do this in the correct order, the implementation of each component must forward a 'kill' signal before terminating. A 'killer' channel can be used to this effect (Listing 2.10). This channel will carry a boolean value that indicates whether or not a process should terminate, where `true` means 'forward the signal and terminate' and `false` means 'carry computation'. Of course, this requires making changes to the body of each procedure: `integrate`, `producer`, and `consumer`; including the `main` procedure (see Appendix D for the complete implementation).

```
public void killer(chan<boolean>.write killProduce) {
  timer t;
  t.timeout(3); // wait some time
  killProduce.write(true); // send kill signal
}
```

Listing 2.10: The `kill` process.

Note that there is still the possibility for deadlock if the `producer` gets stuck in its write call, and the `consumer` get stuck in its read call. The reason is that the 'kill' message may not be received at specific points of communication during the execution of the program. To avoid this, we need to safely distribute the signal to all processes, starting from the `killer` process. We let the `killer` process send a 'kill' signal to `producer`, let `producer` send the 'kill' signal to `integrate`, and, finally, let `integrate` send the 'kill' signal to `consumer`. Since we want a process to terminate, we need to make sure that a communication made on the 'killer' channel gets intercepted by the process. When this happens, the process should respond by sending the 'kill' signal before terminating itself.

Indeed, a 'kill' signal can terminate a process only when it is idling and waiting for a respond, how can we make the sequence in which this signal arrives non-deterministic? Let us suppose that a process has a number of alternatives to choose from as to what to do next. Furthermore, suppose that one of these alternatives causes the process to stop looping and terminate normally. What we need is a mechanism for alternating between these choices; such a primitive is incorporated into ProcessJ. An `alt` (or alternation) consist of a number of guarded statements. Each guard, in this instance, is a *channel-read expression* that when 'ready' represents the communication that can complete. When execution reaches an alt, all guards that are ready are marked and one is chosen at random. The process can then select one alternative from among the available channels (Listing 2.11).

```
25: public void producer(chan<int>.write out,
26:                      chan<boolean>.read killMe,
27:                      chan<boolean>.write killIntegrate) {
28:   int i = 0;
29:   boolean ok = true;
30:   while (ok) {
31:     boolean b;
32:     alt {
33:       b = killMe.read(): {
34:         ok = false;
35:         killIntegrate.write(true);
36:       }
37:       skip: {
38:         out.write(i);
39:         i=i+1;
40:       }
41:     }
42:   }
43: }
```

Listing 2.11: `producer` procedure that includes the kill channel.

All that is left to do is to add a 'killable' behavior. Such behavior will come from doing a read on the 'killer' channel when there is data available (see line 33 in Listing 2.11). Notice how the above code will not deadlock and will now terminate gracefully. This is because when `producer` engages in the communication made on the 'killer' channel, namely, `killMe`, it send the 'kill' signal before terminating itself. This signal is input by the `integrate` process, in parallel with the `consumer` process, which then outputs the signal to `consumer` before terminating itself. `consumer` then inputs the signal, and finally, terminates itself without outputting the current running-sum. It should be mentioned that while the choice itself is made arbitrarily in the above example, it can also be based on highest priority selection or fair selection.

2.2.4 CSP Primitives and Other Types

A ProcessJ program may contain any number of type declarations. At the top-level, ProcessJ supports five different declaration:

- Procedure declarations,
- Protocol declarations,
- Record declarations,
- Constant declarations,
- Extern type declarations

In this section I will give an overview of the ProcessJ language. Before I consider the five different top-level declarations, of which the first three are also type constructors, I will start with a short introduction to ProcessJ's atomic type system.

2.2.4.1 Primitive Types

ProcessJ supports all the typical primitive types known from most other programming languages. These include integral types of various sizes, floating point types, booleans, and strings. In ProcessJ a string is a primitive type (ProcessJ does not have the notion of objects, and since Java implements strings as an object, this seemed necessary); in addition, there are two more primitive types. The first is a `timer` type, which in fact is the reading end of a timing process that is always willing to engage in communication – but it may also be used as timeouts. The second is a `barrier` type, which is a special construct that defines a full barrier. Processes enrolled (using the keyword `enroll`, which can be prefixed by a `par` or a for-loop, if needed) on a barrier can synchronize on it by using the `sync` keyword.

2.2.4.2 Compilation Unit

The Extended Backus-Naur form (EBNF) grammar for a compilation unit is shown in Grammar 2.1.

$$\begin{array}{rcl} \textit{compilation\_unit} & \rightarrow & \textit{pragmas}* \ [\textit{package\_declaration}] \\ & & \textit{import\_declarations}* \ \textit{type\_declarations}* \end{array}$$

Grammar 2.1: Grammar for a compilation unit.

A ProcessJ file constitutes a compilation unit which forms the input for the ProcessJ compiler. A ProcessJ compilation unit (Listing 2.12) has the following structure:

```
[#pragma declaration(s);]

[package declaration(s);]

[import declaration(s);]

[top-level type declaration(s);]
```

Listing 2.12: ProcessJ compilation unit.

- An optional list of pragmas (of the form `pragma ...`). Pragmas are used to pass options to the compiler when compiling a specific file. Currently, pragmas are only utilized in the library generating system.
- An optional package declaration of the form `package <name>`. A package name should, like in Java, correspond to the directory path in which the file is located.
- An optional list of import statements of the form `import ...`. Importing a file allows us to use types and procedures (which are also types) from other compilation units.
- An optional list of type declarations. This includes the definition of records, protocols, and procedures.

2.2.4.3 Process

The EBNF grammar for a procedure declaration is shown in Grammar 2.2.

proc_type	→	*modifier*∗ **type** ID ([**type** ID **,** **type** ID)∗]) [*annotations*]
		[{ *statement*∗ }]
modifier	→	**public** | **private** | **native** | **const** | **mobile** | **protected**
annotations	→	[(ID **=** (ID | *boolean_literal* | *numeric_literal*))∗]

Grammar 2.2: Grammar for a procedure declaration.

A *procedure* is considered a type in ProcessJ. The reason is that procedures, when run, are considered **processes**. Given that processes describe the basic building blocks of any ProcessJ program, a program can be described as a collection of procedures, where each procedure executes concurrently and communicates with each other via channels. Since ProcessJ supports mobile processes, procedures can be communicated between other procedures as a passive piece of data.

In ProcessJ, the only elements required for a procedure[6] declaration are: a name, a return type, a pair of parentheses, and a body. Simply put, a procedure can have a number of modifiers, or none in which case it is assumed to be `protected` (see Appendix B for a list of available modifiers) – although in Java a procedure is assumed to have the `default` access modifier, such modifier does not exists in ProcessJ. Furthermore, all procedures must declare a return value type, or `void` if they do not return a value, a comma-delimited list of input parameters, preceded by their data types, enclosed by parentheses, and a body enclosed between braces.

Finally, all ProcessJ programs need a *main procedure* to run. This main procedure (Listing 2.13) should have the modifier `public`, and it should take in one argument, namely an array of strings.

[6]An annotation such as `[yield=true]`, which in early experimental versions of ProcessJ provided a way for the compiler to differentiate between a regular (Java) procedure and a process, is not longer required to appear after a procedure's enclosing parenthesis.

```
public void main(string args[]) {
  println("Hello World!");
}
```

Listing 2.13: The `main` procedure.

2.2.4.4 Channel

The EBNF grammar for a channel declaration is as shown in Grammar 2.3.

$$\textit{channel} \quad \rightarrow \quad [\ \textbf{shared}\ [\ \textbf{read} \mid \textbf{write}\ ]]\ \textbf{chan} < \textit{type} >$$

Grammar 2.3: Grammar for a channel declaration.

A *channel* is the simplest mechanism for coordinating and establishing communication among processes within an independent or cooperative system. In ProcessJ, channels are synchronous, uni-directional, and unbuffered. The basic channel construct is point-to-point (depicted in Figure 2.4) whereby one process can communicate or exchange data with another process. In addition to this basic channel, other channel versions (like one-to-many, many-to-one, and many-to-many) can also be used.

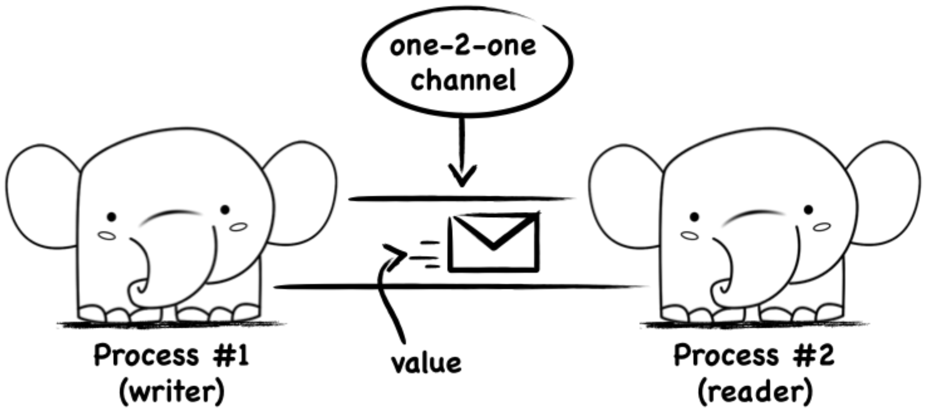

Figure 2.4: One-to-One channel communication in ProcessJ.

To understand how channels work, think of a channel as being a "piece of a garden hose" through (though we say 'on') which data can flow in one direction (channels are not bi-directional).

Other versions of channels can have shared ends, which in the garden-hose analogy means that ends have spreaders on. Rather than water flowing in the hose think of marbles. If the receiving end is shared the marble will eventually end up going down one of the end of the spreader and only one receive will get it (it does not replicate itself like in a traditional broadcast). This is exactly how channels work in ProcessJ. A process can communicate directly by exchanging messages or indirectly by reading from and writing to variables. Whichever process has the writing end can write a value to the channel and whichever process has the reading end can read the value that the sender sent. If a process is ready to send (write) a message, it will block until the receiving process is ready to accept the message. Similarly for a receiving process, if it is ready to accept (read) a message, it will block until the sending process is ready to send the message.

All channels are uni-directional. This means that values flow only from one end to the other and never in the opposite direction; therefore, values are always passed from writers to readers. An example of channel declaration is shown in Listing 2.14. This declares a channel variable called `c` that carries integer values.

```
chan<int> c;
```

Listing 2.14: Example of a channel declaration.

When declaring a channel, we must specify the type of data carried on that channel. We do this by specifying a type between a set of angular brackets (< >). The above declaration declares a channel called `c` as a local variable. Such channel has a reading end and a writing end. We can obtain these ends by using the two expressions in Listing 2.15.

```
c.read
c.write
```

Listing 2.15: Channel-end expressions.

Where `c.read` is a channel-end expression that evaluates to the reading end of the channel, and `c.write` is a channel-end expression that results in the writing end of the same channel. To communicate data along the channel `c`, all we have to do is write something to the writing end and read something from the reading end.

Channel Ends

The EBNF grammar for a channel-end declaration is as shown in Grammar 2.4.

$$\textit{channel\_end} \quad \rightarrow \quad [\ \textbf{shared}\ ]\ \textbf{chan} < \textit{type} > \textbf{.}\ (\ \textbf{read} \mid \textbf{write}\ )$$

Grammar 2.4: Grammar for a channel-end declaration.

A channel consists of two ends: a *reading end* and a *writing end*. We can only read from the reading end, and we can only write to the writing ends. Any channel variables ends can be obtained by suffixing either a `.read` or a `.write` (Listing 2.16).

```
public void f(chan<int>.read in) {
    ...
}

public void main(string args[]) {
    chan<int> c;
    par {
        ...
        f(c.read);
    }
}
```

Listing 2.16: Example of a channel's end declaration.

The declaration of the parameter in: `chan<int>.read` is a type, namely the reading channel end of a channel carrying integer values. Similarly, the writing channel end type is obtained by the `.write` postfix. To declare shared ends of a channel, we have three options: either both are shared, only the reading end is shared, or only the writing end is shared. Examples of all three types of declarations are shown in Listing 2.17.

```
shared chan<int> sharedReadWriteEnd;
shared read chan<int> sharedReadEnd;
shared write chan<int> sharedWriteEnd;
```

Listing 2.17: Example of a channel's reading end and writing end declaration.

Similarly, in the parameter list of a procedure declaration, we can specify how the ends of a channel are shared. However, it should be pointed out that the use of the `shared` modifier only applies to the end in question (Listing 2.18). The idea of sharing both ends of a channel makes no sense, and since channels cannot be passed parameters, the 'both ends shared' option does not apply for parameters.

```
public void f(shared chan<int>.read sharedReadEnd,
              shared chan<int>.write sharedWriteEnd) {
    ...
}
```

Listing 2.18: Example of a channel's ends in a parameter list.

Channel Write Statement

A channel write statement is simply a write to a channel such us the one depicted in Listing 2.19.

```
1: chan<int> c;
2: ...
3: c.write(42);
```

Listing 2.19: Example of a channel write statement.

Line 3 writes the value 42 to the channel `c`. As a matter of fact, it writes 42 to the *writing end* of the channel `c`. Therefore, the correct way to write line 3 (Listing 2.19) is as shown in Listing 2.20.

```
3: c.write.write(42);
```

Listing 2.20: Correct way to declare a channel write statement.

Where `c.write` is an expression that evaluates to the writing end of the channel `c` to which we can then write 42 by calling `write`. Although the compiler allows both ways as the double 'write' seems redundant, the code shown in Listing 2.21 is perfectly legal.

```
chan<int> c;

chan<int>.wrie cw = c.write;
cw.write(42);
```

Listing 2.21: Example of channel's writing end assignment.

2.2.4.5 Stop and Skip

The EBNF grammar for a stop and skip declaration is as shown in Grammar 2.5.

$$\begin{array}{lcl} \textit{stop} & \rightarrow & \textbf{stop} \\ \textit{skip} & \rightarrow & \textbf{skip} \end{array}$$

Grammar 2.5: Grammar for a stop and skip declaration.

`stop` and `skip` are two new keywords that most programmers may not be familiar with. In ProcessJ, `skip` is a no-op operation (it does not do anything), and could mostly be replaced by a semicolon (;) or an empty block ({ }). Contrary to `skip`, `stop` does exactly that (it stops), but does not terminate. `stop` is equivalent to an infinite loop that never does anything (e.g., `for (;;);`).

2.2.4.6 Barrier

The EBNF grammar for a barrier declaration is as shown in Grammar 2.6.

$$\textit{barrier} \rightarrow \textbf{barrier}$$

Grammar 2.6: Grammar for a barrier declaration.

A barrier is a *multi-way synchronization point* between a number of processes. It can be passed to procedures like any other primitive value. Additionally, a barrier can be enrolled on, synchronized on, and temporarily resigned from (Listing 2.22). However, barriers cannot be sent over channels, and they cannot be declared mobile.

```
barrier b;
...
enroll(b) {
    ...
};
...
b.sync();
...
b.resign() {
    ...
};
```

Listing 2.22: Example of the `barrier` construct.

When a process synchronizes on a barrier, it blocks until all other processes enrolled on the barrier have synchronized as well. When the barrier has completed, that is, once all processes have finally synchronized on it, all blocked processes are rescheduled for execution. Therefore, a barrier can be used to synchronized multiple stages of computation between a set of parallel processes; for example, we can guarantee that the computation of (say) variable b cannot proceed until another process has computed its value of a.

To synchronized on a barrier, we use the `sync` keyword (Listing 2.22). It is important to remember that no process can progress beyond a barrier synchronization until each barrier has been enrolled on it. If one of the enrolled processes fails to call `sync`, it will prevent all other processes from progressing in their execution. In addition, a process enrolled on a barrier may terminate at any time without deadlocking the remaining enrolled processes. This is because a terminating process *automatically* resigns from any barrier it is enrolled on.

2.2.4.7 Alternation

The EBNF grammar for an alt(ernation) declaration is as shown in Grammar 2.7.

$$
\begin{array}{lcl}
\textit{alt\_statement} & \rightarrow & [\ \textbf{pri}\ ]\ \textbf{alt}\ \{ \\
 & & \quad ([\ \textbf{(}\ \textit{expression}\ \textbf{)}\ \textbf{\&\&}\ ]\ \textit{guard}\ \textbf{:}\ \textit{statement})+ \\
 & & \} \\
\textit{guard} & \rightarrow & \textit{left\_hand\_side}\ \textbf{=}\ \textit{channel\_read\_expression} \\
 & | & \textbf{skip} \\
 & | & \textit{timeout\_statement}
\end{array}
$$

Grammar 2.7: Grammar for an alt(ernation) declaration.

To facilitate non-determinism in concurrent applications, ProcessJ supports alternations or `alt` for short. An alternation consists of a number of branches referred to as *cases*. Each case is accompanied by a **guard** (Listing 2.23), which must be ready in order to be considered as a possible alternative. The alternation executes exactly only one of these guards and the process which follows it. If no guard is ready, the alt statement is suspended until one (or more) guards become ready. However, if more than one guard is ready, only one is chosen. Naturally, this model of computation is explicitly *non-deterministic* as processes can be blocked waiting to send (if they are guarded by the input from a channel) or receive on any number of channels.

```
alt {
    x = c1.read(): { ... }
    y = c2.read(): { ... }
    t.timeout(500): { ... }
}
```

Listing 2.23: Example of the `alt` construct.

ProcessJ provides three types of guards: a **skip** guard, a **timeout** guard, and a **channel read** guard. A process can therefore choose and have access to several different channels or other guard types like timers. A **skip** guard is always ready, thus it can serve as a default option in an alt statement. A **timeout** guard only commits to synchronization when the timer has expired, that is, is ready if the amount of time given in the timeout has elapsed since the alt was evaluated the first time. Note that a time on a timer continually increments and cannot change while waiting for a timeout. A **channel read** guard, on the other hand, is ready if and only if there is a committed sending process at the other end of the channel and this process has not yet input its data.

The optional `pri` before the `alt` keyword adds a notion of priority. If we wish to prioritize a selection process, we can use a *prioritized alt*. If more than one guard is ready, for example, the first one listed is chosen and executes followed by the process it was guarding. In Listing 2.24, when we introduced that `skip` guard, the `skip` can be chosen (at random) even if there is input ready on the channel. If we wish to favor reading input from the channel over the `skip`, then we use a `pri alt`. The guard that appears first has highest priority, the second one has second highest priority and so on until the one that appears last, which has lowest priority. However, the statement following the `skip` guard will execute by default if channel `c` is not ready to read.

```
pri alt {
    x = c.read(): { ... }
    skip: { ... }
}
```

Listing 2.24: Example of the `pri alt` construct.

2.2.4.8 Par

The EBNF grammar for a block declaration is as shown in Grammar 2.8.

$$\begin{array}{lll} \textit{block} & \rightarrow & \{\ (\textit{block\_statements})*\ \} \\ \textit{par\_block} & \rightarrow & \textbf{par}\ [\ \textbf{enroll}\ \textbf{(}\ (\textit{expression})*\ ]\ \textbf{)}\ \textit{block} \end{array}$$

Grammar 2.8: Grammar for a block declaration.

In ProcessJ, statements surrounded by a set of '{ }' are, by default, executed in order from top to bottom as their Java counterparts. A parallel block, on the other hand, is a normal block with the `par` keyword before the opening '{' character. Processes inside a par-block (like the one in Listing 2.25) are safe to be schedule in any order (e.g., on a single-core processor) or in parallel (e.g., on a multi-core processor), and can only influence each other by communicating along dedicated point-to-point channels. As a consequence of this, a process cannot interfere with another process's state, thus no data race hazard are possible. Naturally, if a process needs to interact, then it must explicitly communicate.

```
...
par {
    foo();
    bar();
}
...
```

Listing 2.25: Example of the `par` construct.

A par-block in ProcessJ is similar to the one in occam-π. That is to say, it dynamically creates and combines a number of processes to be executed in parallel. Furthermore, the order in which these processes run does not matter, and as long as these processes continue to run in a par-block, the process in which the par-block is executed will be blocked from running.

A par-block can also enroll its parallel processes on zero or more barriers. In Listing 2.26, the par-block makes each statement a process and enrolls each of them on the barrier `b` and `c`. The barriers on which all three processes are enrolled are passed as parameters. If we did not do that and one of the processes, say `foo`, did not synchronized on the barrier it is enrolled on, then none of the other processes enrolled on this barrier can progress beyond the synchronization point. Of course, this could result in deadlock. However, if `foo` terminates, it will automatically resign from the barrier to allow other processes to continue.

```
barrier b, c;
par enroll b, c {
    foo(b, c);
    bar(b, c);
    baz(b, c);
}
...
```

Listing 2.26: Example of the `par enroll` construct.

2.2.4.9 Timer

The EBNF grammar for a timer and timeout declaration is as shown in Grammar 2.9.

$$
\begin{array}{lcl}
\textit{timer} & \rightarrow & \textbf{timer} \\
\textit{timeout\_statement} & \rightarrow & \text{ID}\ \textbf{.}\ \textbf{timeout}\ \textbf{(}\ \textit{expression}\ \textbf{)}
\end{array}
$$

Grammar 2.9: Grammar for a timer and timeout declaration.

Timers are used to post an event after a predetermined amount of time and can be read much like a channel (Listing 2.27). While regular channel reads are synchronous, that is, the sender and the reader must both be ready to communicate (otherwise neither can progress and the process is suspended), a timer read is always ready and can never cause a process to be suspended.

```
timer t;

long time;

time = t.read():
```

Listing 2.27: Example of the `timer` construct.

The second operation one can perform on a timer is a timeout as shown in Listing 2.28. Invoking a timeout on a timer prevents the process from progressing until the specified amount of time has passed. Timeout statements can also be used as guards in an alt statement. Like barriers, timers cannot be communicated on channels, and they cannot be declared mobile.

```
timer t;

t.timeout(1000):
```

Listing 2.28: Example of the `timeout` construct.

2.2.4.10 Record

The EBNF grammar for a record declaration is as shown in Grammar 2.10.

$$
\begin{array}{lcl}
\mathit{record\_type} & \rightarrow & \mathit{modifier}{*}\ \textbf{record}\ \mathrm{ID}\ \{ \\
 & & (\mathit{type}\ \mathit{variable\_id}\ (\textbf{,}\ \mathit{type}\ \mathit{variable\_id}\ ){*}\ \textbf{;}){+} \\
 & & \} \\
\mathit{variable\_id} & \rightarrow & \mathrm{ID} \\
 & & \mathit{variable\_id}\ \textbf{[ ]}
\end{array}
$$

Grammar 2.10: Grammar for a record declaration.

A record is similar to a `struct` in C as it consist of a number of fields with a specified type (Listing 2.29), but without the semicolon at the end. However, in ProcessJ, a record can *extend* any number of existing records.

```
public record Client {
    string firtstName;
    string lastName;
    string address;
    string city;
    int zip;
}
```

Listing 2.29: Example of a `record` construct.

Records are dynamically allocated using the **new** keyword followed by a literal record (Listing 2.30), and their members can be accessed using the *dot* syntax.

```
Client c = new Client {
    firstName = "SomeName",
    lastName  = "SomeLastName",
    address   = "SomeAddress",
    city      = "SomeCity",
    zip       = 12345
}
```

Listing 2.30: Example of how to create a record.

Additionally, in ProcessJ, not only can we derive a record from a base record, we can also derive a record from the derived record. This form of inheritance is known as *multilevel inheritance* (Listing 2.31). Furthermore, if any of the extended records have similar field names, the compiler will produce an error.

```
record A {
    ...
}

record B extends A {
    ...
}

record C extends B {
    ...
}
```

Listing 2.31: Example of a record inheritance in ProcessJ.

2.2.4.11 Protocol

The EBNF grammar for a protocol declaration is as shown in Grammar 2.11.

$$
\begin{array}{lcl}
\textit{protocol\_type} & \rightarrow & \textit{modifier}*\ \textbf{protocol}\ \text{ID}\ [\textbf{extends}\ \text{ID}\ (\textbf{,}\ \text{ID})*] \\
 & & (\\
 & & \quad \{ \quad (\text{ID}\ \textbf{:}\ \{\ (\textit{type}\ \text{ID}\ \textbf{;})*\ \})* \quad \} \\
 & & |\ \textbf{;} \\
 & &)
\end{array}
$$

Grammar 2.11: Grammar for a protocol declaration.

Protocols are similar to `unions` in C, except they are used to describe a structure for an individual message transmitted (communicated) on a channel. A protocol in ProcessJ allows various kind of information, containing possibly a mixture of data types, different data types, or different amounts of the same data types based on a program's runtime state, to be declared for individual channels. It is a type that contains one or more elements indexed by a *tag-name*, where a tag-name consists of a list of variables preceded by their data types, separated by semicolons, and enclosed between braces (Listing 2.32).

```
public protocol P {
  request : { int number; double amount; }
  reply : { boolean status; }
}
```

Listing 2.32: Example of a `protocol` construct.

Along with specifying a number of possible tags for communication on a single channel, a protocol can also *extend* any number of existing protocols. This enables programmers to implement additional functionality that other protocol types can benefit from. In Listing 2.33, for example, protocol `P` adopts a number of tag-named variables from protocols `Q`, `R`, and `S`. Note that while multiple inheritance (as well as multilevel inheritance) is allowed in ProcessJ, the compiler will also produce an error if any of the extended protocols have similar tags.

```
1: public protocol P extends Q, R, S {
2:   request : { int number; double amount; }
3:   reply : { boolean status; }
4: }
```

Listing 2.33: Example of a protocol inheritance.

While the definition in line 1 in Listing 2.33 only describes what a `P` protocol will look like, it does not itself describe a specific protocol containing values for a `request` or a `reply` (or for any of the inherited tag-name variables). To do that, that is, to dynamically create a protocol of type `P`, we must use the **new** keyword followed by a protocol literal (Listing 2.34).

```
5: P p = new P { request: number = 7, amount = 4.5 };
```

Listing 2.34: Example of how to create a protocol.

We can access the value (or values) associated with a protocol's tag using the *dot* syntax. We write the name of the variable whose value we want to retrieve immediately after the protocol's name, separated by a period (.), without any spaces. It should be pointed out that such value can only be retrieved using a **switch-case** statement. We pass the protocol type as an expression in the switch statement and only use the protocol's tag excluding its type name as a label (Listing 2.35).

```
 6: ...
 7: switch (p) {
 8: case request:
 9:     println("number: " + p.number + ", amount: " + p.amount);
10:     break;
11: case reply:
12:     println("status: " + p.status);
13:     break;
14: }
15: ...
```

Listing 2.35: Example of how to access a protocol's tag.

2.2.4.12 Constant

The EBNF grammar for a constant declaration is as shown in Grammar 2.12.

$$\textit{constant\_declaration} \quad \rightarrow \quad \textit{modifier}{*}\ \textbf{type}\ \text{ID}\ \textbf{;}$$

Grammar 2.12: Grammar for a constant declaration.

A constant declaration at the top-level is similar to a local variable declaration with the *const* modifier prefixed. For example, Listing 2.36 declares a public constant called `PI`.

```
public const double PI = 3.1415;
```

Listing 2.36: Example of how to declare a constant variable.

Top-level constants can only be declared of primitive types that are not barrier or timer types. Note that the use of the modifier `const` for local variables or parameters simply means they cannot be assigned, and they can be of any type

2.2.4.13 External

The EBNF grammar for an external declaration is as shown in Grammar 2.13.

$$\begin{array}{lcl} \textit{extern\_type} & \rightarrow & \text{ID} \\ & | & \textit{extern\_type}\ \textbf{.}\ \text{ID} \end{array}$$

Grammar 2.13: Grammar for an external declaration.

An external declaration at the top-level is similar to a `typedef` in C, except it creates an alias for an external Java type without having to qualify the access with the type name (Listing 2.37).

```
extern java.util.Hashtable javaHashtable;
```

Listing 2.37: Example of an external type definition.

Consider the code snippet in Listing 2.38.

```
public void foo(javaHashtable myHT) {
    ...
    int a = myHT.get(...);
}
```

Listing 2.38: Example of an external type declaration.

Using the external type `javaHashtable` in Listing 2.38, we can have access to a collection of key/value pairs, namely, a Java *Hashtable* which maps keys to values. Note that we can assign values to this external type and pass it as a parameter between various ProcessJ processes.

Chapter 3

Related Work

In this chapter I consider some of the existing approaches to concurrency and parallel programming. In particular, I consider the most widely used models of concurrency. I describe the difference between them and what the advantages and disadvantages of each are.

3.1 occam-π

occam-π [23, 34, 42, 106, 108] is a programming language based on occam [69, 109] which was a language designed specifically for parallel computing and for the Transputer microprocessor chip in the 1980s [31, 61, 68] – a chip designed by Inmos International, a British semiconductor company founded in July 1978, for efficient on-chip (time-slice) concurrency and for channel communication with other transputer chips. occam-π combines the concepts of CSP [63] with the π-calculus [70, 71, 89] to support concurrency and facilitate the mobility[1] and reconfigurability[1] of network of processes. In addition, it has integrated semantics for concurrency, such as processes and channels, and a number of features [24] that enable complex systems to be built while preventing problems like race hazards, deadlocks, and livelocks.

After the death of the transputer, occam-π became a language that existed only for narrow research purposes. Following the work that began at the university of Kent, numerous papers have been written and published, and several CSP-inspired libraries and languages have been developed [105]; this includes CTJ [59, 60], JCSP [11, 103, 107], CCSP [72], C++CSP [7, 37],

[1]Process mobility is supported by the dynamic, asynchronous communication capability of the π-calculus.

CHP [10, 36], PyCSP [102], Guppy [32], Go [26], and many others including, or shall we say culminating on, ProcessJ. Although occam-π has been used extensively in the past, but much less so today, it is all but a dead language. Unfortunately, it did not gain much popularity in the developer community, perhaps, because of its foreign/old-fashioned syntax (e.g., keywords are all uppercase letters), unfamiliar and forced indentation-based layout as part of its syntax (which was not popular until the popularity of Python began to rise), limited I/O manipulation, and portability issues (it only runs on 32-bit unix architecture, making it a rather restricted programming language). Additionally, it lacks APIs for certain common utilities [80]. For example, collection classes that are often used or, in some cases, are only partially implemented in other programming languages, such as a list, set, queue, etc., are not supported in occam-π.

Along with having secure but expensive communication when sending large data (on shared-memory systems), there is no aliasing in occam-π [33, 105]. The creators sacrificed the 'big-picture' in favor of safety, program correctness, and reducing the amount of mistakes that programmers make. While having restrict-like or no aliasing semantics in a programming language prevents programmers from making mistakes, it also prevents full use of the language. In occam-π, for example, programmers cannot build or implement several useful data structures to work with large amounts of data, and therefore the entire data must always be copied – a sender and a receiver hold separate copies of the data. Indeed, occam-π allows programmers to build massively structured concurrent programs that can be understood and implemented without difficulty, but could we (really) do it without aliasing in this language? I believe that is not the case. Most programming languages allow some kind of aliasing (e.g., two pointers set to point to the same variable in C/C++), or at least go to great lengths to make it appear as if they were not (e.g., reference variables in Java).

While it is widely accepted that aliasing is a dangerous feature to have in a programming language, a language without it is simply not well-suited for real-world problem solving. To optimize productivity, for example, we need different ways of organizing information on our computers. Since data can have a significant impact on the performance and execution time of a program, it is common for programmers to define data types to better represent and organize information so that it can be efficiently used and accessed; this includes the implementation of an Abstract Data Type (ADT) for elementary data structures such as lists, trees, stacks, and queues. Without aliasing, this is almost impossible to accomplish.

In spite of everything, occam-π has been artificially kept alive in some institutions, including UNLV, as it continues to serve as a learning tool for concurrent programming in computer science graduate level courses. A snippet of occam-π code can be seen in Listing 3.1.

```
PROC integrate(CHAN INT in?, out!)
  INT total:
  SEQ
    total := 0
    WHILE TRUE
      INT x:
      SEQ
        in ? x
        total := total + x
        out ! total
:
```

Listing 3.1: Example of occam-π code.

3.2 Java Threads

To support concurrent programming, the Java Virtual Machine (JVM) allows threads to run simultaneously within a program in a way where each thread can handle a different task at the same time. A thread in Java is a lightweight process that shares the same memory and cooperatively shares the resources of the process in which it is created [92]. Java uses the monitor mechanism [62] to ensure that threads are not executing blocks of code marked *synchronized* at the same time; this includes methods with the synchronized keyword and blocks followings the synchronized keyword. Furthermore, condition (synchronized) variables are used to properly manage thread-coordination and execution.

Every Java object and class is associated with a monitor and has one built-in lock by default. This lock is used to determine which thread controls the state of an object inside a monitor. A monitor is therefore a mechanism, influenced by the *critical region* concept [55], used to encapsulate data that cannot be accessed or referenced from outside of the monitor. This means that a monitor protects the data of an object from unstructured access, and it further ensures that the interaction between threads takes place in legitimate ways when data is accessed using the object's methods. Ignoring, for now, some serious traps related to sequential programming with objects, a monitor, supposedly, guarantees the following: that 1) a thread can have access to shared data after acquiring a lock of an object, and that 2) threads can communicate through shared data without interfering with each other.

Monitors, however, break the object oriented model for which Java was built. Since a monitor is not a class, there is no actual monitor object – at least not explicitly. A monitor is instead an instance of any class that has synchronized code in it[2]. Therefore, we cannot create or work directly with a monitor. The JVM provides *monitorenter* and *monitorexit* instructions which execute actions to lock and to unlock a monitor on an object (see Listing 3.2 and Appendix C for complete code). Further still, the Java **synchronized** statement creates these instructions so that multiple threads can coordinate access to an object. Because of this low-level synchronization mechanism, threads are controlled by calling functions inside a monitor and depend on notifications from other threads. Programmers are therefore likely to introduce bugs in their programs if they forget to use the synchronized modifier, wrongly evaluate guard actions, or have access to the data of some object via hidden references even when this data is kept in a private field.

```
public void f();
    Code:
       0: aload_0
       1: getfield      #3 // Field lock:Ljava/lang/Object;
       4: dup
       5: astore_1
       6: monitorenter
       7: aload_1
       8: monitorexit
       9: goto          17
      12: ...
```

Listing 3.2: Example of generated *monitorenter* and *monitorexit* bytecode instructions.

The latter is an alarming issue that, when writing single-threaded programs, many programmers (including myself) often overlook. As described by Welch [104], in Java, a method call made on

[2]Recall that to synchronize code, or a section of it, we must use the keyword *synchronized*.

an instance of a class can lead to unintended consequences. Since objects are *passive*, a method invocation on an object is not executed by the object itself; instead, it is executed by the caller thread. Furthermore, methods can be invoked from inside the body of other methods. If more than one thread of control contains a reference to the same object, they can all invoke methods at the same time, and thus putting the object in an inconsistent state. Since nothing stops passing the reference of an object as a parameter to another object (depicted in Figure 3.1), objects are at the mercy of any other object that has a reference to it. An example of threads executing in and out of objects can be seen in Appendix E.

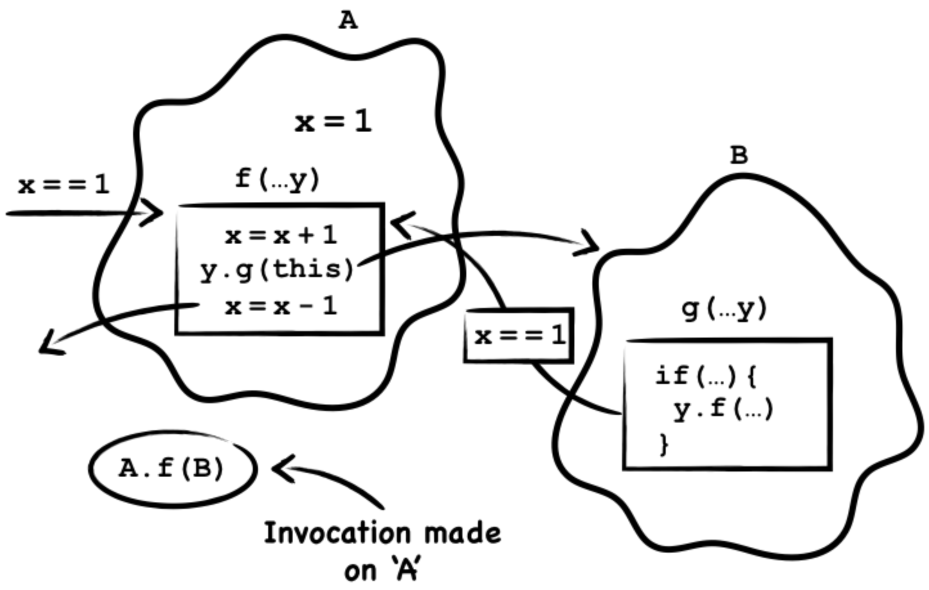

Figure 3.1: Spaghetti trails of threads of execution.

While synchronization is part of the Java language, multi-threaded and well-synchronized applications are often difficult to write due to the low-level abstraction required by monitors. The complexity and the risks associated with multi-threaded Java programming are describe in [52] as follows:

- Safety Hazards

  Proper synchronization is required to prevent race conditions. Resource conflicts may occur if different threads have unorganized access to the same object's synchronized block. When this happens, threads can simultaneously corrupt the object at the same time and produce different computational results.

- Liveness Hazards

  Java does not provide the means of detecting liveness failure such as livelock, deadlock, and starvation situations. Therefore, programs that require locks on multiple objects must use conventional techniques, such as locks, mutexes and semaphores, to avoid problems of this nature.

- Performance Hazards

  The system can fail because of excessive memory consumption when creating too many threads on the JVM. In addition, if more threads than CPUs exists, the JVM may regularly switch from running one thread to running another one. Naturally, the scheduler will have to suspend the running thread so that another thread can run, which means more time may be spent changing the context of threads than executing the program.

Even though Java provides support for parallel computing, it is almost never recommended to use an object (using the *synchronized* keyword) as a locking mechanism [74]. It is a common misunderstanding to believe that simply owning the lock of an object prevents other threads from accessing that object – that is not the case! In order to properly plan parallel operations, special care should be taken to avoid data race hazards, deadlocks, livelock, and starvation. Unfortunately, managing these problems requires some extra effort – but the resulting code is far less reliable and harder to maintain.

3.3 Communicating Sequential Processes for Java (JCSP)

Communicating Sequential Processes for Java (JCSP) [11, 103, 107] is a library of CSP constructs which provides the necessary tools for writing concurrent systems in Java via message passing. This library is based on the union of Hoare's CSP model of concurrency [63] and Milner's π-calculus [70, 71, 89], and follows many of the occam-π [23, 34, 106] principles to support mobility of channels. JCSP accomplishes concurrency by encapsulating the implementation details (using primitives, extension, and wrappers) to support CSP elements such as channels, processes and various other operators. The advantages of using this library are that it hides the complexity of monitor synchronization, prevents race hazards, and gives Java programmers a more convenient parallel computing model.

While JCSP enables programmers to write process-oriented programs in Java, it maps processes directly to the JVM threading mechanism. For example, in Listing 3.3, when a JCSP process (a class that extends the JCSP class `CSProcess`) is created and executed in a par-block (which in JCSP is the `Parallel` class – another class that is part of the JCSP library), it is executed in a Java thread. This indicates that JCSP processes have similar process memory and runtime overhead as threads [82]. The reason is that the JCSP library is built on top of the standard Java Thread mechanism, thus it carries the overhead involved with Java threads.

In general, the JVM does not support hundreds of thousands (or millions) of threads. Since hardware configurations limit the number of threads that the JVM can maintain, the number of processes that JCSP can create is therefore determined by the amount of memory available on the system. Further still, other experiments [87] with JCSP determined that the maximum number of JCSP threads in one JVM is in the tens of thousands. Naturally, with JCSP, the number of processes required for a large and complex parallel model such as the one described in [1, 93, 100] is simply impossible.

```
public class Example implements CSProcess {
    public void run() {
        One2OneChannel chan = new One2OneChannel();
        new Parallel (
            new CSProcess[] {
                new SendProcess(chan),
                new ReadProcess(chan)
            }
        ).run();
    }
}
```

Listing 3.3: Example of a JCSP process.

As mentioned, creating too many threads causes performance overhead. If the CPU needs to switch between threads, the running thread must be suspended and its register values must be saved in memory before the thread is loaded back to resume. Every time we deliberately change the status of a thread (e.g., by sleeping, waiting on an object, changing its priority, etc. – which JCSP *implicitly* does for proper thread synchronization), we will introduce a context switch. While we may notice a boost in performance at the beginning, at some threshold, the number of context switches will impact the overall performance of our program, making processing slower. As a result of this, JCSP becomes unusable for all practical purposes when working with a large network of processes in a single JVM. Mapping JCSP processes to threads is not a viable solution for process-oriented programming in Java.

3.4 Message Passing Interface (MPI)

The Message Passing Interface (MPI) [20, 53] is an industrial standard based on the consensus of the MPI Forum that specifies library routines for programming parallel systems. The MPI library's foundation is based on the different functions used to transmit data that requires cooperation between a sender and a receiver. MPI focuses primarily on communication routines to support concurrent programming. Through the use of well-defined subroutines, it enables users to control data transition between processes in order to achieve parallelism. It is currently the most popular message passing parallel computing tool used on largely parallel machines with distributed-memory architectures.

MPI has a collaborative execution model. Processes work together to solve a single problem but they operate separately from each other. Because of this, each process has its own memory and cannot exchange information in memory variables. Since no shared memory exists, local calculations must be performed and explicit communication must be used to send data back and forth between processes. Consequently, the user has to divide the data between processes – with most of the processes working on subtasks and a few others (often just one process referred to as 'the master') managing these tasks (Figure 3.2), think about the actual message passing infrastructure to effectively divide the work between processes, and use a number of MPI communication routines to transfer data where necessary. The most common basic routines are shown in Listing 3.4.

```
// Used to send a message to another process
MPI_Send(buf, sizeof(buf), MPI_CHAR, 1, 0, MPI_COMM_WORLD);

// Used to receive a message from another process
MPI_Recv(buf, sizeof(buf), MPI_CHAR, other_rank,0,
          MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Listing 3.4: MPI common basic routines.

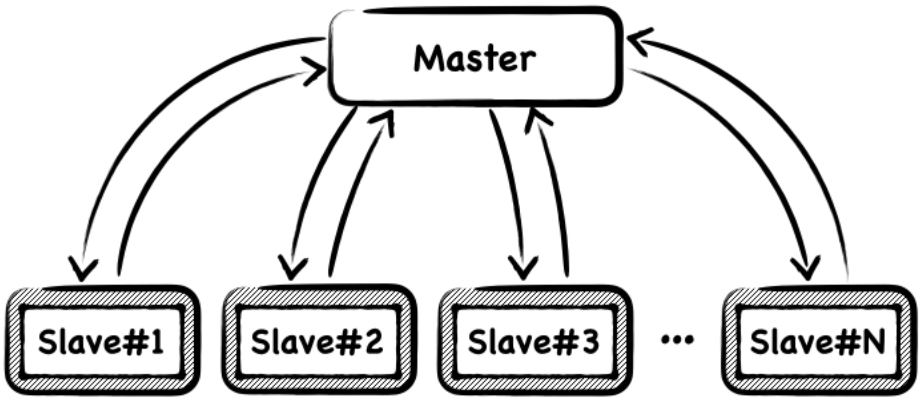

Figure 3.2: MPI's master and slaves processes.

```
main(int argc, char *argv[]) {
    ...
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(..., &rank);
    if (rank == 0) {
        master();
    } else {
        slave();
    }
    ...
    MPI_Finalize();
    ...
}
```

Listing 3.5: Example of an MPI program.

Sequential-to-parallel transformation using MPI requires a great deal of effort. One reason for this is that communication tends to be more expensive than local operations. This is because of the extra overhead of functions calls in point-to-point and collective communication, which eliminates some of the benefits of using the MPI library [54] (e.g., modularity, composability, and completeness). To solve this problem, we must *minimize* the number of messages exchanged and *maximize* the workload of each process to improve performance. The programmer is therefore responsible for correctly implementing parallel algorithms using MPI constructs. As a consequence, it is easy to write code that performs and scales poorly because data must be explicitly distributed with load balancing in mind. Of course, this is often difficult to achieve.

Safety issues are another reason that make MPI difficult to use. The MPI standard guarantees that a message is sent and receive without conflicts and corruptions, but it does not guarantee fairness when sending and receiving messages from any source. This, of course, does not reduce synchronization or other code errors. The programmer is therefore responsible for correctly identifying parallelism when implementing MPI routines. The lack of compiler optimization for MPI calls is another issue. Even if tools such as the ones mentioned in [40, 56] can be used to improve the communication and computation overlap in MPI applications with minimal (and almost no) user interference, they cannot be used in practice as they are still under development. As a final remark, there are still a number of open problems in MPI [101] (e.g., performance, scalability, fault tolerance, support for debugging, topology, etc.) that must be addressed and improved for the new developments in parallel systems.

3.5 Open Multi-Processing (OpenMP)

Open Multi-Processing (OpenMP) [39] is an API that consists of compiler directives and library routines for programming shared memory applications that extend C/C++ and FORTRAN. It enables programmers to explicitly instruct the compiler which loop iterations should be performed in parallel, how to combine the results of these iterations to create a single outcome, and what data should be treated as shared or private in each loop iteration.

Contrary to MPI's execution model, OpenMP's model is based on forks and joins [16]. An application is broken into a group of threads that run on a shared memory architecture. The application initially runs sequentially with only one thread (the master thread) until a parallel region of code in the program is reached. In this region, OpenMP creates (*fork*) a number of worker threads that run concurrently with the master thread. When the worker threads are done executing the statements inside the parallel region, OpenMP synchronizes (*join*) the data of these threads and terminates them, leaving only the master thread to continue until the next parallel region is reached. Figure 3.3 shows the **fork** and **join** sections created after using the `omp for` construct [25] (Listing 3.6). The `#pragma omp parallel for` divides the **for** loop into n number of tasks that are shared among threads.

```c
void simple(int n, float *a, float *b) {
    int i; // i is private by default
# pragma omp parallel for
    for (i = 1; i < n; i++)
        b[i] = (a[i] + a[i-1]) / 2.0;
}
```

Listing 3.6: Example of a simple `parallel loop` in OpenMP.

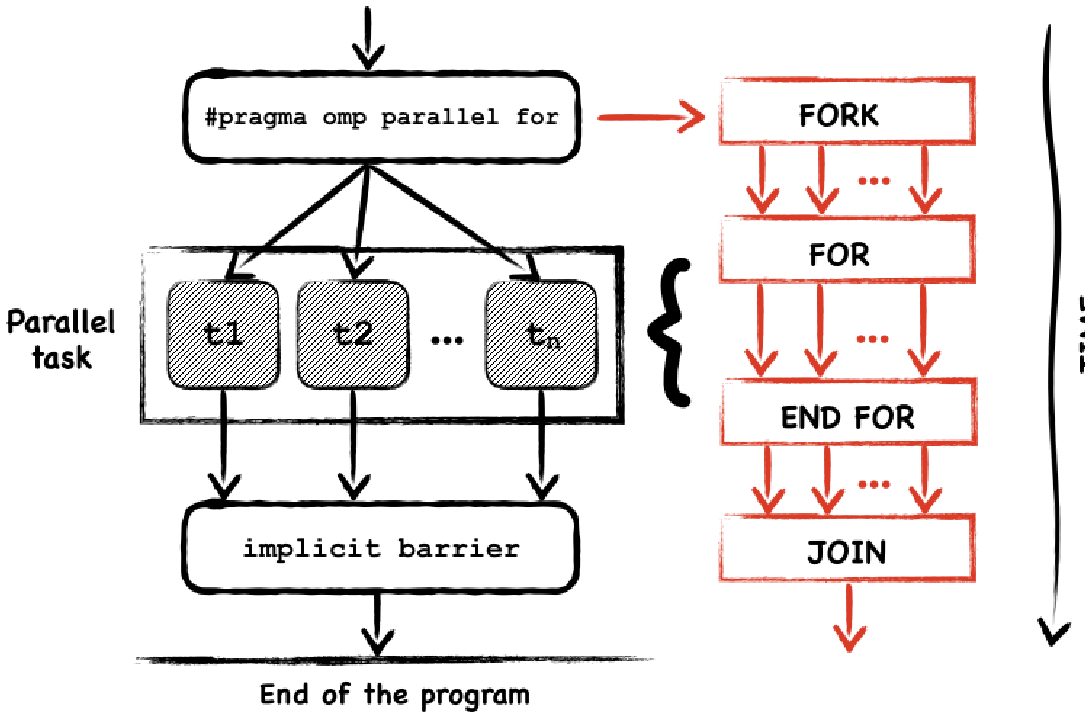

Figure 3.3: Example of how to create threads in OpenMP.

Although OpenMP is easy to use and requires little programming effort, [96, 97] describe the disadvantages of using it when developing distributed applications. Scalability is limited by the architecture of the memory. Since OpenMP runs on shared memory systems, there is a limited number of processors to be used. Consequently, sufficient work must be done in each parallel loop to avoid extra overhead. Like MPI, OpenMP can be difficult to use because it requires explicit synchronization. This means the programmer is responsible for the correct synchronization of threads which can make the implementation of a program as complicated as using MPI routines. When work is distributed among parallel threads, for example, a program can produce wrong results if the data on one thread depends on the data on another. If task (say) *a* relies on the completion of task *b*, we need to guarantee the completion of task *b* before invoking the execution of task *a*. If threads do not bring some order to the sequence in which they do things, the outcome will be disastrous.

It is also the responsibility of the programmer to ensure that data dependency is excluded. Problems arise when a variable that is used (read from) and updated (written to) is declared as a shared variable in a parallel region. The compiler is not smart enough to rule out data dependency when the `parallel` directive is placed around a section of code. Naturally, multiple threads can attempt to write to the same shared variable simultaneously. Since threads use the same memory, precautions must be taken to correctly manage multiple tasks involving access to shared resources. Lastly, OpenMP threads are not particularly notable for handling fine grain data parallelism as they require more communication and computing efforts. This typically results in load imbalance when certain threads waste (a great amount of) time between synchronization points; for example, time wasted for waiting to enter critical regions, or when they have unequal amount of workload.

3.6 Hybrid Approach (MPI/OpenMP)

An MPI process has a single thread of execution, however, the addition of threads to MPI programs (Listing 3.7) gives programmers the ability to execute multiple threads simultaneously within a single MPI process. Programmers can therefore take advantage of MPI's data placement policies and OpenMP's fine grain parallelism [96, 97] to reduce communication requirements, memory consumption, and improve load balance. In a multi-core processor machine, for example, we could have each processor run an MPI process, and, at the same time, have each MPI process execute a set of threads equal to the number of processor in this machine (Figure 3.4).

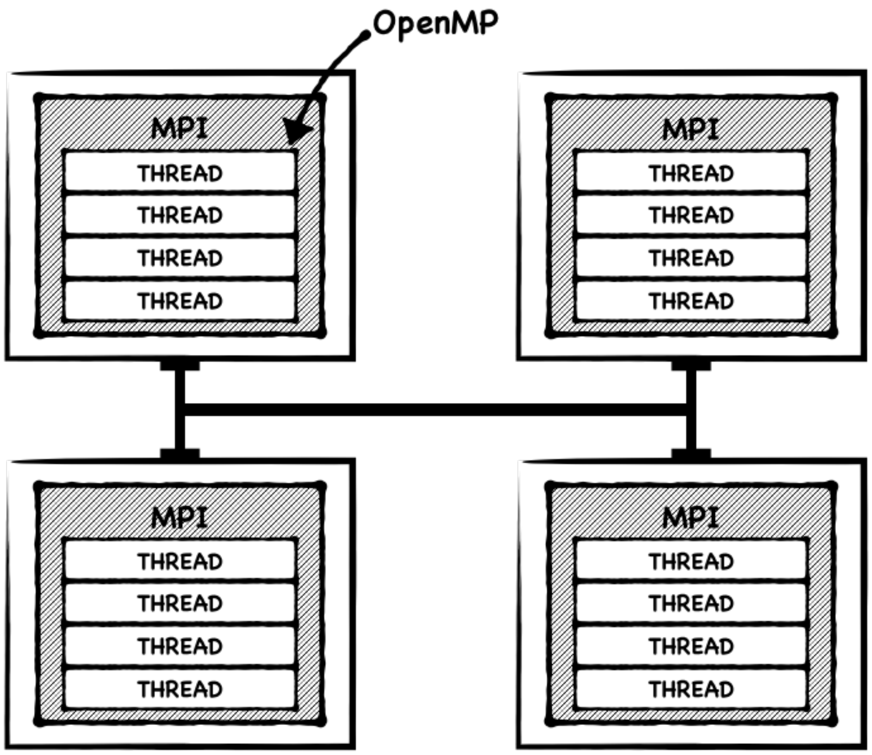

Figure 3.4: Hybrid approach example with OpenMP and MPI.

```
main(int argc, char *argv[]) {
    ...
    MPI_Init(&argc, &argv);
    ... // master thread
    # pragma omp parallel
    {
        ... // group of threads
    }
    ... // master thread
    MPI_Finalize();
    ...
}
```

Listing 3.7: Example of a hybrid program with OpenMP and MPI.

As described in [83], with this hybrid programming approach, we are able to have the following: better support for load balance when having a mix of static and dynamic scheduling (now that we have threads within a process, we can take advantage of OpenMP to make full use of other computational resources), lower overhead when using threads (the creation and placement of threads can reduced communication overhead in MPI applications), good usage of resources in shared memory systems (the amount of replicated data is reduced significantly because there are fewer larger blocks managed with OpenMP, thus there will be fewer MPI messages with larger message sizes), and no extra overhead among MPI processes (the number of processes is also significantly less, resulting in less communication among processes and increased performance).

Note that whether or not a mixed OpenMP/MPI programming approach performs better than pure MPI depends on the number of messages exchanged between processes, and the number of threads and the amount of work done by each thread. Often, the number of threads per MPI process required by any application is difficult to predict without initially creating some performance degradation. Such number can depend on the application itself, the input data, the hardware platform, the number of processes being used, the compiler and the MPI library implementation [22]. Naturally, to determine the correct (or optimal) number of threads, thorough benchmark test must be done. This indicates that the best combination of MPI tasks and threads should be determined by experimentation in order to achieve good performance.

Furthermore, threads can become a major source of overhead [22]. When there is communication among threads of different MPI processes, if a thread involved in an MPI communication becomes idle, then no progress will be made until the thread becomes active again. Similarly, if all threads are idle except for one while an MPI communication is in place. Additionally, if the program is not thread safe, multiple threads can engage in the same MPI communications at the same time. Since multiple threads have access to shared MPI communications, each MPI call must be protected to guarantee correct input and output data. This adds a lot of overhead due to the frequent overlap of threads and processes in collective communication and computation operations. Therefore, making MPI calls within OpenMP blocks is not always safe and limits performance.

3.7 Actor Model

The actor model [57, 58] was first introduced by Hewitt in 1973. A mathematical definition was subsequently presented for the behavior of an actor system by Agha [29]. Since then, this model has been incorporated into many languages (or as libraries in languages that do not have actors built-in), Erlang [13, 30] being one of the most popular. The actor model is a paradigm that defines some basic rules for how components in a system should behave and interact with each other via message passing. An actor is an entity (an active object[3] [76]) that has a *mailbox* to store messages and a *behavior* that changes depending on which message it receives (Figure 3.5). It can send other actors a number of messages, create a number of actors, and determine how to respond to the next message it receives. Such actions are conducted in any order and could be performed in parallel.

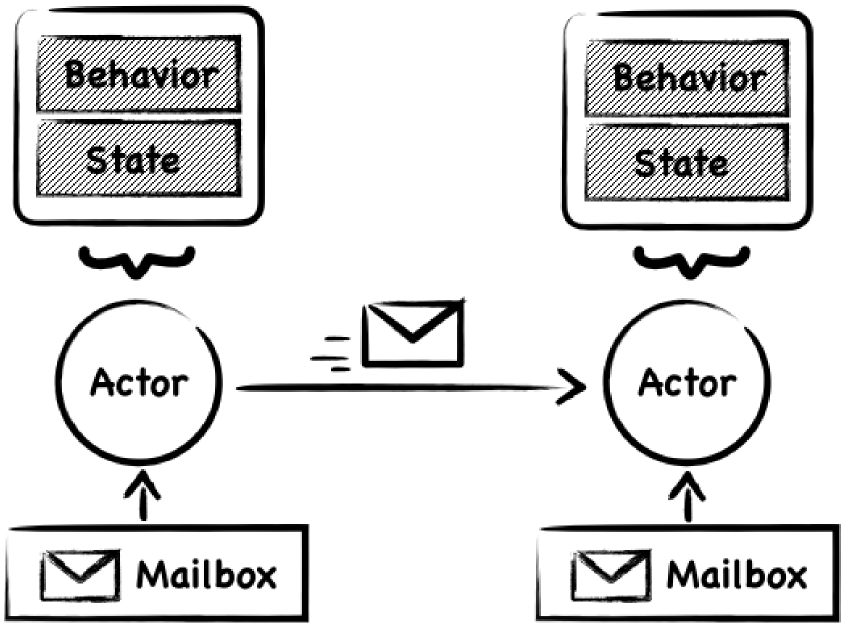

Figure 3.5: Actor model example.

An actor does not share its data and only communicates with other actors to which it is connected by sending asynchronous messages. This means we can write a program consisting of simple sequential processes and avoid a lot of problems resulting from sharing state. Naturally, since there is no shared state between actors, it is only possible to manipulate their internal states through messages. For example, if an actor wishes to obtain information about another actors internal state, it will have to send a message to request this information. In addition, an actor processes its messages one at time, in the order that it receives them, and uses pattern matching[4] to determine

[3]Active objects are a form of multitasking for computer systems. They manipulate their own execution thread instead of using the execution thread of the object that created them. Therefore, they can be used to develop parallel applications.

[4]Pattern matching is nothing more than the action taken in response to the message being processed.

what to do with each message.

While the actor model enables a system to be decomposed into independent and autonomous components that work asynchronously, it has some disadvantages as described in [28]. There is no notion of inheritance or general hierarchy, which means that an actor based implementation can be time consuming and confusing to program. For example, 'over-decomposing' applications into actors with common behavior (e.g., spreading out similar logic in different places) may be more unreadable than code that uses threads and blocking request. Another issue is the ability of actors to create other actors. Sometimes creating (an excessive number of) actors can have a dramatic impact on the responsiveness of a system. In order to avoid this, the system will need to know which actor is executing and where, thus a number of records containing this information is required. This may result in performance penalties, particularly, in highly distributed systems.

Another disadvantage of using the actor model is the asynchronous message passing. The ordering of messages received form multiple actors may be inconsistent in larger systems (e.g., fine/coarse grain) causing problems with certain programs and/or algorithms. For example, in a stack-like structure, push and pop operations may not necessarily be executed in the correct order because a message containing a push operation could be overtaken by a message containing a pop operation and vice versa.

Finally, nothing prevents actors from sharing their state. When using the akka [3] framework in Java, for example, actors can have shared, mutable (changeable), state. Consider the actor example in [2] with some state (Listing 3.8), a `HashMap` that supposedly contains information it needs in order to function. We can accidentally leak a reference to a mutable data (the mutable `HashMap` in this case) when sending a `GetState` message to the sender requesting the status. If two different actors, at any given point in time, (accidentally) obtain a reference to this map, they can both read and write to it at the same time on another thread. Naturally, when sharing mutable state between actors, there is no guarantee that an actor will not concurrently try to modify the state of another actor.

```java
public class BadActor extends AbstractLoggingActor {
  private Map<String, String> stateCache =
                        new HashMap<String, String>();
  public BadActor() {
    receive(
      ReceiveBuilder.match(GetState.class, request -> {
          sender().tell(new State(stateCache), self());
      }).build()
    );
  }

  public static final class GetState {
    public static final GetState Instance = new GetState();
    private GetState() { }
  }

  public static final class State {
    private final Map<String, String> state;
    public State(Map<String, String> state) {
        this.state = state;
    }
    public Map<String, String> getState() {
        return state;
    }
    @Override
    public boolean equals(Object o) { ... }
    @Override
    public int hashCode() { ... }
  }
}
```

Listing 3.8: Example of an actor model with shared mutable state.

Chapter 4

Command Line Interpreter (CLI)

A command line interpreter enables a rudimentary type of interaction between users and application programs by means of entering commands to a computer. This interaction relies on textual input and output in which characters are displayed in a terminal program that accepts various arguments as input on the command line. These arguments are often referred to as parameters, commands, or options (a series of instructions that a program executes) and are typically typed in by the user. In addition, each command line consists of one specific command word usually followed by optional arguments used by the command. The general form of a command in [84] looks like this:

$$\texttt{command}\ \texttt{arg}_0\ \texttt{arg}_1 \ldots \texttt{arg}_n$$

Where `command` is the name of a program and $\texttt{arg}_0 \ldots \texttt{arg}_n$ are individual values passed to the program. To run commands on a terminal, for example, the user types them in the command line, presses the ‘return’/‘enter’ key, and then waits for a response. After receiving the line typed by the user, the command line interpreter splits the argument into separate strings, processes the strings one by one, executes them as requested, and displays output (if any) before giving another prompt. Consequently, a command line interpreter has a straightforward functionality; it allows users to respond to visual prompts by typing commands on a terminal program and, in a similar way, allows them to receive feedback. Figure 4.1 shows the parts of a typical command line in a UNIX environment separated by white spaces.

Although the syntax of a command line can vary from one operating system to another, the

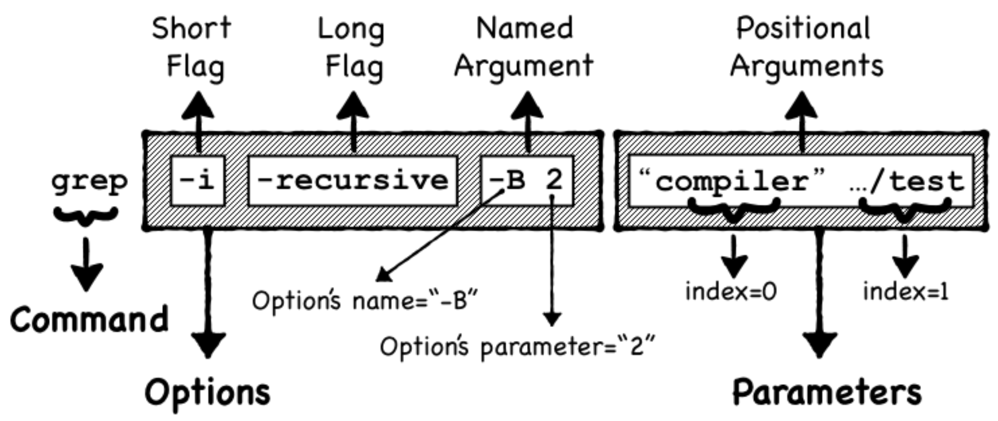

Figure 4.1: Parts of a command line.

command line illustrated in Figure 4.1 consist of three common types of components. These components describe the role of each argument on the above command line as follows:

- Command

  The name of an executable program that may be followed by a list of arguments. A command does not begin with a single or double dash, and it is not associated with an option; therefore, it does not need an option in order to be run.

- Options

  A word that mostly starts with '–' or '––' and can take arguments. When an option is defined as a 'named argument', the option behaves as a key-value argument pair where its key identifies its value. However, when used as a 'stand-alone flag', the option behaves as a Boolean value. That is, its value can be set to `true` after passing the option to a command.

- Positional arguments

  Any word that is not commonly prefix with '–' or '––', and its position in a list of arguments is determined on the command line. Note, a positional argument may follow the above convention of options if it appears after the '––' (also known as the bare double dash) which means the end of options.

- Special characters

  Although not present in Figure 4.1, these are symbols of particular significance beyond their literal meaning in the command line. Some well-known examples are the '<' symbol for input redirection, the '>' symbol for output redirection, and the ';' symbol for putting two or more commands on the same line.

A large number of programs have conformed to the UNIX standard convention for naming and processing commands, options, and parameters. However, many still do not follow this standard. Some programs do not require that options start with '–' or '––'; the Windows' command line interpreter, for example, requires that options start with '/' and be case sensitive. Other programs do not allow multi-options or group of options. Instead, these programs require that options be separated by white space characters and start with '–' or '––'. For ProcessJ, we decided to follow the structure of the UNIX command line in the way most compilers interpret commands but added some small variations. Section 4.1.2 has a summery of the syntax rules that ProcessJ obeys when specifying command line arguments on the command prompt, and for defining commands for the compiler.

4.1 Overview

The ProcessJ command line interpreter is a lightweight command line parsing library that makes writing command line tools for the ProcessJ compiler simple and easy while allowing quick customization when required. The main purpose of the command line interpreter is to improve the parsing of command line arguments, provide good-readable error messages, support a variety of use cases, provide auto-completion for invalid commands, and automatically generate a command usage text from the set of options and parameters defined for the ProcessJ compiler.

This section discuses the implementation and nature of ProcessJ's command line interpreter including: the reason for building the command line parsing library; what the command line interpreter can do; how to define commands, options, and parameters; and how to parse command line arguments.

4.1.1 Designing the Library

Command line parsing tools can be as simple or as complex as programmers want. While there are a lot of command line parsing libraries available in Java such as args4j [5], JOpt Simple [18], CLAJR [8], JArgs [17], JSAP [19], and several others, none of them fit well with what we needed.

For the design of the command line interpreter, we only had two choices. We could use one of the available libraries to build a command line tool that parses options into temporary variables, be it local or global, and provides adequate access to them. Or, design a command line interpreter that follows a specific set of command line syntax rules based on the existing implementation of the ProcessJ compiler. The main advantage that let us to pursue the design of the command line interpreter was the ability to reuse our existing compiler code. We wanted our command line interpreter to be simple, typed safe, and capable of parsing complex arguments. For this reason, we chose the second approach in order to achieve simplicity, type safety, and capability.

Without going into much background detail, the following observations are made about two experimental versions of ProcessJ. One of the earliest experimental version of ProcessJ used Apache Commons CLI [9] for parsing command line options passed to the compiler. However, a newer recent version of ProcessJ uses a simple for-loop to parser command line arguments passed to the *main*() method. Before considering the reasons for building the command line interpreter, I will briefly cover the parsing methods implemented in both versions in the two sections below.

Parsing Command Line Arguments using Apache Commons CLI

With the Apache API, [98] describes the command line arguments of ProcessJ as shown in Listing 4.1.

```
final Options options = new Options();
final OptionGroup targetLanguages = new OptionGroup();
targetLanguages.addOption(new Option("m", "mpi", false,
                    "compiler to c and use mpi"));
targetLanguages.addOption(new Option("j", "java", false,
                    "compile to java"));
options.addOptionGroup(targetLanguages);
```

Listing 4.1: Apache Commons CLI command line declaration.

Apache Commons CLI provides a mechanism (a set of class) that can parse command line arguments containing optional arguments, arguments with parameters, and short/long arguments. Although simple to use, once the command line arguments are parsed, a sequence of cascading **if-else** statements – which [9] calls ‘the interrogation stage’ – is required in order to determine if an option's distinct flag is present. As illustrated in Listing 4.2, the execution begins with the first **if-else** statement (the most-top one) and continues to several levels.

```java
...
final boolean hasJava = commandLine.hasOption("java");
final boolean hasMPI = commandLine.hasOption("mpi");
...
if (hasJava) {
 // do something
} else if (hasMPI) {
 // do something
} else if (....) { ... }
...
```

Listing 4.2: Apache Commons CLI's interrogation stage.

We can clearly see that we have to extract the value of each option to avoid having to call the *hasOption*(...) method regularly in the expression of an **if-else** statement. We can also see that our validation checks get cluttered with similar conditions appearing in multiple branches. In addition, depending on the application, we may need to move early-exit code as close to the top as possible so that these **if-else** statement cases can be ignored and never be executed by accident. Generally speaking, we have to unnecessarily include the same section of code in many places with little or no alteration.

Another problem when using this library to parse arguments is that options within one group are not mutually exclusive by default. This means that all options, be it required or not, must be specified on the command line. If one of the options is missing, the parser will throw an error. To avoid this, we could define two sets of options and parse the command line twice. The first set of options would contain options that come before the required group, and the second set would contain the remaining required options. While there are other temporary solutions to this problem, the workarounds typically contain very complicated code. Therefore, despite being the most widely used command line parsing library, Apache Commons CLI is not a perfect solution.

Parsing Command Line Arguments Using a For-loop

The implementation of a for-loop in [95] is far from being a better solution. The piece of code in Listing 4.3 shows how command line arguments are processed and parsed out of the argument array of the *main*() method.

```java
...
for (int i = 0; i < argv.length; i++) {
 ...
 try {
   if ( argv[i].equals("-")) { // single option
      s = new Scanner( System.in );
   } else if (argv[i].equals("-I")) { // optional flag
      if (argv[i+1].charAt(argv[i+1].length()-1) == '/')
          argv[i+1] = argv[i+1].substring(0, argv[i+1].length()-1);
      Settings.includeDir = argv[i+1];
      i++;
      continue;
   } else if (argv[i].equals("-t")) { // option takes arguments
      if (argv[i+1].equals("c") || argv[i+1].equals("JVM") ||
          argv[i+1].equals("js")) {
        Settings.targetLanguage = argv[i+1];
        i++;
        continue;
      } else {
        System.out.println("Unknown target option for -t");
        System.exit(1);
      }
   } else if (argv[i].equals("-help")) { //optional flag
      usage();
      System.exit(1);
      continue;
   } else if (argv[i].equals("-sts")) { // optional flag
      sts = true;
      continue;
   } else {
      Error.setFileName(argv[i]);
      Error.setPackageName(argv[i]);
      s = new Scanner( new java.io.FileReader(argv[i]) );
   }
   p = new parser(s);
 } catch (java.io.FileNotFoundException e) {
    ...
 }
 ...
}
```

Listing 4.3: Parsing command line arguments using a `for` loop.

Even though the above piece of code is reasonably short and reasonably easy to implement, it raises a number of questions.

- Is `"-I"` in line 8 an option or an argument?
- What is the option's name? What does `"-I"` mean?
- What is the type of `"-I"`? Is it an integer, double, string, etc.?
- Is `"-I"` a required option?
- If `"-I"` is an option, does it required an argument?
  - If `"-I"` does not required an argument, then
    - Does it have a default value?
    - Is there an environment variable that provides this value?
- Can `"-I"` appear only once? Or, can it appear multiple times?

Once the command line arguments are made available in the code, the for-loop handles all of the arguments, one by one, as follows: it parses each argument, checks if the syntax used is valid and supported, and then retrieves the value required (if any) for each specified option before running the code according to the input provided by the user. Although this for-loop is great for handling simple arguments, but as is often the case, some arguments are options that have their own arguments. In this situation, the options and their arguments need to be processed together. Of course, this requires using another loop to parse an option with its own argument. Unfortunately, a for-loop makes it impossible to customize a program without having to rewrite (almost the entire) code. It should therefore be clear that using a for-loop to parse command line arguments is not the best approach.

4.1.1.1 Simplicity

Unlike many other libraries of this kind, ProcessJ's command line interpreter uses Java's runtime *reflection* and *annotation* capabilities to make the parsing and manipulation of command line arguments trivial with minimal syntax and no external dependencies. In the simplest form, for example, to instruct a program on how to process user input, we annotate a class and its fields with descriptions of our options and parameters. The compiler then processes these annotations at runtime and updates the fields of our target class via reflection. This makes the instance of our class very easy to configure and control with a variety of applications. In addition, the command line interpreter makes the binding of different types of arguments to options and parameters effortless through default type converters, simplifies the support for most built-in data types available in Java, encapsulates default settings which can be modified and used in the program, and does most of the basic correctness checking for the compiler.

4.1.1.2 Type Safety

There is no need to know or remember the type of argument we want to parse when using Java's built-in data types such as byte (or Byte), short (or Short), char (or Character), int (or Integer), long (or Long), float (or Float), double (or Double), String, and enum data type, or when using parameterized fields[1] with primitive data types. ProcessJ's command line interpreter will infer the above-mentioned types at compile time. However, we must explicitly specify some way of handling complex types (such as files, list, arrays, and maps) and user-defined types (those that programmers defined themselves) other than the above built-in ones. This type safety results in writing clean code while providing a strong type system that not only prevents the creation of abstract types but also enforces the use of basic Java constructs and data types. Table 4.2 in Section 4.1.3.4 shows in detailed the list of data types supported by the command line interpreter.

4.1.1.3 Capability

In ProcessJ, we can instruct the parser on where to find the input data and how to match this data against a command, an option, or a parameter. This is possible due to the following seven features:

- Multiple option names

  Options can be given various names, or aliases, in the form of a list. Naturally, one of these names can be used to specify the option as it should be entered on the command line.

[1]A Java generic field declaration has a type parameter delimited by angular brackets ($<$ and $>$).

- Multiple option values

  Options may be followed by one or several values (in any order) if they can occur multiple times. For example, the option `-f`, in most programs, is often followed by one or more input files that are typically accessed in the order in which they are given.

- Type converters

  A type converter can be used to translate a string literal value into some typed value. That is, a value of the appropriate data type.

- Group of options

  Options can be grouped into a single command, after which each option can be processed independently and then executed in the program.

- Range specification checking

  A range can be used to specify an exact number of values, or a minimum and maximum number of values required by an option or parameter.

- Custom error handling

  This includes how to handle errors (exceptions) that occur while processing arguments from the command line, and how to provide additional information about the cause of these errors.

- Reading commands from a file

  Options and arguments can be supplied to the compiler over a text file. This reduces the limitation on the number of arguments that can be passed to the compiler on the command line.

- Command auto-completion

  This feature displays a full command name or a list of command suggestions after we press the 'return'/'enter' key and the word entered was not what we intended by the abbreviation. The parser makes use of the Levenshtein edit distance [65] to quantify the difference between two strings. Therefore, it will recognize a command once we have entered enough characters to make the command unique.

All this improves consistency when dealing with different forms of command line arguments. Additionally, the command line interpreter generates a self-describing message (a man-page-like documentation) and a command usage text based upon defined commands, options, and parameters. For example, the ProcessJ compiler provides an option `-help` that shows a summary of all the options and parameters that the compiler understands. This is useful for displaying a typical usage guide with four sections: header, description, option list, and footer. The compiler also provides an option `-error-code` that takes an error code number[2] and displays information in response to compile time and runtime errors. This helps identify the cause of the problem in the source code and helps understand the severity levels of error and warning messages produced by the compiler, such as ERROR, WARN, INFO, etc.

4.1.2 Command Line Syntax Rules

In ProcessJ, a command consists of five patterns each listed with various elements:

- `command -option <argument>`
- `command -option=<argument>`
- `command [-option-A|-or-option-B]`
- `command [<optional-argument>]`
- `command <argument>`

These constructs (a combination of commands, options, and positional arguments) and elements (words delimited by white space characters or either one of these: `=`, `[]`, `<>`, and `|`), together, form and describe valid patterns for executing commands on the command prompt. Table 4.1 describes the notations used above.

The following is a summary of the syntax rules that we use for parsing arguments, and building the format for the command usage text of ProcessJ. The general basic rules for specifying commands, options, and parameters in ProcessJ are:

[2] A list of all ProcessJ compile error and warning messages can be found at `https://processj.cs.unlv.edu/`

Table 4.1: Command line syntax notation

Notation	Description
Text without square or angle brackets	Items that must be present as shown
[Text inside square brackets]	Optional items: commands or options
<Text inside angular brackets>	Values for options or parameters
[<Text inside angular and square brackets>]	Optional values for parameters
Equal sign	Separator for option and value
Vertical bar	Separator for mutually exclusive options

- A command must have a unique name consisting of one, or more, alphanumeric characters. Contrary to UNIX commands, the names of commands in ProcessJ are case sensitive and do not have a character limit. Furthermore, it is illegal to leave the name of a command empty or assign the string literal `""` to it.

- An option is a single dash (–) followed by one or more alphanumeric characters; for example, `-debug`, `-verbose`, `-help`, `-l`, `-ls`. Contrary to UNIX options, short and long options always start with a single dash.

- An option must have a unique short and/or long name that must be specified following a single dash; for example, both `-g` and `-debug` represent the same option. Note, similar to commands, names of options are also case sensitive, do not have a character limit, and cannot be assigned the string literal `""`.

- An option may required a value. If no value is provided, a default one must exist. The compiler will throw an exception if a required value is missing from an option or parameter on the command line.

- An option or parameter marked as *required* must be present on the command line[3]. The compiler will throw an exception if a required option or parameter is missing.

- Options cannot be grouped after a single dash; for example, an option of the form `-xyz` is not equivalent to `-x -y -z`. Therefore, the former is an unknown option to the compiler and will cause a runtime error to occur.

[3]At most one positional argument is required by the ProcessJ compiler, namely a `.pj` file. When running `pjc` with no parameters the command usage is displayed.

- Options can appear multiple times if they are defined to support multiple values. Otherwise, a value may be accidentally overridden if a single-value option is entered multiple times; for example, `"-debug -debug -debug"` sets the option flag to be `true`, `false`, and the back to `true` again.

- Options can appear in any order; for example, `-x -y -z` is equivalent to `-z -x -y` and `-y -z -x`. However, if an option requires a value, the value should be entered after the option's name.

- The 'bare double dash' (`--`) marks the end of command options, after which positional arguments are only accepted even if they start with a single dash; for example, both `filename` and `-verbose` are interpreted as positional arguments in the program arguments `"-debug -- filename -verbose"`.

- A zero-based numbering marks the index of a positional argument on the command line; for example, `fileX` is at index 0, `fileY` is at index 1, and `fileZ` is at index 2 in the program arguments `"-aa fileX -bb fileY -cc fileZ"`.

Similarly, the ProcessJ compiler uses the following rules when parsing arguments on the operating command line system:

- Arguments are delimited by white spaces regardless of the amount of white space characters contained within.

- Arguments enclosed in double quotation marks ignore spaces as value separators in the command line; for example, the program arguments $\texttt{"\ \ \ arg}_0\texttt{"}$, $\texttt{"arg}_0\texttt{\ \ \ arg}_1\texttt{"}$, and $\texttt{"arg}_0\texttt{ arg}_1 \ldots \texttt{arg}_n\texttt{"}$ are $\boxed{\texttt{\ \ \ arg}_0}$, $\boxed{\texttt{arg}_0\texttt{\ \ \ arg}_1}$, and $\boxed{\texttt{arg}_0\texttt{ arg}_1 \ldots \texttt{arg}_n}$ where each square box represents a single string literal value.

- Special characters must be preceded by a backslash and be enclosed in double quotation marks; for example, `"\"arg\""` represents the string literal `"arg"`, `"\\"` represents the backslash character (\), and `"\/"` represents the forward slash character (/).

- Using the `ListParser` class to convert a String into a list of values, an argument of the form "$\texttt{arg}_0$,$\texttt{arg}_1$,$\texttt{arg}_2$,...,$\texttt{arg}_n$" can be split into an `ArrayList` of n String values based on one or more comma (,) delimiters.

- A list of key-value argument pairs must be separated by the equal sign (=); for example, the program argument `"a=12 b=23 c=45"` is interpreted as a set of key-value types where `a` maps to `12`, `b` maps to `23`, and `c` maps to `45`.

- If an option has arguments, these arguments must appear immediately after the option's `split=` attribute value; for example, in `"-debug false"`, the value of the *split* attribute is the white space character between the words `debug` and `false`; in `"-debug=false"`, the value of the *split* attribute is the equal sign (=); and in `"-debug:false"`, the value of the *split* attribute is the colon symbol (:).

4.1.3 Command Line Parsing Library

The ProcessJ command line interpreter is a Java annotation-based library modeled after the GNU Getopt library [15] and Argparser Python module [4]. The difference, of course, is that through annotations and reflection, ProcessJ provides a mapping of instance variables to command line arguments. These instance variables can be accessed, operated on, updated, and then used to shape the behavior of the program. In essence, with annotations we provide information to classes and their fields. When the ProcessJ compilers is invoked, our command line interpreter uses this information to parse arguments in the command line. These annotated fields are then mapped to individual arguments, each argument is parsed using the information gathered from the annotated field, and, finally, the field's default value is updated via reflection at runtime.

Figure 4.2 illustrates three stages for building a command in ProcessJ; these stages are annotation, reflection, and builder pattern. In the **annotation** stage, we use 'custom' annotations to define a command and its set of options and arguments in a class. In the **reflection** stage, the command line interpreter maps *descriptions* to options and arguments according to the specifications established in the **annotation** stage. Finally, in the **builder pattern** stage, the program creates instance of options and arguments in a sequence of steps according to the rules defined by the command line interpreter.

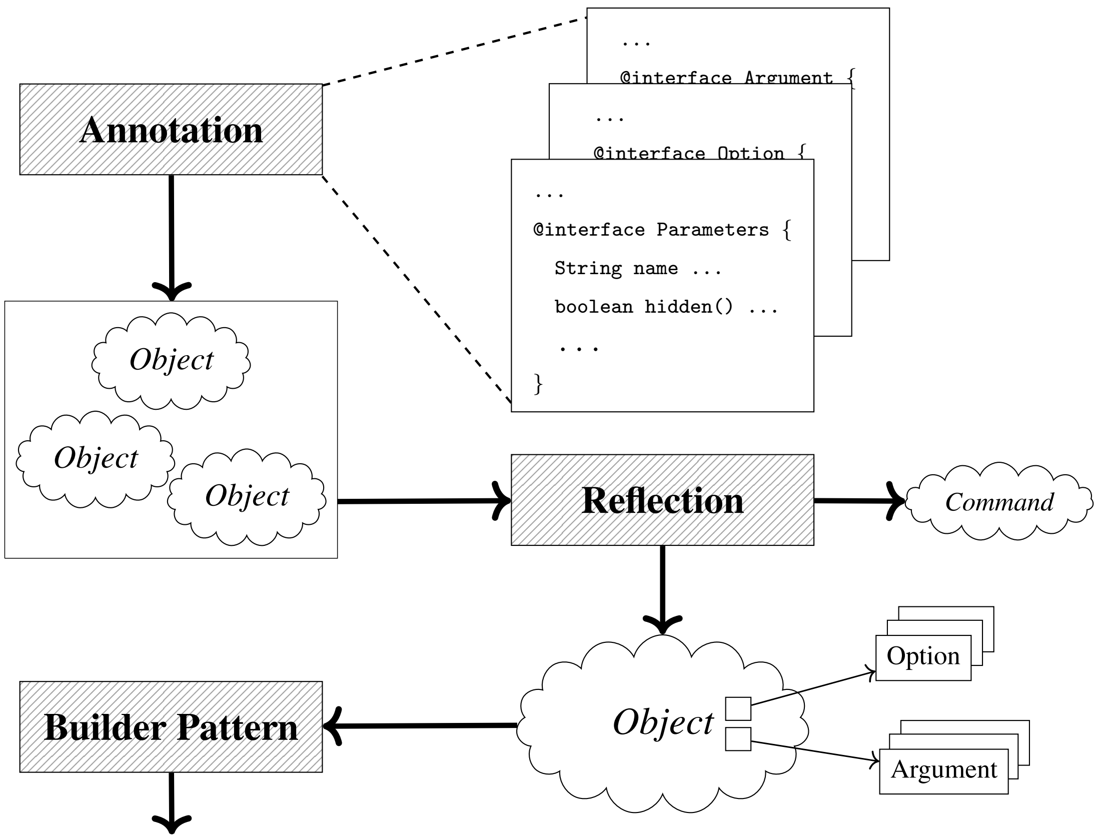

Figure 4.2: Command line parsing library structure.

Before introducing the main components of this library, I will start with a short description of annotations, reflection, and the builder pattern as they were useful when building ProcessJ's command line interpreter.

4.1.3.1 Annotations

A Java annotation [92] is a form of metadata that can be embedded into a source file and later be processed at compiled time or at runtime via reflection. While annotations do not change the semantics of a program – they do not directly affect the operations of the code itself, they can actually be use to affect the way programs are treated by tools and libraries.

We use annotations to 'extend' the Java language by adding additional descriptions to our classes and their fields. This allows us to collect information at compiled time or during runtime and generate code using the information we collect from the annotated classes and fields.

4.1.3.2 Reflection

The Java Reflection API [73] enables a program to see and manipulate itself at runtime. Reflection provides us the means to inspect the structure of classes, interfaces, fields, methods, constructors and their attributes, and determine the capabilities of objects as we modify their runtime behavior. We can also use reflection to make method invocations, instantiate objects, and change field values.

Java Reflection is a powerful tool with an endless list of features, so much that it is too complex to cover in detail. We shall therefore only mention the classes used by the command line interpreter:

- **Class:** This class represents and provides information about classes and interfaces. Therefore, it can be used to examine the runtime properties of the object.
- **Field:** This class provides dynamic access to and information about a field declared in a class or interface, such as its access modifier, name, value, and annotation.
- **Method:** This class provides access to and information about a method declared in a class or interface, such as its access modifier, return type, name, parameter types, and annotation.
- **Constructor:** This class provides access to and information about a constructor for a class, such as its access modifier, name, parameter types, and annotation.
- **Annotation:** This class provides access to and information about all annotation types.

It should be noted that the command line interpreter uses reflection to get access to a class, field, method, and constructor information; to make a method invocation through a Method object that represents the underlying method being invoked, or through a value created at runtime representing an object of the class in which the method is defined; and to make instance fields (variables and methods) accessible, to get the value of a field, to determine the modifiers used on a class, and to create a class object using the constructor defined in the class.

4.1.3.3 Builder Pattern

We use the builder pattern described in [49, 51] to create various objects that required the same type of constructing steps. For example, the construction of an option depends on the number of arguments specified in an annotated field. If there are only two arguments in an annotated field, these two arguments are used to build (say) option A at runtime. Another annotated field, however, could have four arguments instead of two. The process involved in the creation of (say) option B for the second annotated field is the same as option A, but B's internal representation is different. This is shown in Listing 4.4.

```
OptionValue.Builder A = new OptionValue.Builder();
A.addSimpleName(...)
 .addName(...).build();

OptionValue.Builder B = new OptionValue.Builder();
B.addSimpleName(...)
 .addField(...)
 .addArity(...)
 .addOptionType(...).build();
```

Listing 4.4: Building an option.

As seen in Listing 4.4, we can change how an option is built. We do this by indicating the parts that are actually required to construct the option, which simplifies the specifications that are needed during the construction phase.

4.1.3.4 Components

As stated before, we must define which command, options, and arguments are valid in our program. This is required to enable the `CLIBuilder`, the command line interpreter, to parse and issue errors when the command line contains incorrect or invalid input.

Command

A command is a particular type of argument that gets invoked automatically when specified on the command line. Commands are the actions which are supported by the command line interpreter. Consider, for example, some of the built-in commands such as `pjc` and `pj`. Technically, any command specified after the names of these commands is, in fact, a sub-command. In addition, sub-commands to the root command – the name of the executable script (e.g., `pjc`) – may themselves have their own commands. For example, the command `pjc -info cli` have `pjc` as the root command, `-info` as the sub-command, and `cli` as the sub-command to `-info`. Since a command supports sub-parsing which allows for sub-commands, if a program supports several commands and a user enters several of them at the same time, they will be executed in the order in which they were declared.

To integrate a custom command class to the compiler, the command must extend the `Command` class which is the base class for all commands in ProcessJ. Listing 4.5 shows the class and methods declared inside the `Command` class.

```java
public abstract class Command {
    @Override
    public final int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + getClass().hashCode();
        return result;
    }
    @Override
    public final boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null  getClass() != obj.getClass())
            return false;
        return true;
    }
}
```

Listing 4.5: The `Command` class.

The subclass will contain instance fields annotated with @Option and/or @Parameter. Note, there is no direct construction of this class at runtime. That is to say, we do not have to allocate an object of type subclass using the **new** operation. Instead, the `CLIBuilder` is responsible for creating an instance of the class that extends `Command` reflectively. A subclass must therefore never provide a default (no-argument) constructor. An exception is thrown when the `CLIBuilder` is not able to create an instance of a class that sub-classes `Command`. This happens when an invocation of the default constructor of the subclass and super class fails.

Parameters

These are names that can be used to identify commands or options which act as commands. For example, when specifying the include option, we can use any of the following two aliases: `-I` or `-include`. In ProcessJ, parameters is an interface that is used on a command that sub-classes `Command`. We annotate a custom command class with @Parameters so that it gets an invocation of execution by the parser when it is encountered. This annotation is used to specify default settings for a command, and can only be placed on a **class**, an **interface**, or an **enum**. Listing 4.6 shows the settings for this interface.

```
@Inherited
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Parameters {
    String name() default "";
    boolean hidden() default false;
    String help() default "";
    String[] header() default {};
    String[] notes() default {};
    String[] footer() default {};
    String[] version() default {};
    Class<? extends IVersionPrinter> versionPrinter()
            default IVersionPrinter.class;
}
```

Listing 4.6: The `Parameters` interface.

The list of all available settings for a command is:

- name, the name of a command.
- hidden, specifies that this command should, or should not, be included in the help information.
- header, a descriptive text messaged used in help information.
- notes, an optional summary description of a command.

- footer, additional description of a command.
- version, a simple and short version information of a command.
- versionPrinter, a custom version information that gets instantiated when provided. If none is provided then the default specified version information in ProcessJ is used.

Option

As the name implies, options are generally not a required command element. They provide a way to modify the behavior of a command. For example, the `pjc` build command may include the `-console-ansi-code` option, which we can use to enable ANSI color in the console output. In ProcessJ, an option is represented as an interface. It is used to specify default setting for command line options, and it can only be placed on instance fields in classes that subclass `Command`. Since options are parsed in sequence and matched to arguments specified on the command line, we annotate an instance field with @Option so that the parser can set the field's value through its annotation settings. Listing 4.7 shows the settings for this interface.

```java
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface Option {
    String[] names() default {};
    String help() default "";
    String defaultValue() default "";
    String arity() default "";
    String metavar() default "";
    boolean required() default false;
    boolean hidden() default false;
    String split() default "";
    @SuppressWarnings("rawtypes")
    Class<? extends OptionParser>[] handlers() default {};
}
```

Listing 4.7: The `Option` interface.

The list of all available settings for an option is:

- name, the name (or names) of an option.

- help, the descriptive text messaged used in the help information.
- defaultValue, a default value for this option as a String.
- arity, specifies the minimum (and maximum) number of command line argument values an option should consume.
- metavar, the name that represents the values for this argument.
- required, indicates whether an option is required.
- hidden, specifies that an option should or should not be included in the help information.
- split, the separator between an option and its actual value.
- handlers, the *converter* used to parse the value for the option.

Argument

Both commands and options can have associated values. The value associated with a command or option is called the argument. For example, the `pjc` command includes a list of source files. This list of files is required when we specify such command on the command line. Similarly, options may have values associated with them. For example, the `-console-ansi-code` option has an argument for specifying an on/off switch value that enables/disables ANSI color in the console output. In ProcessJ, an argument is an interface similar to the @Option interface, except that it has an index that represents the exact position of a positional argument on the command line. Like options, the @Argument annotation should be placed only on instance fields. Listing 4.8 shows the settings for this interface.

```java
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface Argument {
    String help() default "";
    String defaultValue() default "";
    String order() default "";
    String metavar() default "";
    boolean required() default false;
    boolean hidden() default true;
    String split() default " ";
    @SuppressWarnings("rawtypes")
    Class<? extends OptionParser> handler()
            default OptionParser.class;
}
```

Listing 4.8: The `Argument` interface.

The list of all available settings for an argument is:

- help, the descriptive text messaged used in the help information.
- defaultValue, the default value for this option as a *string*.
- order, specifies the argument on the command line. A field annotated with @Argument must have a specific order. For example, "`order=0,order=1,...,order=`n", or "`order=0..`n" where n is an integer. This is necessary for the parser to capture the exact position of the positional argument from the command line.
- metavar, the name that represents the values for this argument.
- required, indicates whether an option is required.
- hidden, specifies that an option should or should not be included in the help information.
- split, the separator between an option and its actual value.
- handlers, the *converter* used to parse the value for the option.

Factory

The Factory is a **singleton** class consisting of a collection of type/handler pairs associated with registered options and arguments declared in a class that sub-classes `Command`. The job of the Factory is simple. When the program starts, the Factory loads all built-in data types to make them available upon request. The word 'load' in this instance means associating a type name with its corresponding data type. The parser uses the Factory class to parse values for options and arguments specified on the command line.

Since the command line parsing library relies heavily on reflection, the traditional Java-Singleton design pattern described in [50] can easily be destroyed using this feature. We decided to use an *enum* to implement Singleton instead. This is because an *enum* is more concise and provides the serialization mechanism needed against multiple instantiations. An *enum* also prevents serialization and reflection attacks as suggested in [35]. Table 4.2 contains a list of data types that are loaded by the Factory using reflection.

Table 4.2: Default built-in types supported by the command line interpreter

Type Name	Representation	Converter
byte	Byte.class Byte.TYPE	ByteParser
short	Short.class Short.TYPE	ShorParser
character	Character.class Character.TYPE	CharacterParser
integer	Integer.class Integer.TYPE	IntegerParser
long	Long.class Long.TYPE	LongParser
float	Float.class Float.TYPE	FloatParser
double	Double.class Double.TYPE	DoubleParser
big decimal	BigDecimal.class	BigDecimalParser
boolean	Boolean.class Boolean.TYPE	BooleanParser
string	String.class	StringParser
file	File.class	FileParser
url	URL.class	URLParser
enum	Enum.class	EnumParser
list	List.class	ListParser
map	Map.class	MapParser

Custom Handler

Although the parser controls how a string value taken from the command line is mapped to a field, a 'handler' is in charge of translating the string into a typed value suitable for use in our program. To integrate a custom handler class, the handler must extends the abstract `OptionParser` class which is the base class for all handlers in ProcessJ. Instances of type `OptionParser` are used to parse values for options and arguments. Further still, an instance of this class will throw an exception when the value of an option or argument could not be parsed to a type `T` taken from a class declaration. Listing 4.9 shows the class and methods declared inside the `OptionParser` class. An implementation of this class can be seen in Section 4.1.4.

```java
public abstract class OptionParser<T>
                implements IOptionParser<T> {
    protected final String optionName;
    public OptionParser(String optionName) {
        this.optionName = optionName;
    }
    public String getParam() {
        return optionName;
    }
}
```

Listing 4.9: The `OptionParser` class.

4.1.4 A Command Line Example

The example program in Listing 4.10, emphasizes some of the basics concepts for developing a command line parser for the ProcessJ compiler. It illustrates the anatomy of a class that extends `Command` along with declarations of options and parameters within the class body. Note, in the next sections, I will show code snippets for this example. The entire source code is found in Appendix F.

```
@Parameters(name="calc")
public class Example extends Command {
   @Option(names="-op1",
           help="first operand",
           split="=",
           metavar="<num>",
           arity="1",
           handlers=OperandParser.class)
   public int op1;
   @Option(names="-op2",
           help="second operand",
           metavar="<num>",
           arity="1",
           handlers=OperandParser.class)
   public int op2;
   @Option(names="-add",
           help="Adds two numbers",
           defaultValue="false")
   public boolean addition;
   @Option(names="-sub",
           help="subtract two numbers")
   public boolean subtraction;
   @Option(names="-help",
           help="Show this help message and exit",
           defaultValue="false")
   public boolean help;
   ...
}
```

Listing 4.10: A command line example.

This example is used to create a command line utility, called `calc`, that can ‘add’ or ‘subtract’ two integers. We should emphasize that this command is not, in any way, part of the list of commands available in ProcessJ. We only use this program to show how easy it is to create a command in ProcessJ. We will further demonstrate how this program matches and parses values out of the argument array of the *main*() method, terminates if there are errors, computes the result of an addition or subtraction operation, and prints a nicely-formatted list of options and parameters each having a short description that shows their names, default values, required arguments, etc.

Breaking Down the Command Line Example

The structure of the command line example is made of:

- One **command name**, in this case, a string used to reference the command `calc`.
- Three **flag options**; an option `-add` that adds two integers, an option `-sub` that subtract two integers, and an option `-help` that displays a command usage.
- Two **single-value options**; an option `-op1` that takes a number and throws an exception when anything but a number is given, and an option `-op2` which also takes a number and throws the exact exception when a number is not given.

Using the above information, the command line interpreter can generate a help message and issue errors when parsing incorrect or invalid input. Table 4.3 lists the main components of the program, which are used by the `CLIbuilder` during the parsing phase. From these pieces of information, we can define which commands, options, and parameters the compiler must accept before building the parser in line 28 (Listing 4.11).

Table 4.3: Main components of the example program

Type	Name	Parameters	Arity
command	`calc`	option/argument	—
option	`-add`	Boolean	0
option	`-sub`	Boolean	0
option	`-op1`	integer	1
option	`-op2`	integer	1
option	`-help`	Boolean	0

The following should be noted: First, our command does not support positional arguments, thus, any remaining arguments on the command line will cause an exception. Second, none of the above options are required, thus, they do not need to be specified by the user on the command line as they all are optional. Finally, even if options have no default values assigned to them, variables declared as fields (be it static or instance variable of a class) are initialized with a default value by the runtime system in Java.

Walkthrough the Command Line Example

First, it is necessary to register our command. We do this by calling the *addCommand*() method with the runtime class of `Example` as its parameter. This is illustrate in line 28 (Listing 4.11). Suppose we execute the following command:

```
pjc calc -add -op1=13 -op2 34
```

The `args` array is passed to the *handlerArgs*() method in line 31 to do the actual parsing. The parser first splits the arguments according to the description specified in each option and parameter. Figure 4.3 shows the further decomposition of the command line arguments passed to the *main*() method. After splitting the arguments, the parser attempts to match an argument from the decomposed `args` array to one of the specified options on the command line. When a match for an option is found, and if the option requires a value, the next argument from the decomposed `args` array is grabbed. The parser will then attempt to parser this argument. If the argument fails to be parsed, an exception is thrown. Otherwise, the parser continues until all specified options are matched and all of their values successfully parsed.

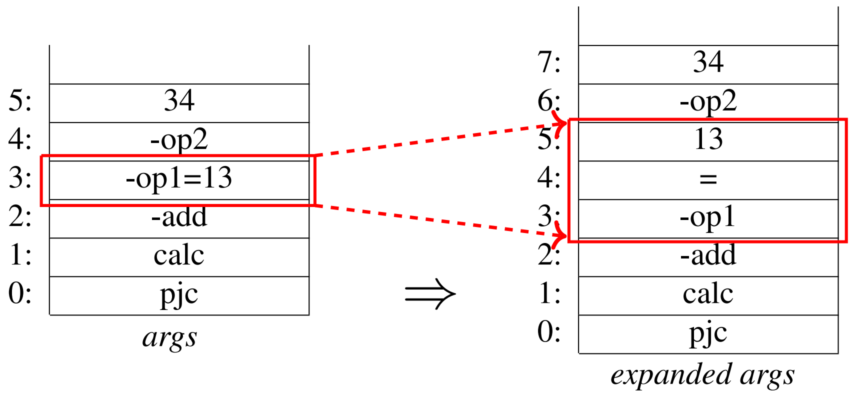

Figure 4.3: Expanded command arguments.

Once the command and all of its options are finally matched, including the options' values, lines 37 and 38 are executed (Listing 4.11).

```java
27:  public static void main(String[] args) {
28:     CLIBuilder builder =
           new CLIBuilder().addCommand(Example.class);
29:      Example sp = null;
30:      try {
31:         builder.handleArgs("-add -op1 13 -op2 34"
                               .split(" "));
32:         sp = builder.getCommand(Example.class);
33:      } catch(Exception e) {
34:         System.out.println(e.getMessage());
35:         System.exit(1);
36:      }
37:      if (sp.addition) {
38:         System.out.println(String.format("Add operation: " +
39:         "%s + %s = %s", sp.op1, sp.op2, (sp.op1 + sp.op2)));
40:      } else if (sp.subtraction) {
41:         System.out.println(String.format("Add operation: " +
42:         "%s + %s = %s", sp.op1, sp.op2, (sp.op1 + sp.op2)));
43:      }
44: ...
```

Listing 4.11: Executing the `-add` option.

To display the command usage, we execute the following command:

```
pjc calc -help
```

After the option `-help` is matched, lines 35, 36, and 37 are executed in Listing 4.12.

```java
43: ...
44:      if (sp.help) {
45:         Formatter formatHelp = new Formatter(builder);
46:         System.out.println(formatHelp.buildUsagePage());
47:         System.exit(0);
48:      }
49: ...
```

Listing 4.12: Executing the `-help` option.

```
> pjc calc -help
usage: calc [-add] [-help] [-op1=<num>] [-op2 <num>] [-sub]

parameters:

options:
 -add           Adds two numbers (default=false)
 -help          Show this help message and exit (default=false)
 -op1=<num>     first operand
 -op2 <num>     second operand
 -sub           subtract two numbers
```

Figure 4.4: Command usage message.

The help message is generated and formatted as shown in Figure 4.4.

Suppose we now invoke the command as follows:

```
pjc calc -add -op1=1r -op2 34
```

Figure 4.5 shows the error message generated after invoking the above command with the wrong arguments.

```
> pjc calc -add -op1=1r -op2 34
"-op1" could not convert "1r" to Integer.
```

Figure 4.5: Command error message.

Consider making one last invocation to this command with the following arguments:

```
pjc calc -add -o=13 -op2 34
```

Figure 4.6 shows the auto-completion feature in action. The parser uses auto-completion to provide a list of suggestion to the user when an incomplete or invalid command is entered.

A nice feature of the command line interpreter is that it can perform simple checks on the given arguments and complete or make corrections of commands that begin with a specific set of characters. While a lot has been achieved, the future work section contains more information on what is to come.

```
> pjc calc -add -o=13 -op2 34
Unknown @Option "-o" for @Parameters "calc". Did you mean to say?
-op1
-op2
```

Figure 4.6: Command auto-complete message.

4.1.5 Options and Parameters in ProcessJ

The list of option and parameters along with their descriptions can be seen in Appendix G.

Chapter 5

Runtime System and Code Generation

Before considering the code generation, we need to understand what the ProcessJ compiler – we referred to it as the ProcessJ-JVM compiler – tries to generate. The code generation is in reality a framework consisting of various component parts (templates) which can be customized individually. Such components are then put together to generate Java code that after compilation is instrumented and rewritten. In order to build a java class from a ProcessJ file, the ProcessJ compiler generates actual Java source code, and then invokes a Java compiler using this source code to create `.class` files. Of course, these class files are executed by the JVM which executes the ProcessJ program in turn. It should be clear that access to a Java compiler is needed due to the code generated for the JVM.

In this chapter, I describe the ProcessJ's scheduler and its constituent parts, as well as the overall design of the Java runtime system. In addition, I explain how classes are created using templates to generate compilable Java code.

5.1 Bytecode Rewriting

The ProcessJ compiler generates Java source code that is further compiled with the Java compiler to produce class files. These class files are then rewritten using the ASM [6, 38] bytecode rewriting tool to handle yielding and resumption. Despite the fact that Java does allow labels, which can be used with *breaks* and *continues* to transfer the flow of control to labels, it does not allow labels for explicit *gotos*. However, explicit jumps in Java source can be achieved in ProcessJ through the **Instrumenter** – a class that implements the ASM library to dynamically generate classes directly in bytecode form. We use the Instrumenter to mark `labels` along with `resumes` and `yields` using 'dummy' method calls which are replaced by actual *gotos*, *labes*, and *returns* in the bytecode. This is done to allow the code with explicit yield and resume points to be scheduled cooperatively. `label` is a dummy method used by the ASM tool to record the location where the code should resume depending on the value of the `runLabel`. These labels are eventually removed from the generated code. In the same way, `resume` invocations are replaced with `goto` instructions to perform jumps to the appropriate place in the code. Finally, `yield` is translated into a *return* statement by the bytecode rewriting along with some logic to set the `runLabel`. For example:

```
  ...
  yield(1);
L1:
```

Where 1 represents the `runLabel` that the yield returns. Note that we represent the label as `L1:`, but in the generated code we use a dummy method called `label(...)`. Therefore, when a process is scheduled to run again, it must continue from right after the `yield(...)` method that suspended the process – it should continue from `L1` in the code above.

Consider the implementation of the *prefix* process shown in Section 2.2.3 (Listing 5.1). The equivalent Java code and byte code rewriting is explained in this section.

```
public void prefix(int initVal,
                   chan<int>.read in,
                   chan<int>.write out) {
  out.write(initVal);
  while (true) {
    int x;
    x = in.read(); // read from Delta
    out.write(x); // write to Plus
  }
}
```

Listing 5.1: The `prefix` process – revisit.

Figure 5.1 shows the code generated for the *prefix* process. After compilation, the cases in the switch statement (labeled with '∗') are adjusted to jump to the locations of the *label*() method. A *label*() method is an empty method used to mark the address of a particular place in the program. Note that each case in the switch is followed by a call to an empty *resume*() method. A *resume*() method is a goto instruction to which the jumps of the switch statement will jump. These are labeled with '−'.

```
  public static class _proc$prefix extends PJProcess {
     /** Locals and parameters transformed into fields */
     protected int _pd$initVal1;
     protected PJChannel<Integer> _pd$in2;
     protected PJChannel<Integer> _pd$out3;
     protected int _ld$x1;

     public _proc$prefix(int initVal, PJChannel<Integer> in, PJChannel<Integer> out,
                         int x) {
         _pd$initVal1 = initVal;
         _pd$in2 = in;
         _pd$out3 = out;
         _ld$x1 = x;
     }

     @Override
     public synchronized void run() {
         /** Jumps */
         switch(runLabel) {
         case 0: break;
*        case 1: resume(1); break;
*        case 2: resume(2); break;
*        case 3: resume(3); break;
*        case 4: resume(4); break;
         default: break;
         }
         /** Write operation */
         _pd$out3.write(this, _pd$initVal1);
         runLabel = 1;
         yield();
−        label(1);
         /** Read and write operations */
         while (TRUE) {
             if(!_pd$in2.isReadyToRead(this)) {
                 runLabel = 2;
                 yield();
             }
```

Figure 5.1: Java implementation of the *prefix* process.

```
                      /** Read operation */
–                     label(2);
                      _ld$x1 = _pd$in2.read(this);
                      runLabel = 3;
                      yield();
                      /** Write operation */
–                     label(3);
                      _pd$out3.write(this, _ld$x1);
                      runLabel = 4;
                      yield();
–                     label(4);
                } // end while
          } // end run method
      } // end prefix class
```

Figure 5.2: Java implementation of the *prefix* process continuation.

Labels

The code generated for such labels (the invocation of a dummy `label` method) has the form as illustrated in Figure 5.3, where each label can be substituted for cases 1, 2, 3, etc. (line 100 indicates that it is label 1 because the constant 1 is loaded onto the stack) in the switch statement.

```
 99:   aload_0
100:   iconst_1
101:   invokevirtual label:(I)V
 ...
130:   aload_0
131:   iconst_2
132:   invokevirtual label:(I)V
 ...
162:   aload_0
163:   iconst_3
164:   invokevirtual label:(I)V
 ...
```

Figure 5.3: *label*() invocation before the bytecode rewriting.

All label invocations, like the ones in Figure 5.3, should become `nop` instructions as illustrated in Figure 5.4.

```
 99:   nop   // was aload_0
100:   nop   // was iconst_1
101:   nop   // was invokevirtual label:(I)V
 ...
130:   nop   // was aload_0
131:   nop   // was iconst_2
132:   nop   // was invokevirtual label:(I)V
 ...
162:   nop   // was aload_0
163:   nop   // was iconst_3
164:   nop   // was invokevirtual label:(I)V
 ...
```

Figure 5.4: *label*() invocation after the bytecode rewriting.

Switch

The switch statement is translated into a **tableswitch** instruction like the one shown in Figure 5.5. The labels are incorrect and must be replaced by those we mentioned earlier. We used the ASM bytecode rewriting tool to locate the addresses of all label invocations and replace then by `nop` instructions. Similarly for resume invocations, we replace them with `goto` instructions (as shown in Figure 5.6 and Figure 5.7) which transfer control to the appropriate address determined by the location of the label invocations.

4: tableswitch 0 to 4 0: 40 1: 43 2: 51 3: 59 4: 67 default: 75	4: tableswitch 0 to 4 0: 40 1: 43 2: 54 3: 65 4: 76 default: 87

Figure 5.5: *resume*() invocation before (left) and after (right) the bytecode rewriting.

```
43:   aload_0
44:   iconst_1
45:   invokevirtual resume:(I)V
48:   goto 75
...
```

Figure 5.6: *resume*() invocation before the bytecode rewriting.

```
43:   nop          // was aload_0
44:   nop          // was iconst_1
45:   goto 101     // was invokevirtual resume:(I)V
48:   goto 75
...
```

Figure 5.7: *resume*() invocation after the bytecode rewriting.

Returns

We implement a return following yield invocations. The generated bytecode looks like the code shown in Figure 5.8.

```
107:   aload_0
108:   invokevirtual yield:()V
111:   goto 227
 ...
223:   aload_0
224:   invokevirtual terminate:()V
227:   return
```

Figure 5.8: *yield*() invocation after the bytecode rewriting.

5.2 The Runtime Scheduler

To make use of the JVM as an execution platform for processes that are willing to give up the CPU, our system relies on *cooperative non-preemptive scheduling*. As already discussed, ProcessJ uses a very simple cooperative scheduler running a single tread at any one time. The **scheduler** interacts with a **run queue** in which process are added to the tail of the queue when they yield and are removed from the head of the queue when they are ready to run again. In addition, a process can add new processes to the run queue, which usually occurs when the compiler generates code for a par-block or a par-for. It should be noted that the thread executing the process is that of the scheduler. A process must therefore, after some time, voluntarily return control to the scheduler by giving up the CPU.

Although the scheduler runs its own thread, at the same time as the scheduler's thread is running, a second Java thread is run individually, namely, the **timer handler**. The timer handler is a separate thread that consistently attempts to remove expired timer objects from the **timer queue**. These timer objects are used to mark processes that are ready to run again after a present time. Furthermore, much like the run queue, timer objects are only removed from the head of the timer queue when their *delay* expires. When a timer object expires, the timer handler sets the process with expired timer object ready to run again. This process is eventually executed by the scheduler when it gets to the head of the run queue. Figure 5.9 illustrates the interaction between these four components.

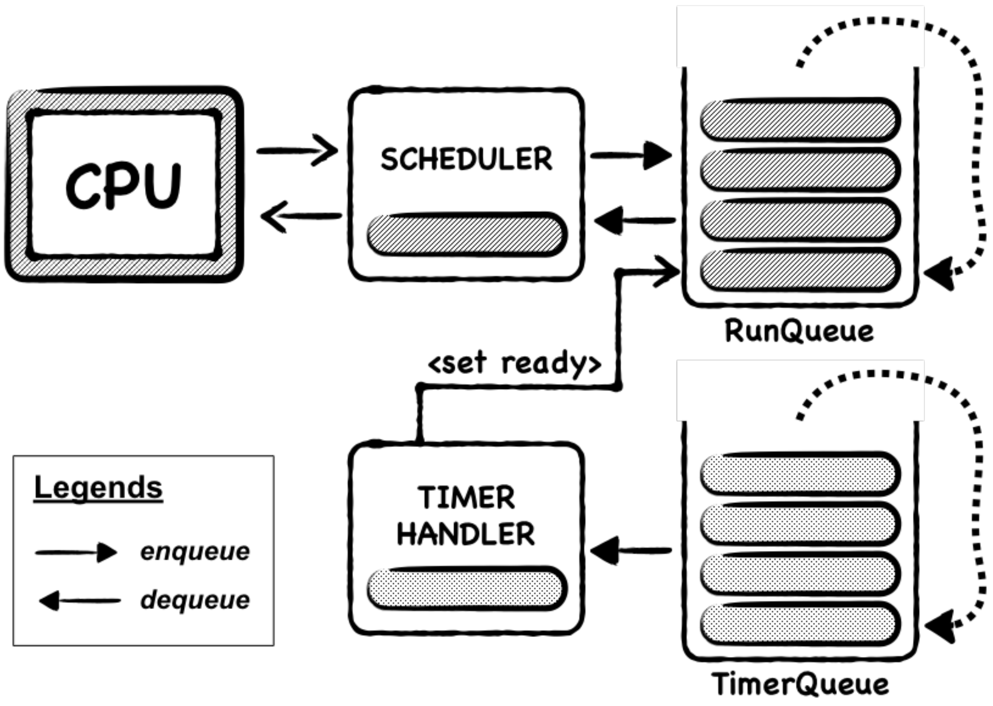

Figure 5.9: Runtime system overview.

5.3 The CSP Runtime System for Java

This sections introduces the set of classes built for the Java runtime system – we refer to them as the JVMCSP – and a code generation scheme, which, as it names suggests, turns ProcessJ code to Java.

5.3.1 StringTemplate Library

We assemble Java code using the visitor pattern technique [49] together with a number of *templates* each having a unique name. A template consists of various pieces of text and attribute expressions, which are combined and rendered to text using the StringTemplate library [77]. To create a compilable Java file, the template engine replaces the **attributes** of a template with the proper intermediate code given by a visitor object at runtime. These attributes, which may be untyped values or template instances, are evaluated only when asked to during rendering. Typically, this occurs when the ProcessJ compiler access an argument, namely an attribute name, used by a template expression in a tree-traversal method.

Listing 5.2 shows a piece of template code that gets rendered by the ProcessJ compiler after a tree-traversal operation. Note how this template definition is similar to a function definition in most programming languages. However, it does not have a return type, and the list of input parameters is a comma-separated list of just names. When declaring a template with parameters, we replace a parameter with a value by using a set of angular brackets (<>) in the template region (the body of the template). For example, in the `BinaryExpr` template, each argument (*lhs*, *rhs*, and *op*) is replaced with the expression returned by a visitor object during the traversal of a tree node.

```
BinaryExpr(lhs, rhs, op) ::= <<
<lhs> <op> <rhs>
>>
```

Listing 5.2: The template for a ProcessJ binary expression.

Listing 5.3 shows a tree-traversal method that represents an expression that has a binary operator. The visitor object creates an instance of a template class in line 5, namely `BinaryExpr`. We reference the attributes of this template in lines 11, 12, and 13, and then we ‘inject’ the appropriate values, respectively. Finally, we ‘render’ the proper Java code, that is, we generate some text after combining all other rendered code, in line 15, which is then return as a string. An example of a ProcessJ program with a binary expression can be seen in Listing 5.4. The generated piece of code after evaluating the binary expression in shown in Listing 5.5.

```
 1: public T visitBinaryExpr(BinaryExpr be) {
 2:     Log.log(be.line + ": Visiting a BinaryExpr");
 3:
 4:     // Generated template after evaluating this visitor
 5:     ST stBinaryExpr = _stGroup.getInstanceOf("BinaryExpr");
 6:     String op = be.opString();
 7:     String lhs = (String) be.left().visit(this);
 8:     lhs = be.left().hasParens ? "(" + lhs + ")" : lhs;
 9:     String rhs = (String) be.right().visit(this);
10:     rhs = be.right().hasParens ? "(" + rhs + ")" : rhs;
11:     stBinaryExpr.add("lhs", lhs);
12:     stBinaryExpr.add("rhs", rhs);
13:     stBinaryExpr.add("op", op);
14:
15:     return (T) stBinaryExpr.render();
16: }
```

Listing 5.3: The binary expression visitor.

```
public void main(string args[]) {
    int a = ((5 + 4) * 6) - 7;
}
```

Listing 5.4: ProcessJ code with a binary expression.

```
...
_ld$a1 = ((5 + 4) * 6) - 7;
...
```

Listing 5.5: Example of a binary expression generated by a string template.

Lastly, certain attributes in a template may need to be evaluated, be it to determine types, create variables, etc., before they are rendered during the traversal of a parse-tree node. To test if an attribute has a value or is a Boolean object that evaluates to `true`, we use a conditional expression such as `<if(...)>`. Note that the `...` is a place holder for some attribute whose value is rendered if and only if it has one. For example, the attribute *body* in Listing 5.6 is rendered if the conditional expression evaluates to true.

```
...
<if(body)><body; separator="\n\n"><endif>
...
```

Listing 5.6: Evaluating the presence or absence of an attribute's value.

5.3.2 JVMCSP Runtime Components

Like occam/occam-π before it, at the heart of any ProcessJ program are processes and channels – processes being pieces of code that can be executed sequentially or concurrently and channels being the means of communication between processes. ProcessJ, however, takes an object-oriented approach to implementing CSP. The core of the JVMCSP runtime system is composed of the following set of CSP primitives implemented as Java classes: `PJProcess`, `PJChannel` (from which `PJOne2OneChannel`, `PJOne2ManyChannel`, `PJMany2OneChannel`, and `PJMany2ManyChannel` are derived), `PJBarrier`, `PJPar`, `PJTimer`, `PJRecord`, and `PJProtocolCase`.

Since the runtime system of ProcessJ is written in Java, we use classes and objects to create complex functionality for layered networks of communicating processes. In reality, these layers of composable processes are objects of type `PJProcess`. This means that we use an object to **maintain** the encapsulation of data and algorithms for managing that data (which is something that is easily lost in OOP) using the above primitives. In addition, the interaction between processes is only possible through communication channels (objects of a derived `PJChannel` type used to transmit any assignable type, such as a primitive, a record or a protocol) and barriers (an object of type `PJBarrier`) on which parallel processes enroll, synchronize, and resign. This is due to classes requiring to be set up before processes can run sequentially or concurrently. Therefore, we use channels and barriers as means by which objects communicate (can access shared data) without invoking each other's methods.

5.3.2.1 State Management

Procedures, parameters, and locals are encoded into unique names so that the JVM can separate common names in a ProcessJ program. This is also done to facilitate method overloading and visibility of variables and methods within different scopes. Additionally, local and parameters are transformed into fields in the generated Java code. Procedures, on the other hand, are transformed into classes if they yield or regular Java static methods if they do not. A field representing a local variable will have a name of the form `_ldXname`, where `_ld` stands for 'local declaration', `X` is an integer number (a number associated with every new local variable declaration), and `name` represents the actual name of the local. In much the same way, formal parameters are transformed in to fields but with `_pd` prefixed, where `_pd` stands for 'parameters declaration'. Furthermore, a class representing a procedure that yields will have a name of the form `_proc$name`, where `_proc` stands for 'procedure' and `name` represents the actual name of the procedure. A static method representing a regular procedure is transformed in a similar manner but with `_method` prefixed.

5.3.2.2 Process

An instance of `PJProcess` encapsulates data and algorithms for managing that data, where both its data and algorithms are private to the outside world. A process is therefore an instance of a class that extends the Java class called `PJProcessJ` (Listing 5.7), and its actions are defined by the *run*() method in line 7. To create an object of this type, the *run*() method must be implemented in the derived `PJProcessJ` class. The scheduler calls the *run*() method in order to execute the process until it voluntarily yields or terminates.

```
1: public class PJProcess {
2:   protected int runLabel = 0;
3:   protected boolean ready = true;
4:   protected boolean terminated = false;
5:   public static Scheduler scheduler;
6:
7:   public void run() {
8:   };
9: ...
```

Listing 5.7: The `PJProcess` class.

Any parameters passed to the process must be passed to the constructor of the derived class as its *run*() method takes no parameters. This allows parameters to be kept as fields of the `PJProcess` subclass. In addition, all local variable declarations that appear in the body of the procedure are translated into fields. These fields are used to maintain the local state of the procedure between invocations. The `PJProcess` class contains a *yield*() method in line 43 (Listing 5.9), which is called when a process wants to yield and be descheduled. When a process yields or is not ready to run, it is put back in the run queue using the *schedule*() method in line 10 (Listing 5.8). A call to this method moves the suspended or not-ready to run process to the end of the scheduler's process queue for future scheduling. This process queue is accessed through a static object reference, namely, the `scheduler` variable in line 5 (Listing 5.7).

As mentioned previously, a process must voluntarily give up the CPU and return control to the scheduler so that it can run a different process. This can be done by returning control to the scheduler through a **return** statement. Naturally, yielding is just a return statement. Whenever a synchronization primitive is called, a process voluntarily yields by setting itself ready or not-ready to run. The process can be set ready or not-ready to run by using the *setReady*() method in line 21 (Listing 5.8), and the *setNotReady*() method in line 28 (Listing 5.8), respectively. Waiting is performed via the *yield*() method in line 43 (Listing 5.9). This call sets the *runLabel* in line 2 (Listing 5.7), which is used for resumption when the process is rescheduled. The *runLabel* is simply an address that marks the point where the process should resume execution. The *resume*() method in line 49 (Listing 5.9) is used as a 'goto' label to transfer control to the appropriate address determined by the location of the *label*() method in line 46 (Listing 5.9).

```
10:    public void schedule() {
11:        scheduler.insert(this);
12:    }
13:
14:    public void finalize() {
15:    }
16:
17:    public boolean isReady() {
18:        return ready;
19:    }
20:
21:    public synchronized void setReady() {
22:        if (!ready) {
23:            ready = true;
24:            scheduler.inactivePool.decrement();
25:        }
26:    }
27:
28:    public void setNotReady() {
29:        if (ready) {
30:            ready = false;
31:            scheduler.inactivePool.increment();
32:        }
33:    }
34: ...
```

Listing 5.8: The `PJProcess` class – scheduling and descheduling a process.

The *finalize*() method in line 14 (Listing 5.8) is executed when the process terminates. A process in a par-block, for example, uses the *finalize*() method to decrement the number of running processes wrapped in the par. This way, we can assure that the process containing the par-block will be set ready to run and then rescheduled again when all of its processes have terminated. Additionally, the *terminate*() method in line 35 (Listing 5.9) sets the field *terminated* in line 4 (Listing 5.7) to `true` to indicate that the process has terminated, while the *terminated*() method in line 39 (Listing 5.9) returns `true` if and only if the process has in fact terminated.

```java
    public void terminate() {
        terminated = true;
    }

    public boolean terminated() {
        return terminated;
    }

    public void yield() {
    }

    public void label(int label) {
    }

    public void resume(int label) {
    }
}
```

Listing 5.9: The `PJProcess` class – yielding and terminating a process a process.

Template Layout

Listing 5.10 shows an example of a simple ProcessJ program with a main procedure that concurrently runs processes inside the par-block. The *main*() procedure starts some processes concurrently. The *foo*() procedure is passed the writing-end of a channel which in turn sends the value 4 across the same channel. Since the statement in line 3 performs a write operation, such a statement will generate code that yields. The procedure will then implicitly yield, thus it will be transformed into a class; that is, a class that the extends `PJProcess`. Listing 5.11 shows the generated Java code for `foo`. The template in Listing 5.14 is used to generate this class.

```
public void foo(chan<int>.write out) {
    ...
    out.write(4);
}

public void main(string args[]) {
    chan<int> c;
    par {
        ...
        foo(c.write);
    }
}
```

Listing 5.10: Example of a simple `PJProcess` program.

```java
public static class _proc$foo$cwI extends PJProcess {
    protected PJChannel<Integer> _pd$out1;

    public _proc$foo$cwI(PJChannel<Integer> _pd$out1) {
        this._pd$out1 = _pd$out1;
    }

    @Override
    public synchronized void run() {
        switch (this.runLabel) {
            case 0: break;
            case 1: resume(1); break;
            default: break;
        }

        _pd$out1.write(this, 4);
        this.runLabel = 1;
        yield();
        label(1);

        terminate();
    }
}
```

Listing 5.11: The generate the Java class for `foo`.

```
ProcClass(name, types, vars, ltypes, lvars, methods, main,
            switchBlock, syncBody) ::= <<
public static class <name> extends PJProcess {

     <if(vars)><types,vars:{t,v | protected <t> <v>};
      separator=";\n">;<\n><endif>

     <if(lvars)><ltypes,lvars:{t,v | protected <t> <v>};
      separator=";\n">;<\n><endif>

     <if(methods)><methods; separator="\n"><\n><endif>

     public <name>(<types,vars:{t,v | <t> <v>}; separator=", ">) {
          <! Initialize member fields !>
          <if(vars)><vars:{v | this.<v> = <v>};
           separator=";\n">;<endif>
     }

     <! Synchronized run method !>
     @Override
     public synchronized void run() {
          <if(switchBlock)><switchBlock; separator="\n"><endif>
          <if(syncBody)><syncBody; separator="\n"><endif>
          terminate();
     }

<! Entry point of the program, e.g.,
    public static void main(String[] args) { ... }
!>
<if(main)>}<\n><\n><main><else>}<endif>
>>
```

Listing 5.12: The template for a ProcessJ process.

Using the attributes of the `ProcClass` template in Listing 5.12, the ProcessJ compiler translates `foo` to the Java class in Listing 5.13.

```java
public static class _proc$foo$cwI extends PJProcess {
    protected PJChannel<Integer> _pd$out1;
...
```

Listing 5.13: The `PJProcess` class generated from a template – Java class.

Note that the name of the procedure is modified so that the JVM can separate common names that may (or may not) belong to the same compilation unit. In Listing 5.14, the attribute *name* is then replaced with the modified name of the procedure in line 3. Additionally, all locals and parameters are transformed into fields in the generated code in lines 5 and 8 (Listing 5.14). Line 5 takes in a list of types and variable names, namely, the parameters, and renders each value along with the `protected` modifier. Similarly for line 8, however, the values being rendered are local variables instead.

```
 1: ProcClass(name, types, vars, ltypes, lvars, methods, main,
 2:           switchBlock, syncBody) ::= <<
 3: public static class <name> extends PJProcess {
 4:
 5:     <if(vars)><types,vars:{t,v | protected <t> <v>};
 6:      separator=";\n">;<\n><endif>
 7:
 8:     <if(lvars)><ltypes,lvars:{t,v | protected <t> <v>};
 9:      separator=";\n">;<\n><endif>
10:
11:     <if(methods)><methods; separator="\n"><\n><endif>
12: ...
```

Listing 5.14: The template for a ProcessJ process – fields.

To keep the state of all parameters between invocations (yields/resumes), the constructor in Listing 5.15 must be defined for the generated class.

```
4:     public _proc$foo$cwI(PJChannel<Integer> _pd$out1) {
5:         this._pd$out1 = _pd$out1;
6:     }
7: ...
```

Listing 5.15: The `PJProcess` class generated from a template – constructor.

In Listing 5.16, line 13 does exactly that. We generate a constructor for the class as follows: the attribute *name* is replaced with the modified name of the procedure, followed by the same kind of variables rendered in line 5. These values become the actual parameters passed to the constructor of the procedure when it is dynamically allocated. As with lines 5 and 8 (Listing 5.14), line 13 takes in the same list of parameters. This time, however, each value is rendered with the `this` keyword prefixed in line 15. This is done to avoid shadowing an instance field by a parameter.

```
13:     public <name>(<types,vars:{t,v | <t> <v>}; separator=", ">) {
14:         <! Initialize member fields !>
15:         <if(vars)><vars:{v | this.<v> = <v>};
16:          separator=";\n">;<endif>
17:     }
18: ...
```

Listing 5.16: The template for a ProcessJ process – constructor.

Recall, the *runLabel* marks the starting point of the process being scheduled, and the *resume*() method marks the point after the statement that suspended the process (a call to the *yield*() method). In order to instantiate this class, we need to generate code for the *run*() method in Listing 5.17.

```
    @Override
    public synchronized void run() {
        switch (this.runLabel) {
            case 0: break;
            case 1: resume(1); break;
            default: break;
        }

        _pd$out1.write(this, 4);
        this.runLabel = 1;
        yield();
        label(1);

        terminate();
    }
```

Listing 5.17: The `PJProcess` class generated from a template – run method.

We do that in line 21 in Listing 5.18. The attribute *SwithBlock* in line 22 takes in a template instance that renders a label and a number of resume points to which the label of the **switch** statements jumps to. Furthermore, the attribute *syncBody* in line 23 takes in a template instance that renders the body of the procedure, which gets evaluated during the traversal of a tree node. The generated section of this code can be seen in Listing 5.17.

```
19:     <! Synchronized run method !>
20:     @Override
21:     public synchronized void run() {
22:         <if(switchBlock)><switchBlock; separator="\n"><endif>
23:         <if(syncBody)><syncBody; separator="\n"><endif>
24:         terminate();
25:     }
```

Listing 5.18: The template for a ProcessJ process – run method.

All ProcessJ programs require a main procedure in order to run such as the one in Listing 5.19.

```
    public static void main(String[] _pd$args1) {
        Scheduler scheduler = new Scheduler();
        PJProcess.scheduler = scheduler;
        (new simple._proc$main$arT(_pd$args1)).schedule();
        PJProcess.scheduler.start();
    }
```

Listing 5.19: The `PJProcess` class generated from a template from a template – main method.

In Listing 5.20, the attribute *main* in line 30 generates code for a procedure that represents the entry point in a ProcessJ program. Like the attributes *SwithBlock* and *SyncBody*, *main* takes in a template instance, namely, the template in Listing 5.21; however, it only generates code if it has a value to render.

```
27: <! Entry point of the program, e.g.,
28:    public static void main(String[] args) { ... }
29: !>
30: <if(main)>}<\n><\n><main><else>}<endif>
31: >>
```

Listing 5.20: The template for a ProcessJ process – main procedure.

```
Main(class, name, types, vars) ::= <<
public static void main(<types,vars:{t,v | <t> <v>};
    separator=", ">) {
    Scheduler scheduler = new Scheduler();
    PJProcess.scheduler = scheduler;
    (new <class>.<name>(<vars; separator=", ">)).schedule();
    PJProcess.scheduler.start();
}
```

Listing 5.21: The template for a ProcessJ process – main.

5.3.2.3 Channel

ProcessJ supports four different kind of channels, where each channel sub-classes the abstract template class (the mechanism for generic programming in Java) called `PJChannel` shown in

Listing 5.22. This abstract class provides methods for constructing these different kind of channels. Considering that the functionality of the reading and writing methods is ‘abstract’, the sub-classes are responsible for providing the implementation in which each method operates. The following classes represent these four different kind of channels: `PJOne2OneChannel`, `PJOne2ManyChannel`, `PJMany2OneChannel`, and `PJMany2ManyChannel`. The class diagram in Figure 5.10 shows the relationship of `PJChannel` and its children.

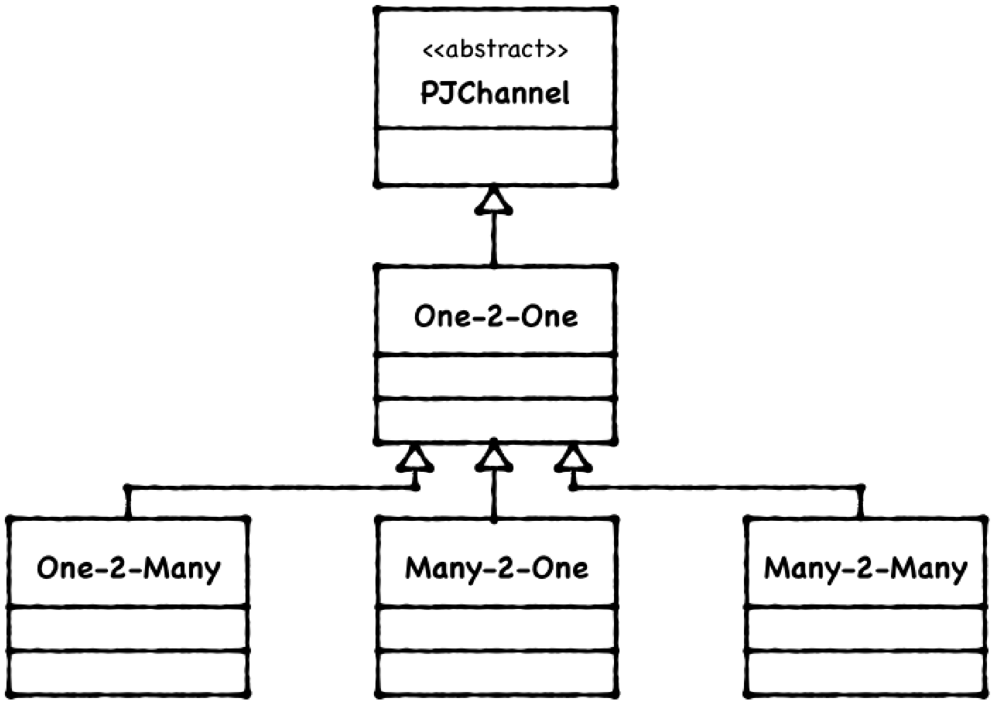

Figure 5.10: Class diagram of the different channels in ProcessJ.

```java
public abstract class PJChannel<T> {

    protected T data;
    protected PJChannelType type;

    public abstract void write(PJProcess p, T data);

    public abstract T read(PJProcess p);

    public abstract boolean isReadyToRead(PJProcess p);

    public abstract boolean isReadyToWrite();

    // ************************************
    // One-2-Many Channel: Shared reading end
    // ************************************
    public boolean claimRead(PJProcess p) {
        return false;
    }

    public void unclaimRead() {
        // empty on purpose
    }

    // ************************************
    // Many-2-One Channel: Shared writing end
    // ************************************
    public boolean claimWrite(PJProcess p) {
        return false;
    }

    public void unclaimWrite() {
        // empty on purpose
    }
}
```

Listing 5.22: The `PJChannel` class.

This class contains seven fields: a protected *data* field in line 3 that holds the data to be transmitted over the channel, a *type* in line 4 that describes the various forms of communication between two processes, a *write*() method in line 6 used by a sending process to write data across the channel, a *read*() method in line 8 used by a receiving process to read data from a channel, a *isReadyToRead*() method in line 10 used to check if a sending process is present at the other end of the channel, a *isReadyToWrite*() method in line 12 used to check if a receiving process is present, a *claimRead*() method in line 17 used by a receiving process to hold on to the reading-end of the channel, an *unclaimRead*() method in line 21 used to release the reading-end of the channel, a *claimWrite*() method in line 28 used by a sending process to hold on to a writing-end of the channel, and, finally, an *unclaimWrite*() method in line 32 used to release this channel-end.

The implementation of the last four methods may change depending on the kind of channel used. However, the definition of the first four remains the same as the receiver needs to set the sender ready in order to be rescheduled and vice versa. Naturally, then, this will depend on whoever gets their respective read or write operation first. One thing worth mentioning is that these channels are drastically different from [95] – the previous experimental version of ProcessJ.

Channels are either used to connect a single writer process with a single reader (one-to-one), connect a single writer process with any number of readers (one-to-many), connect any number of writer processes with a single reader (many-to-one), connect any number of writer processes with any number of readers (many-to-many). These channels are therefore used to construct networks of communicating processes. It must be mentioned that a process should never be given a whole channel – only the end that it needs.

5.3.2.4 One-to-One Channel

`PJOne2OneChannel` represents a one-to-one channel communication based on the correct implementation written by Sr. Pedersen and Chalmers in [78]. The definition of this class is shown in Listing 5.23.

```
 1: public class PJOne2OneChannel<T> extends PJChannel<T> {
 2:
 3:     protected PJProcess writer;
 4:
 5:     protected PJProcess reader;
 6:
 7:     public PJOne2OneChannel() {
 8:         writer = null;
 9:         reader = null;
10:         type = PJChannelType.ONE2ONE;
11:     }
12: ...
```

Listing 5.23: The `PJOne2OneChannel` class.

This class has a *reader* in line 3 and a *writer* in line 5, which hold references to the writing and reading processes. The *write*() method in line 14 (Listing 5.24) and *read*() method in line 23 (Listing5.24) are the only methods provided to obtain the **end** of the channel by which writing and reading operations are performed.

```
13:     @Override
14:     synchronized public void write(PJProcess p, T data) {
15:         this.data = data;        // set data on channel
16:         writer = p;              // register the writer
17:         writer.setNotReady();    // set writer not ready
18:         if (reader != null)      // if a reader is there
19:             reader.setReady();   // set it ready to run
20:     }
21:
22:     @Override
23:     synchronized public T read(PJProcess p) {
24:         writer.setReady();       // set writer ready
25:         writer = null;           // clear writer field
26:         reader = null;           // clear reader field
27:         return data;             // return data
28:     }
29: ...
```

Listing 5.24: The `PJOne2OneChannel` class – `write` and `read`.

Recall, channels are unbuffered and their operations are fully synchronous: the reader process must wait for a matching writer to appear and vice versa. A writer process cannot continue past a write operation if the reader is not at the other end of the channel. For example, consider a channel communication in which the sending process arrives first. Since the receiving process is absent, the sending process can copy its data to the channel, register itself as the writer on the channel, set itself not-ready to run before returning control to the scheduler, and then wait for the receiving process to appear at the other end of the channel – here, waiting implies setting the `runLabel` used for resumption when the writer is rescheduled. When the reader finally appears, the writer is set ready to run and is resumed when it appears at the front of the run queue. Similarly for a reading process, it cannot move on if the writer process is not present. It should be mentioned that these operations (read and write) can only be performed if the *isReadyToRead*() method in line 31 (Listing 5.25) and *isReadyToWrite*() method in line 42 (Listing 5.25) are invoked and return true, indicating that both processes are present at each end of the channel, respectively.

```
30:     @Override
31:     synchronized public boolean isReadyToRead(PJProcess p) {
32:         if (writer != null)         // if a writer is present
33:             return true;            // return true
34:         else {                      // otherwise
35:             reader = p;             // register 'p' as the reader
36:             reader.setNotReady();   // set reader not ready
37:         }
38:         return false;
39:     }
40:
41:     @Override
42:     public boolean isReadyToWrite() {
43:         return true;                // always ready to write
44:     }
45: }
```

Listing 5.25: The `PJOne2OneChannel` class – `isReadyToRead` and `isReadyToWrite`.

Template Layout

A **write** operation on a write channel end in ProcessJ is shown in Listing 5.26, where `c` is a variable of a channel type.

```
c.write(4);
```

Listing 5.26: Example of a write statement.

The write statement is converted to the Java code in Listing 5.30 using the `ChanWriteStat` template.

```
    _pd$out1.write(this, 4);
    this.runLabel = 1;
    yield();
    label(1);
```

Listing 5.27: The `PJChannelOne2One` class generated from a template – write.

The template in Listing 5.28 takes in three attribute values: the name of the writing-end of a channel variable, the statement to be transmitted across the channel, and a label used for resumption when the writing process is rescheduled. In line 2, the attribute *chanName* is replaced with the name of channel variable, the writer process. This variable is used to call the *write*() method. The attribute *writeExpr*, following the `this` keyword within ( ), is replaced with the expression (or value) transmitted over the channel. When the statement in line 2 is rendered and is later executed by the compiler, the writer sends the data via the `write` method. In line 3, the attribute *resume0* is replaced with a label used to reschedule the writer process when the reader is present. This same label is used again in line 5 to mark the point from which the writer process should restart.

```
ChanWriteStat(chanName, writeExpr, resume0) ::= <<
<chanName>.write(this, <writeExpr>);
this.runLabel = <resume0>;
yield();
label(<resume0>);<\n>
>>
```

Listing 5.28: The template for a ProcessJ channel-write expression.

The expression to **read** from a channel in ProcessJ is `var = c.read()`, where `var` is the some variable and `c` is a variable of a channel type. This is is illustrated in Listing 5.29.

```
int x = in.read();
```

Listing 5.29: Example of a read expression.

The read expression is translated to the Java code in Listing 5.30 using the `ChannelReadExpr` template.

```
    if (!_pd$in1.isReadyToRead(this)) {
        this.runLabel = 1;
        yield();
    }

    label(1);
    _ld$x1 = _pd$in1.read(this);
    this.runLabel = 2;
    yield();

    label(2);
    io.println("x: " + _ld$x1);
    terminate();
```

Listing 5.30: The `PJChannelOne2One` class generated from a template – read.

The template in Listing 5.31 takes in the name of the reading end of a channel variable, the operand on the left hand side of the operator, a binary operator, and two labels used for resumption when the reader process is rescheduled to run again: the first is used when the writer is absent, and the second is used after the reader reads data.

```
 1: ChannelReadExpr(chanName, lhs, op, resume0, resume1) ::= <<
 2: if (!<chanName>.isReadyToRead(this)) {
 3:     this.runLabel = <resume0>;
 4:     yield();
 5: }
 6:
 7: label(<resume0>);
 8: <if(lhs)><lhs> <op> <chanName>.read(this);
 9: <else><chanName>.read(this);
10: <endif>
11: this.runLabel = <resume1>;
12: yield();
13:
14: label(<resume1>);
15: >>
```

Listing 5.31: The template for a ProcessJ channel-read expression.

In line 2, the attribute *chanName* is replaced with the name of the channel variable, namely, the reader process. This variable is used to call the *isReadyToRead*() method when attempting a read. For example, when the writer is not present, and the reader attempts to read (by calling `isReadyToRead`) and receives false, the reader must yield and at the time of rescheduling it should try again. Yielding at this point means setting the `runLabel` in line 3 with the value rendered by the attribute *resume0* – a **label** used to jump back to the code where the yield left off. This rendered value is used again in line 7. In line 8, the attribute *lhs* is replaced with some variable that is used to receive the data returned by `read`. Further still, on the same line, the attribute *op* is replaced with some binary operator (typically, the assignment operator), and the attribute *chanName* is replaced with the name of the reader process which is used to call `read`. Naturally, a call to the *read*() method will return the data in the channel. Note, line 9 is only rendered if the value of the attribute *lhs* in line 8 was `null`. Finally, in line 11, the attribute *resume1* is replaced with the second label, which is used in line 14 as a 'courtesy' yield for fairness. This courtesy yield forces the reader process to go around the run queue at least once so other processes get run.

5.3.2.5 One-to-Many Channel

A one-to-many channel communication is defined by the class called PJOne2ManyChannel (Listing 5.32). An instance of this class is safe for use by only one writer process and many readers.

```java
public class PJOne2ManyChannel<T> extends PJOne2OneChannel<T> {

    protected PJProcess readclaim = null;

    protected Queue<PJProcess> readQueue = new LinkedList<>();

    @Override
    public synchronized boolean claimRead(PJProcess p) {
        if (readclaim == null || readclaim == p) {
            readclaim = p;     // claim the channel
            return true;       // wait for reader
        } else {
            p.setNotReady();   // someone has claimed it
            readQueue.add(p); // add reader to the queue
        }
        return false;
    }

    @Override
    public synchronized void unclaimRead() {
        if (readQueue.isEmpty()) {
            readclaim = null; // make the channel available
        } else {
            PJProcess p = readQueue.remove();
            readclaim = p;     // release channel
            p.setReady();      // set process ready to run
        }
    }
}
```

Listing 5.32: The PJChannelOne2One class.

A number of reading processes and only a writer can commit to this channel. However, at any given time, only one reader and the writer will actually use the channel. This implies that all reading processes must compete with each other in order to use the shared channel-end. Although a shared channel-end is used in the same way as a one-to-one channel, before a reader may use the reading-end of the channel, the reader must first **claim** exclusive access to it. This can be done using the *claimRead*() method in line 8. For example, when a reader manages to claim a shared channel-end, this process will have exclusive rights to it. However, two things can happen – either there is data and the process can read or it has to wait for the writer to appear. For the writer, once a claim has been successful, the process proceeds like it did before in a one-to-one channel communication; that is, the writer sets itself ready to run again because it was the reader who woke the writer up. Once the communication is established, a reader must **unclaim** the shared channel-end upon completing its read operation for other reading processes to use it. This is done using the method *unclaimRead*() in line 20.

Template Layout

Listing 5.33 show an example of a reading shared channel-end.

```
public void reader(int id,
                   shared chan<int>.read in) {
  while (true) {
    int v; v = in.read();
    println(id + ": " + v);
  }
}
...
```

Listing 5.33: Example of a shared reading-end of a channel.

The Java code in Listing 5.34 is generated by the `ChannelOne2Many` template.

```
    if (!_pd$in2.claimRead(this)) {
        this.runLabel = 1;
        yield();
    }
    label(1);

    if (!_pd$in2.isReadyToRead(this)) {
        this.runLabel = 2;
        yield();
    }

    label(2);
    _ld$v1 = _pd$in2.read(this);
    this.runLabel = 3;

    _pd$in2.unclaimRead();

    yield();
    label(3);
```

Listing 5.34: The `PJChannelOne2Many` class generated from a template – shared read.

The template in Listing 5.35 is similar to that of Listing 5.33 (`ChanWriteStat`), except for the if-statement in line 3 and the `unclaimRead` statement that gets rendered in line 20. The attribute *chanName* in line 3 is replaced with the name of a read process, which, once again, is used to call the method *claimRead*() to aquire the end of the channel. The attribute *resume0* in line 4 is replaced with a label representing the point were the yield left off. When the call to `claimRead` is executed, and if no other process has already claimed the channel, the rendered value of this attribute will be used for resumption if a writer is not present or when data is available in the channel. Finally, line 20 is used to unclaim the channel.

```
ChannelOne2Many(chanName, lhs, op, resume0, resume1,
resume2) ::= <<
if (!<chanName>.claimRead(this)) {
    this.runLabel = <resume0>;
    yield();
}
label(<resume0>);

if (!<chanName>.isReadyToRead(this)) {
    this.runLabel = <resume1>;
    yield();
}

label(<resume1>);
<if(lhs)><lhs> <op> <chanName>.read(this);
<else><chanName>.read(this);
<endif>
this.runLabel = <resume2>;

<chanName>.unclaimRead();

yield();
label(<resume2>);
>>
```

Listing 5.35: The template for a ProcessJ channel-read.

5.3.2.6 Many-to-One Channel

A many-to-one channel communication is defined by the class called `PJMany2OneChannel` (Listing 5.36). In contrast with the one-to-one channel, an instance of this class is safe for use by only one reader process and multiple writers.

```
 1: public class PJMany2OneChannel<T> extends PJOne2OneChannel<T> {
 2:
 3:     protected PJProcess writeclaim = null;
 4:
 5:     protected Queue<PJProcess> writeQueue = new LinkedList<>();
 6:
 7:     @Override
 8:     public synchronized boolean claimWrite(PJProcess p) {
 9:         if (writeclaim == null || writeclaim == p) {
10:             writeclaim = p;     // claim the channel
11:             return true;        // wait for writer
12:         } else {
13:             p.setNotReady();    // someone has claimed it
14:             writeQueue.add(p); // add writer to the queue
15:         }
16:         return false;
17:     }
18:
19:     @Override
20:     public synchronized void unclaimWrite() {
21:         if (writeQueue.isEmpty()) {
22:             writeclaim = null; // make the channel available
23:         } else {
24:             PJProcess p = writeQueue.remove();
25:             writeclaim = p;     // release channel
26:             p.setReady();       // set process ready to run
27:         }
28:     }
29: }
```

Listing 5.36: The `PJChannelOne2One` class.

A number of writing processes and only a reader can commit to this channel. However, the channel will only be used by one writer and the reader at any one time. Similar to how reading processes compete with each other for a shared channel-end in a one-to-many channel communication, writing processes also have to compete but for the shared writing-end of a channel. Furthermore, a writer can attempt a claim on the shared channel-end by calling the *claimWrite*() method in line 8. When the claim is successful, and once the communication between the writer and the reader is established, the writer must unclaim the claimed channel end by calling the *umclaimWrite*() in line 20 upon completing its write operation.

Template Layout

Listing 5.37 show an example of a writing shared channel-end.

```
public void writer(int id,
                   shared chan<int>.write out) {
  int v = 0;
  while (true) {
    println(id + ": " + v);
    out.write(v);
    v = v + 1;
  }
}
...
```

Listing 5.37: Example of a shared writing-end of a channel.

The Java code in Listing 5.38 is generated by the `ChannelMany2One` template.

```
if (!_pd$out2.claimWrite(this)) {
    this.runLabel = 1;
    yield();
}
label(1);

_pd$out2.write(this, _ld$v1);
this.runLabel = 2;

yield();
label(2);

_pd$out2.unclaimWrite();
```

Listing 5.38: The `PJChannelMany2One` class generated from a template – shared write.

The template in Listing 5.39, apart from lines 2 and 14, is similar to the one in Listing 5.28 (`ChanWriteStat`). The attribute *chanName* in line 2 is replaced with the name of a writer process, which is used to call the method *claimRead*(). A call to this method allows a writer to set itsef not-ready to run if a reader is absent or to be ready for writing data. Line 20 is used to unclaim the channel.

```
ChannelMany2One(chanName, writeExpr, resume0, resume1) ::= <<
if (!<chanName>.claimWrite(this)) {
    this.runLabel = <resume0>;
    yield();
}
label(<resume0>);

<chanName>.write(this, <writeExpr>);
this.runLabel = <resume1>;

yield();
label(<resume1>);

<chanName>.unclaimWrite();<\n>
>>
```

Listing 5.39: The template for a ProcessJ channel many-2-one.

5.3.2.7 Many-to-Many Channel

The class `PJMany2ManyChannel` (Listing 5.40) defines a many-to-many channel communication. This channel is safe for use by many writers and many readers. Like the one-to-many and many-to-one channels, the only methods provided to obtain and release the end of the channel are: *claim-Read*() and *unclaimRead*() methods, and *claimWrite*() and *unclaimWrite*() methods. Consequently, this channel is handled in a similar manner using the `ChannelOne2Many` and `ChannelMany2One` templates. It must be pointed out that we had to re-implement these methods (though their definitions remain the same) as Java does not support multiple inheritance.

```java
public class PJMany2ManyChannel<T> extends PJOne2OneChannel<T> {

    // ***********************************
    // Shared reading end
    // ***********************************
    protected PJProcess readclaim = null;

    protected Queue<PJProcess> readQueue = new LinkedList<>();

    @Override
    public synchronized boolean claimRead(PJProcess p) {
        if (readclaim == null  readclaim == p) {
            readclaim = p;
            return true;
        } else {
            p.setNotReady();
            readQueue.add(p);
        }
        return false;
    }
```

Listing 5.40: The `PJChannelMany2Many` class – shared read.

```java

    @Override
    public synchronized void unclaimRead() {
        if (readQueue.isEmpty()) {
            readclaim = null;
        } else {
            PJProcess p = readQueue.remove();
            readclaim = p;
            p.setReady();
        }
    }

```

Listing 5.41: The `PJChannelMany2Many` class – shared read cont.

```java
    // ***********************************
    // Shared writing end
    // ***********************************
    protected PJProcess writeclaim = null;

    protected Queue<PJProcess> writeQueue = new LinkedList<>();

    @Override
    public synchronized boolean claimWrite(PJProcess p) {
        if (writeclaim == null  writeclaim == p) {
            writeclaim = p;
            return true;
        } else {
            p.setNotReady();
            writeQueue.add(p);
        }
        return false;
    }
```

Listing 5.42: The `PJChannelMany2Many` class – shared write.

```java
    @Override
    public synchronized void unclaimWrite() {
        if (writeQueue.isEmpty()) {
            writeclaim = null;
        } else {
            PJProcess p = writeQueue.remove();
            writeclaim = p;
            p.setReady();
        }
    }
}
```

Listing 5.43: The `PJChannelMany2Many` class – shared write cont.

A declaration of shared channel-ends can be seen in Listing 5.44.

```
...
public void main(string args[]) {
  shared chan<int> c; // shared both ends
  par {
    reader(1, c.read);
    reader(2, c.read);
    reader(3, c.read);
    writer(c.write);
    writer(c.write);
  }
}
```

Listing 5.44: Example of sharing both ends of a channel.

5.3.2.8 Barrier

Unlike a channel, in which only two processes (a reader and a writer) can synchronize, a barrier can be used to synchronize any number of processes. A barrier synchronization in ProcessJ is represented by the Java class called `PJBarrier` (Listing 5.45).

```
1: public class PJBarrier {
2:     public List<PJProcess> synced = new ArrayList<PJProcess>();
3:     public int enrolled = 0;
4:
5:     public PJBarrier() {
6:         this.enrolled = 1;
7:     }
8: ...
```

Listing 5.45: The `PJBarrier` class.

This class contains a number of methods that concurrent processes can be use to enroll, synchronize, and resign. The class contains a counter in line 3 (Listing 5.45), called `enrolled`, which represents the number or processes enrolled on a barrier object. When the *sync*() method in line 19 (Listing 5.47) is invoked, the counter in the barrier object is incremented by 1. This is done so that when a process is added to the queue in line 2 (Listing 5.45), it can be set ready to run when every process enrolled on the barrier has called the *sync*() method. Once all processes have finally enrolled on the barrier, this queue is emptied.

```
 9:     public synchronized void enroll(int m) {
10:         this.enrolled = this.enrolled + m - 1;
11:     }
12:
13:     public synchronized void resign() {
14:         if (this.enrolled > 1) {
15:             this.enrolled = this.enrolled - 1;
16:         }
17:     }
18: ...
```

Listing 5.46: The `PJBarrier` class – `enroll` and `resign`.

A process may choose at any time to enroll or resign from any barrier. In order to gracefully terminate every process enrolled on the barrier object, the *finalize*() method of a terminated process is invoked. A call to the *finalize*() method results in an invocation to the *resign*() method in line 5.46. When this invocation is made, the counter is decremented by 1. There are two important observations that are worth making. First, the counter holds the number of processes enrolled on the barrier object, including the barrier itself. Since the counter decrements after a process resigns, the counter will never become zero. This is because the barrier is also included in the counter. Therefore, when the counter becomes 1, this implies that all other processes except the process originally declaring the barrier have resigned. Since this process has not resigned, the barrier can still be used in this instance. Second, the *enroll*() method in line 9 is only used when the number of processes that wish to enroll on a barrier is known ahead of time.

```
    public synchronized void sync(PJProcess process) {
        process.setNotReady();
        synced.add(process);
        if (synced.size() == enrolled) {
            for (PJProcess p : synced) {
                p.setReady();
            }
            synced.clear();
        }
    }
}
```

Listing 5.47: The `PJBarrier` class – sync.

Template Layout

In ProcessJ, a barrier synchronization looks like the one in Listing 5.48.

```
b.sync();
```

Listing 5.48: Example of barrier.

The Java code in Listing 5.49 is generated by the `BarrierDecl` template.

```
_pd$b1.sync(this);
this.runLabel = 1;
yield();
label(1);
```

Listing 5.49: The `PJBarrier` class generated from a template.

The template in Listing 5.50 takes two attribute values. The attribute *barrier* in line 2 is replaced with some reference variable (a barrier object), which is used to call the *sycn*() method when a process enrolls on the barrier. Recall, a counter (the number of processes enrolled) is kept in the allocated `PJBarrier` object. This counter decrements every time the *sync*() method is invoked – a call to `sync` sets the enrolled processes ready/not-ready to run. Finally, the attribute *resume0* is replaced with the point to return to when all processes have sync'd on the barrier.

```
SyncStat(barrier, resume0) ::= <<
<barrier>.sync(this);
this.runLabel = <resume0>;
yield();
label(<resume0>);
>>
```

Listing 5.50: The template for a ProcessJ barrier.

5.3.2.9 Par

A par-block in ProcessJ is implemented as a `PJPar` class (Listing 5.51). The `PJPar` constructor takes the number of processes and returns an object that represents the parallel composition of the statements (sub-processes) inside the body of the par. A run of a par-block object terminates when, and only if, all its sub-processes terminate.

```
public class PJPar {
    private PJProcess process;
    private int processCount;

    public PJPar(int processCount, PJProcess p) {
        this.processCount = processCount;
        this.process = p;
    }

    public void setProcessCount(int count) {
        this.processCount = count;
    }

    public synchronized void decrement() {
        processCount--;
        if (processCount == 0) {
            process.setReady();
        }
    }
}
```

Listing 5.51: The `PJPar` class.

This class contains a counter in line 3, called *processCount*, that keeps track of how many processes within the par-block object are still running. The counter is initialized when the par-block object is created. This is done using the constructor in line 5. The `process` field in line 2 holds a reference to the process enclosing the par. This variable is used to set the process ready to run when all of its sub-processes have terminated, thus, the process remains not-ready to run for as long as one of its sub-processes is running. Once a sub-process has terminated, its *finalized*() method is called to decrement the value stored in *processCount* via the *decrement*() method in line 14. When *processCount* reaches zero, the process in which the par-block appears is set ready to run again.

Template Layout

A par-block in ProcessJ is a block of curly braces ({ }) with the `par` keyword prefixed. The example in Listing 5.52 shows a par-block with two statements, namely the invocation of the *foo*() and *bar*() procedures. In general, any number of processes can be executed in parallel. When the program executes, the par-block will turn each statement into a process that will run concurrently with the other. Note that a par-block also introduces a new scope, and they can be nested.

```
...
par {
    foo();
    bar();
}
...
```

Listing 5.52: Example of a par-block.

The Java code in Listing 5.53 is generated by the `ParBlock` template.

```
    final PJPar _ld$par1 = new PJPar(2, this);

    new PJProcess() {
        @Override
        public synchronized void run() {
            parExp._method$foo();
            terminate();
        }

        @Override
        public void finalize() {
            _ld$par1.decrement();
        }
    }.schedule();

    new PJProcess() {
        @Override
        public synchronized void run() {
            parExp._method$bar();
            terminate();
        }

        @Override
        public void finalize() {
            _ld$par1.decrement();
        }
    }.schedule();

    if (_ld$par1.shouldYield()) {
        this.runLabel = 1;
        yield();
        label(1);
    }
```

Listing 5.53: The `PJPar` class generated from a template.

The template in Listing 5.54 takes in four attribute values in the following sequence: the name of the process declaring the par-block, the number of process created inside the par, the statements in the par, a jump label being the instruction after the par, and an optional barrier (or barriers) on which the par-block's processes are enrolled.

```
 1: ParBlock(name, count, process, body, jump, barrier) ::= <<
 2: final PJPar <name> = new PJPar(<count>, <process>);
 3: <if(barrier)><barrier:{b | <b>.enroll(<count>)};
 4:  separator=";\n">;<endif>
 5:
 6: <if(body)><body; separator="\n\n"><endif>
 7:
 8: if (<name>.shouldYield()) {
 9:     this.runLabel = <jump>;
10:     yield();
11:     label(<jump>);
12: }<\n>
13: >>
```

Listing 5.54: The template for a ProcessJ par-block.

The attribute *name* in line 2 is replaced with the process (a reference variable) in which the par-block appears. On the same line, the attributes *count* and *process* are replaced with the number of processes in the par-block and the `this` keyword (the process enclosing the par), respectively. When this line is rendered, an instance of a `PJPar` is created using the parameters passed to the constructor. If any barriers are declared, the attribute *barrier* in line 3 is replaced with one or more reference variables representing each barrier. These variables are used to call the *enroll*() method to enroll the sub-processes. In line 6, the attribute *body* is replaced with one or more derived `PJProcess` classes while extending their implementation of the *finalize*() method to decrement the par-block's process counter. Finally, in line 9, the attribute *jump* is replaced with the resume point after the yield – an address to jump to when this process is reschedule to run again.

5.3.2.10 Timer

A timer provides the current system time, which can be used to delay execution by calling *timeout*() and as a **guard** for setting timeouts in an alt(ernation). A timer is represented by the Java class called PJTimer (Listing 5.55). This class implements the Java Delayed interface to mark processes that are ready to be used after a certain delay.

```
1: public class PJTimer implements Delayed {
2:   private PJProcess process;
3:
4:   private long delay;
5:   private boolean killed = false;
6:   public boolean started = false;
7:   public boolean expired = false;
8: ...
```

Listing 5.55: The PJTimer class.

A timer object can be started, terminated, or killed through its *started* (line 6), *expired* (line 7), and *killed* (line 5) fields. The *start*() method in line 20 (Listing 5.56) sets the *delay* field in line 4 (Listing 5.55) with an absolute timeout value for a timer object, and the *started* field in line 6 (see Listing 5.56) to true to indicate the specified wait time. It then inserts the timer object into the timer queue of the scheduler. The *expire*() method in line 26 (Listing 5.56) is used by the scheduler to indicate that a process's delay has expired, namely its timer object. When a timer object expires, the process with the expired timer object is set ready-to run, which the scheduler then executes when it gets to the head of the queue.

```
20:   public void start() throws InterruptedException {
21:       this.delay = System.currentTimeMillis() + timeout;
22:       PJProcess.scheduler.insertTimer(this);
23:       started = true;
24:   }
25:
26:   public void expire() {
27:       expired = true;
28:   }
```

Listing 5.56: The PJTimer class – start and expire methods.

A process with a timeout call sets itself not-ready to run after its timer object has started. Expiration occurs when the *getProcess*() method in line 38 (Listing 5.57) returns a reference to a process with an expired timer object. When this happens, the timer queue uses the returned value as an indication of having a process whose delay expired furthest in the past. In this instance, the process is set ready to run so that it can be scheduled in the future. Even though unexpired timer objects are not removed from the delay queue unless their *delay* has expired, they can be killed off when necessary (currently this is only done by an alt). Consider an alt with two guards: the first is a guard with a read expression and the second is a timeout statement. If the first guard is marked as ‘ready’, the alt will be set ready to run again. Therefore, the process declaring the alt will be scheduled to run, and after some time it will yield. In the mean time, the timer object linked to this process expires. This will result in setting the process ready to run again. If this process is stuck in a read operation, it may get woken by a timer that timed out. To avoid this from happening, a timer object must be killed off as it may set the process ready to run. In a situation such as this, the timer object is killed by the *kill*() method in line 34 (Listing 5.57). Finally, timers can be read much like a channel. In order to read the current time, the method *read*() in line 30 can be used.

```
30:   public static long read() {
31:       return System.currentTimeMillis();
32:   }
33:
34:   public void kill() {
35:       killed = true;
36:   }
37:
38:   public PJProcess getProcess() {
39:       if (killed) {
40:           return null;
41:       } else {
42:           return process;
43:       }
44:   }
```

Listing 5.57: The `PJTimer` class – read and kill methods.

Template Layout

A timer is declared in a manner similar to a channel and a variable. For example, consider the piece of ProcessJ code in Listing 5.58.

```
1: Timer t;
2: t.timeout(100);
```

Listing 5.58: Example of a timer declaration and timeout statement.

The type of the variable declaration in line 1 (Listing 5.58) is determined in the generated Java code as an instance of the `PJTimer` class. For the timeout call in line 2, the generated Java code in Listing 5.59 is translated by the `TimeOutStat` template.

```
    _ld$t1 = new PJTimer(this, 100);
    try {
        _ld$t1.start();
        setNotReady();
        this.runLabel = 1;
        yield();
    } catch (InterruptedException e) {
        System.out.println("An Interrupted exception " +
            "occurred for a timer!");
    }
    label(1);
    ...
```

Listing 5.59: The `PJTimer` object generated from a template – timeout read.

The template in Listing 5.60 takes three parameters in the following sequence: the name of the timer object, a delay used to block for a specified time period (in milliseconds), and a resume point – the instruction after a yield statement. In line 2, the attribute *name* is replaced with the name of the timer object. In addition to replacing this attribute, a call to the `PJTimer` constructor follows the `new` operator on the same line. The attribute *delay* is replaced with the absolute timeout value, which is passed to the constructor as a parameter to start the time of the timer object. When the statement in line 2 is rendered, the compiler creates a timer object with the allocated delay time. The attribute *name* in line 4 is replaced with the name of the timer object allocated in line 2 and is used to call the *start*() method. A call to this method inserts the timer object into the timer queue in a separate thread. Finally, the *resume0* attribute in lines 6 is replaced with the label representing the resume point where the processes with the timer object should restart from when its delay has expired.

```
TimeoutStat(name, delay, resume0) ::= <<
<name> = new PJTimer(this, <delay>);
try {
    <name>.start();
    setNotReady();
    this.runLabel = <resume0>;
    yield();
} catch (InterruptedException e) {
    System.out.println("An Interrupted exception occurred " +
                       "for a timer!");
}
label(<resume0>);
>>
```

Listing 5.60: The template for a ProcessJ timeout expression.

To return the current system time in milliseconds as shown in Listing 5.61, the *read*() method is used.

```
    ... = PJTimer.read();
```

Listing 5.61: The `PJTimer` object generated from a template – timeout read.

The `TimerReadExpr` template in Listing 5.62 generates code that is used to read input from a timer object. Since the *read*() method is static, an object is no required to make the call. In line 1, the attribute *name* is replaced with the name of some local variable (where . . . is a place holder for a variable), and the call to the *read*() method is used to return a `long` value representing the time measured in milliseconds.

```
1: TimerReadExpr(name) ::= "<name> = PJTimer.read();"
```

Listing 5.62: The template for a ProcessJ timer-read expression.

5.3.2.11 Record

A record is one way of structuring data in ProcessJ. It is represented by a class that implements the Java interface called `PJRecord` (Listing 5.63), which is an empty interface that serves as the 'parent' record for all other records. The idea behind using this interface is to facilitate multiple inheritance in ProcessJ. In contrast to Java, where a class can inherit methods, properties, and other characteristics from another class, fields are subsequently added to a derived record in ProcessJ. More specifically, there is no concept of partially inheriting fields, thus all inherited fields must become part of the body of a derived record. This is necessary as it is the only way we can approach multiple inheritance using the Java compiler.

```java
public interface PJRecord {
    /* Empty on purpose */
}
```

Listing 5.63: The `PJRecord` class.

Template Layout

The example in Listing 5.65, defines a new record called `Car`. This base record has two fields in lines 2 and 3, namely, `model` and `make`. These fields are variables that are bundled up and stored as part of the record in the generated Java class.

```
1: public record Car {
2:     string model;
3:     string make;
4: }
```

Listing 5.64: Example of a record definition.

The ProcessJ compiler generates the Java class in Listing 5.65 using the `RecordClass` template.

```
public static class Car implements PJRecord {
    public String model;
    public String make;

    public Car(String model, String make) {
        this.model = model;
        this.make = make;
    }
}
```

Listing 5.65: The `PJRecord` class generated from a template – Java class.

In Listing 5.66, line 1 takes in the following attributes in sequence: a list of modifiers, the name of the record being created, and a list of types and variable names. To enable multiple inheritance, the generated class must implement the `PJRecord` interface. In line 2, the list of modifiers is rendered with the name of the record type, including the `implements` keyboard followed by `PJRecord`. Naturally, appending the `implements` keyword to a class declaration forces the class to adhere the contract built by the compiler at runtime. The `public` modifier is rendered, followed by the list of types and variable names in line 4. To keep the state of all rendered fields, a constructor is generated in line 7. This constructor takes the same kind of variables rendered in line 4, which are used to initialize the fields. In line 8, to avoid shadowing a field by a parameter in the body of the constructor, each field is rendered with the the `this` keyboard prefixed.

```
 1: RecordClass(modifiers, name, types, vars) ::= <<
 2: <if(modifiers)><modifiers; separator=" "> <else>protected
 3:    <endif>static class <name> implements PJRecord {
 4:   <if(vars)><types,vars:{t,v | public <t> <v>};
 5:    separator=";\n">;<\n><endif>
 6:
 7:   public <name>(<types,vars:{t,v | <t> <v>}; separator=", ">) {
 8:       <if(vars)><vars:{v | this.<v> = <v>};
 9:        separator=";\n">;<endif>
10:   }
11: }
```

Listing 5.66: The template for a ProcessJ record.

When a record is defined, it creates a user-defined Java class. However, no memory is allocated. To dynamically allocate memory for a given record type, and work with it, the temple `RecordLiteral` in is used to generated the Java code in Listing 5.68. Using the derived type `record Car` defined in Listing 5.65, we can allocate memory for a record variable with the `new` operator in ProcessJ. This can be seen in Listing 5.67.

```
Car car = new Car { mode = "Lancer", make = "Mitsubishi" };
```

Listing 5.67: Example of a record object created with the `new` operator.

```
... = new Car("Lancer", "Mitsubishi");
```

Listing 5.68: The `PJRecord` class generated from a template – new operator.

The template in Listing 5.69 takes in two attribute values. The attribute *type* is replaced with the name for the record type, and the attribute *name* is replaced with a reference variable, that is, a variable that holds a reference to an object of type *type*. In Listing 5.68, the ... is a place holder for a variable that will hold a reference to an object of type `Car`. The attribute *type* following the `new` operator is replaced with the name for the record type in line 2. Furthermore, the attribute *vals* within ( ) is replaced with a set of valid actual parameters. These parameters are eventually used to initialize the fields of the record. For example, when line 2 is rendered and then executed by the compiler, the `new` operator will make a call to the constructor of the class `Car` with the specified parameters, `"Lancer"` and `"Mitsubishi"`. It will then return a reference to the object it created, that is, an object of type `Car`.

```
1: RecordLiteral(type, vals) ::= <<
2: new <type>(<vals; separator=", ">)
3: >>
```

Listing 5.69: The template for a ProcessJ record literal.

When a record object is created, its member(s) can be accessed directly. The syntax for that is to inserts a dot (.) between the object name and the member name as shown in Listing 5.70.

```
1: ...
2: println("model: " + car.model);
```

Listing 5.70: Example of access to a record's member.

The generate Java code in Listing 5.71 is created using the `RecordAccess` template.

```
42: ...
43: println("model: " + _ld$car1.model);
```

Listing 5.71: The `PJRecord` class generated from a template – member access.

The template in Listing 5.72 takes in two attribute values: the name of the record whose member we want to access, and the member name. The attribute *name* in line 1 is replaced with an object name (a reference variable), and the attribute *member* is, as the name would suggest, replaced with the name of a field that belongs to the object. Using the previous code (Listing 5.70), *name* is replaced with `car` and *member* is replaced with `model` in line 1. When this line is rendered, the non-static field of `car` its access through its modified name `_ld$car1`. Listing 5.71 shows the section of code generated by the template.

```
1: RecordAccess(name, member) ::= "<if(name)><name>.<member><endif>"
```

Listing 5.72: The template for a ProcessJ record member access.

5.3.2.12 Protocols

A protocol is another way of structuring data in ProcessJ. Unlike a record, a protocol is a class that encloses (encapsulates) its tags, where each tag extends the Java class called `PJProtocolCase` (Listing 5.73). The `PJProtocolCase` class makes an existing protocol instance/value adopt a tag whose data may vary in type from one instance to another. This allows protocols to provide support for values that can be one of a number of *tag-name* cases, possibly each with different values and types.

```java
public class PJProtocolCase {

    public String tag = null;
}
```

Listing 5.73: The `PJProtocolCase` class.

While single and multiple inheritance is allowed with protocols (Listing 5.74), the inheritance of a protocol is different from that of Java. For example, a protocol that extends another protocol can behave like the parent of the extended protocol. Further still, a protocol never inherits members of another protocol. It instead describes a value that can be one of several types. This means that the value of a protocol can only be determined by the tag it is initialized with. In other words, at any given time, a protocol instance/value can contain no more than one tag no matter how many tag members it has or inherits. This is further illustrated in Listing 5.75.

```java
public protocol P {
    request : { int number; double amount; }
    reply : { boolean status; }
}
public protocol P1 extends P {
    deny : { int code; }
}
```

Listing 5.74: Example used to generate the `PJProtocol` class from a template.

```java
...
P1 p = new P1 { deny: code = 4 };

switch (p) {
    case request: println("request"); break;
    case reply: println("reply"); break;
    case deny: println("deny"); break;
}
```

Listing 5.75: Example of access to a protocol's tag.

In listing 5.75, although we can match individual protocol values with a switch statement, only tags from the derived protocol and its parent protocols can be used. The compiler will throw an error if a tag that is not defined in either protocol is used in a case statement.

Template Layout

Protocols have a similar definition syntax to `unions` in C. An example of a protocol definition was introduced in Listing 5.74. The `ProtocolClass` template is used to generate the appropriate Java class for protocol `P` in Listing 5.76.

```
public static class P {
    ...
}
```

Listing 5.76: The `PJRecord` class generated from a template.

The template in Listing 5.77 takes in three attribute values: a list of modifiers, the name of the protocol being created, and a list of tag values. In line 2, the list of modifiers is rendered with the name of the protocol. In this instance, the value being rendered is the name for the protocol type `P`.

```
1: ProtocolClass(modifiers, name, body) ::= <<
2: <if(modifiers)><modifiers; separator=" "> <else>protected
4:  <endif>static class <name> {
5:     <body; separator="\n\n">
6: }
```

Listing 5.77: The template for a ProcessJ protocol.

Each member of a protocol can be of a different type, thus different classes need to be created for each of them as shown in Listing 5.79. The `ProtocolCase` template in Listing 5.78 is used for that purpose. This template takes in four attribute values: a list of modifiers, a tag name, and a list of types and variable names associated with the tag being created. The list of modifiers is rendered along with the name for the tag type, including the `extends` keyword followed by `ProtocolCase` in line 2. Extending this class enable us to use tags as place holders for their values when they are dynamically created. In line 5, the list of tag values is turned into fields. A constructor for the extended class is defined in line 8 while the attribute *name* is replaced with the name of the class the constructor belongs to, namely the name of the tag member. In addition, the attribute *vars* is replaced with a set of valid actual parameters. These parameters are used to initialize the tag member fields. Finally, the `this` keyword is prefixed to the rendered values of the attribute *vars* in line 10.

```
 1: ProtocolCase(modifier, name, types, vars) ::= <<
 2: <if(modifier)><modifier> <else>protected <endif>static class
 3: <name> extends PJProtocolCase {
 4:
 5:   <if(vars)><types,vars:{t,v | public <t> <v>};
 6:    separator=";\n">;<\n><endif>
 7:
 8:   public <name>(<types,vars:{t,v | <t> <v>}; separator=", ">) {
 9:      <! Initialize member fields !>
10:      <if(vars)><vars:{v | this.<v> = <v>};
11:       separator=";\n">;<endif>
12:      this.tag = "<name>";
13:   }
14: }
```

Listing 5.78: The template for a ProcessJ protocol type.

```java
public static class P {
  protected static class request extends PJProtocolCase {
      public int number;
      public double amount;

      public request(int number, double amount) {
          this.number = number;
          this.amount = amount;
          this.tag = "request";
      }
  }

  protected static class reply extends PJProtocolCase {
      public boolean status;

      public reply(boolean status) {
          this.status = status;
          this.tag = "reply";
      }
  }
}

public static class P1 {
    protected static class deny extends PJProtocolCase {
        public int code;

        public deny(int code) {
            this.code = code;
            this.tag = "deny";
        }
    }
}
```

Listing 5.79: The `PJRecord` class generated from a template – tag name.

When a protocol object is created, each member is allocated as if it is the only member of the protocol. Furthermore, the lifetime of a protocol member (the values associated with a tag-name) starts when it becomes active, that is, when it is created using a protocol literal. An example can be seen in Listing 5.80.

```
P1 p = new P1 { deny: code = 4 };
```

Listing 5.80: The `PJRecord` class generated from a template – new operator.

The generated Java code in Listing 5.81 is created using the `ProtocolLiteral` template.

```
... = new P1.deny(4);
```

Listing 5.81: The `PJRecord` class generated from a template – creating a tag.

The template in Listing 5.82 takes in three attribute values: the name for a protocol type, the tag, and a list of tag values. In line 2, the attribute *type* is replaced with the name for the protocol type that the tag member belongs to. Furthermore, the attribute *tag* is replaced with the name for the tag member being accessed. It should be mentioned that the name for the tag member is the name of the class surrounded by the class type of the protocol. Finally, the attribute *vals* is replaced with the list of values passed to the constructor as declared by the code in Listing 5.79.

```
1: ProtocolLiteral(type, tag, vals) ::= <<
2: new <type>.<tag>(<if(vals)><vals; separator=", "><endif>)
3: >>
```

Listing 5.82: The template for a ProcessJ protocol literal.

A switch statement is used to compare a protocol's tag in its expression with the expression associated with each case label. As shown in Listing 5.83, we compare the switch expression, namely p, with several case values.

```
...
    switch (p) {
    case request:
        println("Done!");
        break;
    case reply:
        println("status = " + p.status);
        break;
    }
```

Listing 5.83: Example of access to a protocol's tag value.

The Java code in Listing 5.84 is generated by the `ProtocolAccess` template.

```
switch(_ld$p1.tag) {
case "request":
    io.println("Done!");
    break;
case "reply":
    io.println("status = " + (((P.reply) _ld$p1).status));
    break;
}
```

Listing 5.84: The `PJProtocolCase` class generated from a template – tag value.

The template in Listing 5.85 takes in four attribute values. The attribute *name* is replaced with the name for the protocol type being used as the expression in the switch statement. The attribute *tag* is replaced with the `tag` field (Listing 5.73), which is the case identifier that we compare to the several case values in the switch. The attribute *var* is replaced with some reference variable, a protocol case object. Finally, the attribute *member* is replaced with the case field whose value we want to retrieve.

```
ProtocolAccess(protocName, tag, var, member) ::= <<
<if(protocName)>(((<protocName>.<tag>) <var>).<member>)<endif>
>>
```

Listing 5.85: The template for a ProcessJ protocol member access.

Chapter 6

Implementation

The primary objective of this thesis project is to develop a compiler that generates reliable Java code using CSP primitives translated into Java classes. Two example/test programs are used to determine the correctness of the generated code and runtime components, and to get an idea of the capabilities of the system I have implemented.

6.1 Full Adder

This test program implements a full eight-bit adder as a network of communicating processes, each of which functions as a small Boolean logic gate. The layout of this network is very naïve and simple. It is composed of several gates such as **AND**, **OR**, **NOT**, **NAND**, and **XOR**. These gates are grouped in order to form more complex components such as adders, multiplexers, etc. Such components are used to transform input signal levels into output signal levels that are transmitted to other network components on a continuous basis. The circuit diagram in Figure 6.1 can be straightforward implemented except for the wiring. While wiring these components electrically is trivial (mainly when diagram tools are used), they need to be modeled explicitly in ProcessJ as point-to-point, synchronized, and unbuffered channel communication each feeding into a single process. The ProcessJ implementation of a full one-bit adder diagram is shown in Listing 6.1 (see Appendix H for the complete implementation). A total of 632 processes composed the complete eight-bit adder. The generated Java code for a one-bit adder can be seen in Listing 6.2.

As mentioned, the *run*() method is responsible for executing the content of a ProcessJ process. The par-block in Listing 6.1 turns each procedure into a process that runs concurrently with the other procedures in the par-block. When the `run` method of the ProcessJ *oneBitAdder*() procedure is called by the scheduler (Listing 6.3), the entire par-block does not terminate until every process of the block has terminated

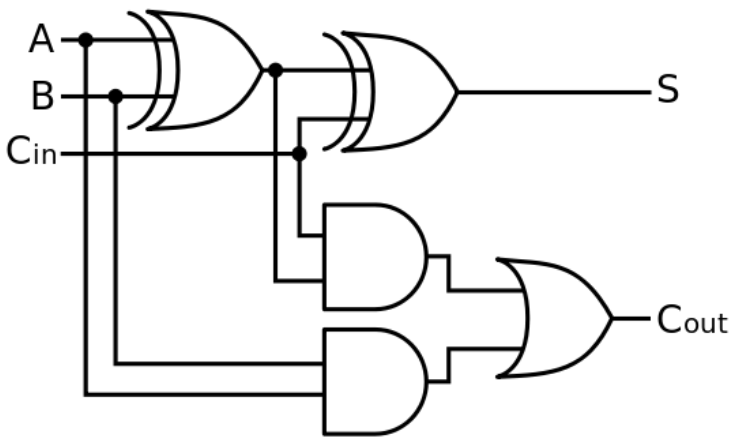

Figure 6.1: A full one-bit adder diagram.

```
public void oneBitAdder(chan<boolean>.read in1,
                        chan<boolean>.read in2,
                        chan<boolean>.read in3,
                        chan<boolean>.write result,
                        chan<boolean>.write carry) {
    chan<boolean> a, b, c, d, e, f, g, h, i, j, k;
        par{
            muxGate(in1, a.read, b.write);
            muxGate(in2, c.read, d.write);
            xorGate(a.read, c.read, e.write);
            muxGate(e.read, f.read, g.write);
            muxGate(in3, h.read, i.write);
            xorGate(f.read, h.read, result);
            andGate(g.read, i.read, j.write);
            andGate(b.read, d.read, k.write);
            orGate(j.read, k.read, carry);
        }
}
```

Listing 6.1: A one-bit adder implementation in ProcessJ.

```java
public static class _proc$oneBitAdder extends PJProcess {
    protected PJChannel<Boolean> _pd$in11;
    protected PJChannel<Boolean> _pd$in22;
    protected PJChannel<Boolean> _pd$in33;
    protected PJChannel<Boolean> _pd$result4;
    protected PJChannel<Boolean> _pd$carry5;

    protected PJChannel<Boolean> _ld$a1;
    protected PJChannel<Boolean> _ld$b2;
    protected PJChannel<Boolean> _ld$c3;
    protected PJChannel<Boolean> _ld$d4;
    protected PJChannel<Boolean> _ld$e5;
    protected PJChannel<Boolean> _ld$f6;
    protected PJChannel<Boolean> _ld$g7;
    protected PJChannel<Boolean> _ld$h8;
    protected PJChannel<Boolean> _ld$i9;
    protected PJChannel<Boolean> _ld$j10;
    protected PJChannel<Boolean> _ld$k11;

    public _proc$oneBitAdder(PJChannel<Boolean> _pd$in11,
                             PJChannel<Boolean> _pd$in22,
                             PJChannel<Boolean> _pd$in33,
                             PJChannel<Boolean> _pd$result4,
                             PJChannel<Boolean> _pd$carry5) {
        this._pd$in11 = _pd$in11;
        this._pd$in22 = _pd$in22;
        this._pd$in33 = _pd$in33;
        this._pd$result4 = _pd$result4;
        this._pd$carry5 = _pd$carry5;
    }

```

Listing 6.2: The generated Java code for a one-bit adder – part 1.

```
    @Override
    public synchronized void run() {
        switch (this.runLabel) {
            case 0: break;
            case 1: resume(1); break;
            default: break;
        }

        _ld$a1 = new PJOne2OneChannel<Boolean>();
        _ld$b2 = new PJOne2OneChannel<Boolean>();
        _ld$c3 = new PJOne2OneChannel<Boolean>();
        _ld$d4 = new PJOne2OneChannel<Boolean>();
        _ld$e5 = new PJOne2OneChannel<Boolean>();
        _ld$f6 = new PJOne2OneChannel<Boolean>();
        _ld$g7 = new PJOne2OneChannel<Boolean>();
        _ld$h8 = new PJOne2OneChannel<Boolean>();
        _ld$i9 = new PJOne2OneChannel<Boolean>();
        _ld$j10 = new PJOne2OneChannel<Boolean>();
        _ld$k11 = new PJOne2OneChannel<Boolean>();
        final PJPar _ld$par1 = new PJPar(9, this);

        (new fullAdder._proc$muxGate(_pd$in11, _ld$a1,
                                     _ld$b2) {
            @Override
            public void finalize() {
                _ld$par1.decrement();
            }
        }).schedule();

        (new fullAdder._proc$muxGate(_pd$in22, _ld$c3,
                                     _ld$d4) {
            @Override
            public void finalize() {
                _ld$par1.decrement();
            }
        }).schedule();

```

Listing 6.3: The generated Java code for a one-bit adder – part 2.

```java
        (new fullAdder._proc$xorGate(_ld$a1, _ld$c3,
                                     _ld$e5) {
            @Override
            public void finalize() {
                _ld$par1.decrement();
            }
        }).schedule();

        (new fullAdder._proc$muxGate(_ld$e5, _ld$f6,
                                     _ld$g7) {
            @Override
            public void finalize() {
                _ld$par1.decrement();
            }
        }).schedule();

        (new fullAdder._proc$muxGate(_pd$in33, _ld$h8,
                                     _ld$i9) {
            @Override
            public void finalize() {
                _ld$par1.decrement();
            }
        }).schedule();

        (new fullAdder._proc$xorGate(_ld$f6, _ld$h8,
                                     _pd$result4) {
            @Override
            public void finalize() {
                _ld$par1.decrement();
            }
        }).schedule();

```

Listing 6.4: The generated Java code for a one-bit adder – part 3.

```
        (new fullAdder._proc$andGate(_ld$g7, _ld$i9,
                                     _ld$j10) {
            @Override
            public void finalize() {
                _ld$par1.decrement();
            }
        }).schedule();

        (new fullAdder._proc$andGate(_ld$b2, _ld$d4,
                                     _ld$k11) {
            @Override
            public void finalize() {
                _ld$par1.decrement();
            }
        }).schedule();

        (new fullAdder._proc$orGate(_ld$j10, _ld$k11,
                                    _pd$carry5) {
            @Override
            public void finalize() {
                _ld$par1.decrement();
            }
        }).schedule();

        if (_ld$par1.shouldYield()) {
            this.runLabel = 1;
            yield();
            label(1);
        }

        terminate();
    }
}
```

Listing 6.5: The generated Java code for a one-bit adder – part 4.

Figure 6.2 shows the output generated by the eight-bit adder program. The program takes a0 through a7, that is, 8-bits of input that constitute parts of a, where a0 is the lowest order bit and a7 is the highest order bit. Then, b0 through b7 indicates the bits of the second input, where b0 is again the lowest order bit of b and b7 is the highest.

```
> pj fullAdder
  false true  false false false true  false false (InCarry:false)
+ true  false true  false true  false true  true
  --------------------------------------------------
  true  true  true  false true  true  true  true
Carry was: false
```

Figure 6.2: Full eight-bit adder output.

6.2 CommsTime

To further demonstrate the abilities of the runtime elements, another test program (one that is commonly used for CSP frameworks) is commstime [72, 90]. This consists of building a simple network with prefix, successor, delta, and consumer process that communicate over channels. Figure 6.3 illustrates the process network. The process *Prefix* starts the communication by sending the initial integer value to *Delta*. This process then sends the value to *Succ* and to *Consumer*. *Succ* increments the value by one and then sends it to *Prefix*, which in turn sends the value to *delta*. The network operates as long as *Delta* is able to send the value over its first channel to *Succ*. Listing 6.6 shows the ProcessJ version of the CommsTime network. The output of this program, after 1 million loops, is shown in Figure 6.4. Appendix I illustrates, in detail, how each of its processes is implemented.

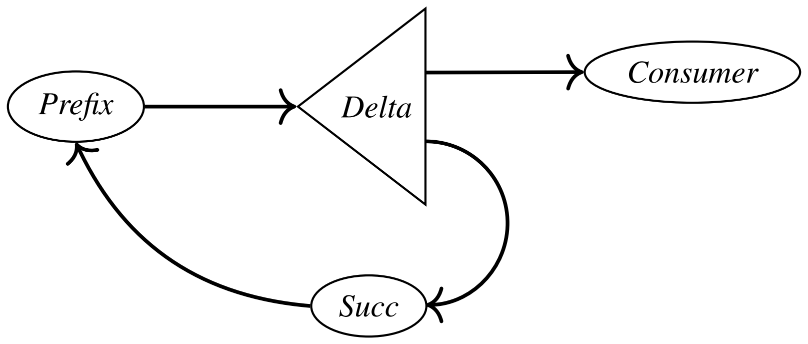

Figure 6.3: The CommsTime network.

```
public void main(string args[]) {
  chan<long> a,b,c,d;
  par {
    delta(d.read, a.write, b.write);
    succ(b.read, c.write);
    prefix(0, c.read, d.write);
    consumer(a.read);
  }
}
```

Listing 6.6: CommsTime implementation in ProcessJ.

```
> pj commstime
...
999994
999995
999996
999997
999998
999999
1000000
```

Figure 6.4: CommsTime output.

Chapter 7

Conclusion

In this thesis I presented ProcessJ, the reimplementation of the JVMCSP code generator and through several examples demonstrated its practical application. Although I focused on the rewriting of the code generation and some parts of the runtime system, I also considered rewriting the original command line parsing for the compiler.

The command line parsing library presented in this thesis uses a declarative approach to specifying command line options for the ProcessJ compiler. By looking at the features and problems of other libraries, I designed the command line library to have the following key features: allow for the configuration of a command line with little effort; enable parsing of command line arguments into distinct constructs while being typed safe – currently, the constructs supported include commands, options, and arguments; and capable of invoking functionality that is configured to run based on the command line value.

In addition, I have presented the implementation of a code generation scheme and the runtime system for ProcessJ. I described how the ProcessJ compiler generates code from the information it obtains using the visitor pattern and the StringTempalte library, and demonstrated the rewriting steps required to produce a compilable Java source file. For the runtime system, I utilized the approach proposed by Sr. Pedersen and Chalmers in [78] of a one-to-one channel communication for a multi-core cooperatively scheduled runtime system. This approach differs dramatically from the one previously implemented in [95] as it was translated from ProcessJ to Java and subsequently into CSP before being verified with the model checker FDR. Additionally, through this improved approach, I was able to simplify significantly the amount of code involved for the remaining shared channel ends; that is, one-to-many, many-to-one, and many-to-many.

Furthermore, I have explained the reconstruction of CSP primitives, such as channels, process, timers, barriers, and other process-oriented primitives, into object-oriented Java code. I also explained how the ProcessJ compiler enables process mobility using only code generation and a Java bytecode rewriting technique. I described the steps involved in retaining the local state of a process in between resumptions of code execution, and how the execution following the previous suspension point continuous with the same local state. The saving and restoration of local state work with a simple cooperative scheduler responsible for executing processes in the JVM as previously discussed.

Although ProcessJ and its implementation still do not provide full support for CSP-like features that other more mature languages contain, I am optimistic about its future progress. I will conclude this thesis with a number of future work to be addressed in the following chapter.

Chapter 8

Future Work

Much work remains to be done to make ProcessJ a successful programming language. A number of significant improvements can be made to the runtime system and code generator. I will mention some of the most important improvements in this section.

8.1 Multi-core Scheduler

A multi-core scheduler for the ProcessJ runtime system will be developed in the near future. There are extremely efficient runtime systems that support process-oriented programming for multi-core architectures [85, 86], thus, we intend to use them to make full use of all available computing resources in today's multi-core systems. We are fully aware that writing a multi-core scheduler is not a simple task. Although our runtime elements have already been designed for a single-core scheduler with proper synchronized access, we believe such elements will not required many changes in order to work with a multi-core scheduler. Alternatively, we intend to adapt some of the techniques outline in [85] for a naïve version of a multi-core scheduler if needed.

8.2 Run Queue

Currently, ProcessJ has a single run queue which holds both ready to run and not-ready to run processes. To take advantage of a single-threaded scheduler, we could split the process queue into two queues: the first for ready processes and the second for the not ready processes. When a not ready process becomes ready, it is move to the ready queue to be scheduled to run. Similarly for ready processes, when they become not ready, they are moved to the not ready queue. This should decrease the overhead of cycling through not ready processes as there is really no reason for that if they cannot be run. The only drawback to this is that a lot of bookkeeping may be needed for the runtime elements depending on the application. It is not known whether this would significantly improve performance in a single-core scheduler. However, we should still explore this alternative.

8.3 Blocking I/O Calls

Blocking I/O can be a problem with a single-threaded scheduler. This is because when an external I/O call blocks, the thread running the scheduler will block. One way to solve this problem is using Java threads. We can allocate a new thread and place the blocking I/O call there so that it can be executed separately by the JVM scheduler. For this to happen, we would need to have the `PJProcess` class implement the runnable interface. This will allow the blocking code to be placed in a new thread after its *run*() method is invoked.

8.4 Mobile processes

To implement mobile processes with polymorphic resumption interfaces [81] the code generator needs the following changes:

- A number of additional parameter for setter methods must be added to the class generated for the mobile process to support the polymorphic nature of the mobile processes.
- When the mobile process is invoked (it resumes) it needs to be wrapped in a par-block
- Support accessors must be implemented to determine the available procedure interface

8.5 Alternation

Alts were initially implemented as ‘busy-wait’. They cycled through the run queue but remained ready to run without being resumed by the scheduler. Only when one of theirs guards became ‘ready’ were the processes declaring the alts ready to run again. We would like to improve them by yielding with a not ready to run status and have one of their guards wake up the processes in which the alts appear – presently, a 2-way alternation (equivalent to an external choice in CSP) is currently being formally verified using the model checker FDR tool. We expect to report on this very soon. Finally, we would like to allow barriers in alt blocks, as well as nested alts.

8.6 Libraries

We plan to develop more sophisticated libraries for data structures and graphics, specially for graphics since the majority of GUIs available in Java are not thread safe, a classic example being Java's Swing [27]. We would like to make graphical interfaces thread safe simply by conforming to the parallel rules of CSP.

8.7 Command line library

The command line interpreter uses a very sophisticated way of correcting and suggesting candidate options using Levenshtein edit distance among other techniques. These techniques may have many potential uses including using some of the implementations to help special cases of the string alignment problems and accelerating OCR post processing training by more easily being able to perform ground truth verification [47, 46]. The alignment algorithm used there is elegant and was very useful when developing a way to match suggestions to mistyped arguments in the command line interpreter. Further work and standardization on the development of command line interpreters like the one used for ProcessJ would go a great way to improve current command line interpreters on popular programming languages.

8.8 Other Backends and Runtimes

Having different back-ends will allow developers to easily modify their programs. They will have the advantage of using existing libraries and frameworks from different languages while retaining backward compatibility. This will encourage code reuse and code sharing when writing process-oriented programs in ProcessJ. As a result, we are planning on developing new back-ends targeting different execution architectures, such as C and C++ (which have the same semantics as Java for well-known sequential constructs), and JavaScript (as an online tool). However, a new C++

runtime system will be developed as part of a master thesis project. We expect to be able to report on this in the very near future.

Appendix A

List of Primitive Types in ProcessJ

In Java, the atomic types are: **boolean**, **byte**, **short**, **char**, **int**, **float**, **long**, **double**. ProcessJ uses the same names, but in addition, string is an atomic type. In addition to the regular atomic types, ProcessJ has an additional number (currently two) of atomic types inherited from occam-π, namely **timers** and **barriers**. Table A.1 contains a list of the primitive types.

Table A.1: List of primitive types in ProcessJ

Name	Representation	Range
`boolean`		{`true`, `false`}
`byte`	8-bit signed 2's complement	{-128,...,127}
`short`	16-bit signed 2's complement	{-32768,...,32767}
`char`	16-bit Unicode character	{'\u0000',...,'\ufff'}
`int`	32-bit signed 2's complement	$\{-2^{31}, \ldots, 2^{31}-1\}$
`long`	64-bit signed 2's complement	$\{-2^{63}, \ldots, 2^{63}-1\}$
`float`	single-precision 32-bit IEEE 754 floating point	$\pm 1.18 \times 10^{-38}$ - $\pm 3.4 \times 10^{38}$
`double`	double-precision 64-bit IEEE 754 floating point	$\pm 2.23^{-308}$ - $\pm 1.80 \times 10^{308}$
`string`		—
`barrier`		—
`timer`		—

Appendix B

List of Modifiers Available in ProcessJ

ProcessJ currently supports six different modifiers. Modifiers are keywords that can be placed in front of type and variable declarations. Not all modifiers can be used on all types. The list of modifiers can be seen in Table B.1

Table B.1: List of modifiers in ProcessJ

Modifier	Description
`mobile`	The `mobile` modifier can only be used on procedure types, channel, and channel end declarations. A mobile process is a process that can be communicated in a channel that carries the appropriate procedure type. A mobile channel is a channel whose ends can be sent over a channel, and a mobile channel end denoted the channel ends of a mobile channel.
`const`	The modifier `const` can only be used to declare constants at the compilation unit level like `const double PI = 3.1415` or it can be used to declare a local variable or even a parameter constant. A constant local cannot be assigned apart from its initializer, and a constant parameter cannot be assigned to at all apart from the value it gets when the procedure is called.
`native`	The `native` modifier does not appear in normal code. It is used only when writing native libraries for ProcessJ.

`public`	Only top-level types (that is, types declared outside procedures) can use the `public` modifier. The `public` modifier makes a type available to anyone who either imports the file in which it is declared or who implicitly causes an import by referring to the type using the `::` operator.
`private`	Like `public`, `private` is a top-level modifier. Any type declared `private` can only be accessed within the compilation unit in which it was declared.
`protected`	Again, `protected` is a top-level modifier only. A `protected` type is accessible only within the package.

Appendix C

Monitor example in Java

```java
public class Sample {
    public Object lock = new Object();

    public void f() {
        synchronized(lock) {
            // lock shared state
        }
    }
}
```

Listing C.1: Example of a locking mechanism in Java.

```
Compiled from "Sample.java"
public class Sample {
  public java.lang.Object lock;

  public Sample();
    Code:
       0: aload_0
       1: invokespecial #1 // Method
                                java/lang/Object."<init>":()V
       4: aload_0
       5: new           #2 // class java/lang/Object
       8: dup
       9: invokespecial #1 // Method
                                java/lang/Object."<init>":()V
      12: putfield      #3 // Field lock:Ljava/lang/Object;
      15: return

  public void f();
    Code:
       0: aload_0
       1: getfield      #3 // Field lock:Ljava/lang/Object;
       4: dup
       5: astore_1
       6: monitorenter
       7: aload_1
       8: monitorexit
       9: goto          17
      12: astore_2
      13: aload_1
      14: monitorexit
      15: aload_2
      16: athrow
      17: return
    Exception table:
       from    to  target type
           7     9    12   any
          12    15    12   any
}
```

Listing C.2: Byte code generated by the Java compiler.

Appendix D

Complete ‘kill’ Implementation

```
public void integrate(chan<int>.read in,
                     chan<int>.write out,
                     chan<boolean>.read killMe,
                     chan<boolean>.write killConsumer) {
  int total = 0;
  boolean ok = true;
  while (ok) {
    boolean y;
    int x;
    pri alt {
      y = killMe.read() :
        {
          killConsumer.write(true);
          ok = false;
        }
      x = in.read() :
        {
          total = total + x;
          out.write(total);
        }
    }
  }
}

public void producer(chan<int>.write out,
                     chan<boolean>.read killMe,
                     chan<boolean>.write killIntegrate) {
  int i = 0;
  boolean ok = true;
  while (ok) {
    boolean b;
    alt {
      b = killMe.read(): {
        ok = false;
        killIntegrate.write(true);
      }
      skip: {
        out.write(i);
        i=i+1;
      }
    }
  }
}

public void consumer(chan<int>.read in,
                     chan<boolean>.read killMe) {
  boolean ok = true;
  while (ok) {
    int x;
    boolean b;
    pri alt {
      x = in.read() : {
        println(x);
      }
      b = killMe.read(): {
        ok = false;
      }
    }
  }
}

public void killer(chan<boolean>.write killProduce) {
  timer t;
  t.timeout(3);
  killProduce.write(true);
}

public void main(string args[]) {
  chan<int> in, out;
  chan<boolean> killProduce, killIntegrate, killConsume;
  par {
    produce(in.write, killProduce.read, killIntegrate.write);
    integrate(in.read, out.write, killIntegrate.read,
                killConsume.write);
    killer(killProduce.write);
    consume(out.read, killConsume.read);
  }
}
```

Listing D.1: Correct kill implementation.

Appendix E

An Example of Threads Executing In and Out of Objects

```java
interface I {
    public void h() ;
}

class B implements I {
    public void h() {
        test.a.g();
    }
}

class A {
    private int i = 0;

    synchronized void f(I b) {
        int j = i;
        b.h();
        j = j + 1;
        i = j;
    }

    synchronized void g() {
        int j = i;
        j = j + 1;
        i = j;
    }

    public int get() {
        return i;
    }
}

public class test {
    public static A a;

    public static void main(String args[]) {
        a = new A();
        B b = new B();
        a.f(b);

        System.out.println(a.get());
    }
}
```

Listing E.1: Threads executing in and out of objects.

Appendix F

A Command Line Example Program

```
@Parameters(name="calc")
public class Example extends Command {
   @Option(names="-op1",
           help="first operand",
           split="=",
           metavar="<num>",
           arity="1",
           handlers=OperandParser.class)
   public int op1;
   @Option(names="-op2",
           help="second operand",
           metavar="<num>",
           arity="1",
           handlers=OperandParser.class)
   public int op2;
   @Option(names="-add",
           help="Adds two numbers",
           defaultValue="false")
   public boolean addition;
   @Option(names="-sub",
           help="subtract two numbers")
   public boolean subtraction;
   @Option(names="-help",
           help="Show this help message and exit",
           defaultValue="false")
   public boolean help;
   public static void main(String[] args) {
      CLIBuilder builder =
          new CLIBuilder().addCommand(Example.class);
      Example sp = null;
      try {
        builder.handleArgs("-add -op1=13 -op2 34".split(" "));
        sp = builder.getCommand(Example.class);
      } catch(Exception e) {
        System.out.println(e.getMessage());
        System.exit(1);
      }
      if (sp.addition) {
         System.out.println(String.format("Add operation: " +
         "%s + %s = %s", sp.op1, sp.op2, (sp.op1 + sp.op2)));
      } else if (sp.subtraction) {
         System.out.println(String.format("Add operation: " +
         "%s + %s = %s", sp.op1, sp.op2, (sp.op1 + sp.op2)));
      }
      if (sp.help) {
         Formatter formatHelp = new Formatter(builder);
         System.out.println(formatHelp.buildUsagePage());
         System.exit(0);
      }
   }
   public static class OperandParser
                    extends OptionParser<Integer> {
      public OperandParser(String optionName) {
         super(optionName);
      }
      @Override
      public Integer parseValue(String value)
                     throws Exception {
         try {
            return Integer.parseInt(value);
         } catch (NumberFormatException e) {
            throw new NumberFormatException(String.format(
                 "'%s' could not convert '%s' to "
                 + "Integer.", optionName, value));
         }
      }
   }
}
```

Listing F.1: An example of a command line program.

Appendix G

List of Options and Parameters in ProcessJ

Table G.1: List of options and parameters in ProcessJ

Type	Name	Parameters	Default Value	Meaning
command	`pjc`	option / argument	—	The primary command in ProcessJ is this command
option	`-cli`	boolean	false	ProcessJ command line interpreter and conventions
option	`-console-ansi-code`	integer	0	Try and use color on terminals that support ANSI espace codes
option	`-g` *or* `-debug`	boolean	false	Generate all debugging info
option	`-error-code`	integer	0	What error code information do you want?
option	`-h` *or* `-help`	boolean	false	Show this help message and exit
option	`-I` *or* `-include`	string	ProcessJ default directory	Override the default include directory which is set to be the "include" subdirectory of the ProcessJ directory
option	`-info`	string	null	Provide additional information about a specific command or option

option	`-logfile`	file	null	Use given file for log
option	`-sts`	boolean	false	Dump global symbol table structure
option	`-t` *or* `-target`	enum	JVM	Specify the target language. C: C source code is written, compiled, and linked with the CSSP runtime; C++: C++ source code is generated and compiled into an executable; JVM: JVM class files are written and compiled; JS: JavaScript is written
option	`-V` *or* `-verbose`	boolean	false	Output messages of the exact sequence of commands used to compile a ProcessJ program
option	`-v` *or* `-version`	boolean	false	Print version information and exit
option	`-visit-all`	boolean	false	Generate all parse tree visitors (not default)
argument	`args`	file	null	The file (or files) to compile

Appendix H

A Full 8-bit Adder Implementation in ProcessJ

```
public void main(string args[]) {
    chan<boolean> a0, a1, a2, a3, a4, a5, a6, a7;
    chan<boolean> b0, b1, b2, b3, b4, b5, b6, b7;
    chan<boolean> r0, r1, r2, r3, r4, r5, r6, r7;
    chan<boolean> inCarry, outCarry;

    boolean p0, p1, p2, p3, p4, p5, p6, p7;
    boolean q0, q1, q2, q3, q4, q5, q6, q7;

    // Addition results
    boolean f0, f1, f2, f3, f4, f5, f6, f7;
    boolean c, inC;

    // Selected numbers
    p0 = false;
    p1 = false;
    p2 = true;
    p3 = false;
    p4 = false;
    p5 = false;
    p6 = true;
    p7 = false;

    q0 = true;
    q1 = true;
    q2 = false;
    q3 = true;
    q4 = false;
    q5 = true;
    q6 = false;
    q7 = true;

    par {
        // First number
        a7.write(p7);
        a6.write(p6);
        a5.write(p5);
        a4.write(p4);
        a3.write(p3);
        a2.write(p2);
        a1.write(p1);
        a0.write(p0);

        // Second number
        b7.write(q7);
        b6.write(q6);
        b5.write(q5);
        b4.write(q4);
        b3.write(q3);
        b2.write(q2);
        b1.write(q1);
        b0.write(q0);

        // Initial carry
        inCarry.write(inC);

        eightBitAdder(a0.read, a1.read, a2.read, a3.read,
                      a4.read, a5.read, a6.read, a7.read,
                      b0.read, b1.read, b2.read, b3.read,
                      b4.read, b5.read, b6.read, b7.read,
                      inCarry.read, r0.write, r1.write,
                      r2.write, r3.write, r4.write, r5.write,
                      r6.write, r7.write, outCarry.write);

        f0 = r0.read();
        f1 = r1.read();
        f2 = r2.read();
        f3 = r3.read();
        f4 = r4.read();
        f5 = r5.read();
        f6 = r6.read();
        f7 = r7.read();

        c = outCarry.read();
    }

    println("   " + p7 + " " + p6 + " " + p5 + " " + p4 + " " + p3 + " " + p2 + " " + p1 + " " + p0 + " (InCarry:"  + inC + ")");
    println("+ " + q7 + " " + q6 + " " + q5 + " " + q4 + " " + q3 + " " + q2 + " " + q1 + " " + q0);
    println("----------");
    println(" " + f7 + " " + f6 + " " + f5 + " " + f4 + " " + f3 + " " + f2 + " " + f1 + " " + f0);
    println("Carry was: " + c);
}

public void notGate(chan<boolean>.read in, chan<boolean>.write out) {
    boolean x = false;
    x = in.read();
    out.write(!x);
}

public void orGate(chan<boolean>.read in1, chan<boolean>.read in2, chan<boolean>.write out){
    boolean x = false, y = false;
    par{
        x = in1.read();
        y = in2.read();
    }
    out.write(x  y);
}

public void andGate(chan<boolean>.read in1, chan<boolean>.read in2, chan<boolean>.write out) {
    boolean x = false, y = false;
    par {
        x = in1.read();
        y = in2.read();
    }
    out.write(x && y);
}

public void nandGate(chan<boolean>.read in1, chan<boolean>.read
↪ in2, chan<boolean>.write out) {
    chan<boolean> a;
    par {
        andGate(in1, in2, a.write);
        notGate(a.read, out);
    }
    return;
}

public void muxGate(chan<boolean>.read in, chan<boolean>.read
↪ out1, chan<boolean>.write out2) {
    boolean x = false; x = in.read();
    par {
        out1.write(x);
        out2.write(x);
    }
    return;
}

public void xorGate(chan<boolean>.read in1, chan<boolean>.read
↪ in2, chan<boolean>.write out){
    chan<boolean> a, b, c, d , e, f, g, h, i;
    par {
        muxGate(in1, a.read, b.write);
        muxGate(in2, c.read, d.write);
        nandGate(b.read, d.read, e.write);
        muxGate(e.read, f.read, g.write);
        nandGate(a.read, f.read, h.write);
        nandGate(c.read, g.read, i.write);
        nandGate(h.read, i.read, out);
    }
}

public void oneBitAdder(chan<boolean>.read in1,
↪ chan<boolean>.read in2, chan<boolean>.read in3,
↪ chan<boolean>.write result, chan<boolean>.write carry) {
    chan<boolean> a, b, c, d, e, f, g, h, i, j, k;
        par{
            muxGate(in1, a.read, b.write);
            muxGate(in2, c.read, d.write);
            xorGate(a.read, c.read, e.write);
            muxGate(e.read, f.read, g.write);
            muxGate(in3, h.read, i.write);
            xorGate(f.read, h.read, result);
            andGate(g.read, i.read, j.write);
            andGate(b.read, d.read, k.write);
            orGate(j.read, k.read, carry);
        }
}

public void fourBitAdder(chan<boolean>.read inA0,
 ↪ chan<boolean>.read inA1, chan<boolean>.read inA2,
 ↪ chan<boolean>.read inA3, chan<boolean>.read inB0,
 ↪ chan<boolean>.read inB1, chan<boolean>.read inB2,
 ↪ chan<boolean>.read inB3, chan<boolean>.read inCarry,
 ↪ chan<boolean>.write result0, chan<boolean>.write result1,
 ↪ chan<boolean>.write result2, chan<boolean>.write result3,
 ↪ chan<boolean>.write carry) {
    chan<boolean> a, b, c;
    par {
        oneBitAdder(inA0, inB0, inCarry, result0, a.write);
        oneBitAdder(inA1, inB1, a.read, result1, b.write);
        oneBitAdder(inA2, inB2, b.read, result2, c.write);
        oneBitAdder(inA3, inB3, c.read, result3, carry);
    }
}

```

```
public void eightBitAdder(chan<boolean>.read inA0,
↪ chan<boolean>.read inA1, chan<boolean>.read inA2,
↪ chan<boolean>.read inA3, chan<boolean>.read inA4,
↪ chan<boolean>.read inA5, chan<boolean>.read inA6,
↪ chan<boolean>.read inA7, chan<boolean>.read inB0,
↪ chan<boolean>.read inB1, chan<boolean>.read inB2,
↪ chan<boolean>.read inB3, chan<boolean>.read inB4,
↪ chan<boolean>.read inB5, chan<boolean>.read inB6,
↪ chan<boolean>.read inB7, chan<boolean>.read inCarry,
↪ chan<boolean>.write result0, chan<boolean>.write result1,
↪ chan<boolean>.write result2, chan<boolean>.write result3,
↪ chan<boolean>.write result4, chan<boolean>.write result5,
↪ chan<boolean>.write result6, chan<boolean>.write result7,
↪ chan<boolean>.write outCarry) {
    chan<boolean> a;
    par {
        fourBitAdder(inA0, inA1, inA2, inA3,
                     inB0, inB1, inB2, inB3,
                     inCarry, result0, result1,
                     result2, result3, a.write);
        fourBitAdder(inA4, inA5, inA6, inA7,
                     inB4, inB5, inB6, inB7,
                     a.read,
                     result4, result5, result6,
                     result7, outCarry);
    }
}
```

Listing H.1: An 8-bit adder implementation in ProcessJ.

Appendix I

CommsTime Implementation in ProcessJ

```
public void prefix(long n, chan<long>.read in,
                   chan<long>.write out) {
   out.write(n);
   long l = 0;
   while (l < 1000000) {
     l = in.read();
     out.write(l);
   }
}

public void succ(chan<long>.read in, chan<long>.write out) {
   long l = 0;
   while (l < 999999) {
     l = in.read();
     out.write(l+1);
   }
}

public void delta(chan<long>.read in, chan<long>.write out1,
                  chan<long>.write out2) {
   long l = 0;
   while (l < 1000000) {
     l = in.read();
     par {
       out1.write(l);  // consumer
       if (l != 1000000)
         out2.write(l);  // succ
     }
```

```
    }
}

public void consumer(chan<long>.read in) {
    long l = 0;
    while (l < 1000000) {
      l = in.read();
      println(l);
    }
}

public void main(string args[]) {
    chan<long> a,b,c,d;
    par {
      delta(d.read, a.write, b.write);
      succ(b.read, c.write);
      prefix(0, c.read, d.write);
      consumer(a.read);
    }
}
```

Listing I.1: CommsTime code.

Bibliography

[1] 3D Blood Clotting - Programming Languages and Systems Research Group Wiki. https://www.cs.kent.ac.uk/research/groups/plas/wiki/3D_Blood_Clotting. (Accessed on 03/31/2019).

[2] Akka anti-patterns: shared mutable state - manuel bernhardt. https://manuel.bernhardt.io/2016/08/02/akka-anti-patterns-shared-mutable-state/. (Accessed on 03/30/2019).

[3] Akka: build concurrent, distributed, and resilient message-driven applications for Java and Scala — Akka. https://akka.io/. (Accessed on 03/20/2019).

[4] Argparse — Parser for command-line options, arguments and sub-commands — Python 3.7.2 documentation. https://docs.python.org/3/library/argparse.html. (Accessed on 6/17/2018).

[5] args4j parent - How to use Args4J. http://args4j.kohsuke.org/sample.html. (Accessed on 04/24/2019).

[6] ASM 4.0 A Java bytecode engineering library. https://asm.ow2.io/asm4-guide.pdf. (Accessed on 04/21/2019).

[7] C++CSP2. https://www.cs.kent.ac.uk/projects/ofa/c++csp/. (Accessed on 03/20/2019).

[8] CLAJR. http://clajr.sourceforge.net/. (Accessed on 04/24/2019).

[9] Commons CLI Home. https://commons.apache.org/proper/commons-cli/. (Accessed on 12/27/2018).

[10] Communicating Haskell Processes. https://www.cs.kent.ac.uk/projects/ofa/chp/. (Accessed on 03/20/2019).

[11] Communicating Sequential Processes for Java (JCSP). https://www.cs.kent.ac.uk/projects/ofa/jcsp/. (Accessed on 01/17/2019).

[12] cspbook.pdf. http://www.usingcsp.com/cspbook.pdf. (Accessed on 03/14/2019).

[13] Erlang Programming Language. https://www.erlang.org/. (Accessed on 03/19/2019).

[14] FDR4 - The CSP Refinement Checker. https://www.cs.ox.ac.uk/projects/fdr/.

(Accessed on 03/16/2019).

[15] Getopt (The GNU C Library). https://www.gnu.org/software/libc/manual/html_node/Getopt.html. (Accessed on 6/7/2018).

[16] iwomp2005_tutorial_openmp_rvdp.pdf. http://www.nic.uoregon.edu/iwomp2005/iwomp2005_tutorial_openmp_rvdp.pdf. (Accessed on 03/17/2019).

[17] JArgs command line option parsing suite for Java. http://jargs.sourceforge.net/. (Accessed on 04/24/2019).

[18] JOpt Simple - a Java command line parsing library JOpt Simple. https://jopt-simple.github.io/jopt-simple/. (Accessed on 04/24/2019).

[19] JSAP: Java Simple Argument Parser. http://www.martiansoftware.com/jsap/. (Accessed on 04/24/2019).

[20] Message Passing Interface (MPI). https://computing.llnl.gov/tutorials/mpi/#What. (Accessed on 03/17/2019).

[21] Microsoft PowerPoint - JCSP. https://www.cs.kent.ac.uk/projects/ofa/pdpta/jcsp.pdf. (Accessed on 03/16/2019).

[22] Microsoft Word - INTERTWinE_Best_Practice_Guide_MPI+OpenMP_1.1.docx. https://www.intertwine-project.eu/sites/default/files/images/INTERTWinE_Best_Practice_Guide_MPI%2BOpenMP_1.1.pdf. (Accessed on 03/28/2019).

[23] occam-pi and KRoC: blending CSP and the pi-calculus. https://www.cs.kent.ac.uk/projects/ofa/kroc/. (Accessed on 01/17/2019).

[24] OccamPi Reference - Programming Languages and Systems Research Group Wiki. https://www.cs.kent.ac.uk/research/groups/plas/wiki/OccamPiReference/. (Accessed on 01/19/2019).

[25] OpenMp-examples-4.5.0.pdf. https://www.openmp.org/wp-content/uploads/openmp-examples-4.5.0.pdf. (Accessed on 03/21/2019).

[26] The Go Programming Language. https://golang.org/. (Accessed on 03/20/2019).

[27] Threads and Swing. https://www.comp.nus.edu.sg/~cs3283/ftp/Java/swingConnect/archive/tech_topics_arch/threads/threads.html. (Accessed on 04/28/2019).

[28] Why has the actor model not succeeded? https://www.doc.ic.ac.uk/~nd/surprise_97/journal/vol2/pjm2/. (Accessed on 03/20/2019).

[29] G. A. Agha. Actors: A Model of Concurrent Computation in Distributed Systems. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB, 1985.

[30] J. Armstrong. Erlang. *Communications of the ACM*, 53(9):68–75, 2010.

[31] D. Aspinall. *The Microprocessor and its Application: An Advanced Course*. Cambridge University Press, dec 1978.

[32] F. R. Barnes. Guppy: Process-Oriented Programming on Embedded Devices. 2015.

[33] F. R. Barnes and P. H. Welch. Mobile Data, Dynamic Allocation and Zero Aliasing: An occam Experiment. *Communicating process architectures*, 59:243–264, 2001.

[34] F. R. Barnes and P. H. Welch. Communicating Mobile Processes. *Communicating process architectures*, 62:201–218, 2004.

[35] J. Bloch. *Effective Java (2nd Edition)*. Addison-Wesley, may 2008.

[36] N. C. Brown. Communicating Haskell Processes: Composable Explicit Concurrency using Monads. In *CPA*, pages 67–83, 2008.

[37] N. C. Brown and P. H. Welch. An introduction to the kent c++ csp library. *Communicating Process Architectures 2003*, 61:139–156, 2003.

[38] E. Bruneton, R. Lenglet, and T. Coupaye. Asm: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, 30(19), 2002.

[39] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[40] A. Danalis, L. Pollock, M. Swany, and J. Cavazos. MPI-aware compiler optimizations for improving communication-computation overlap. In *Proceedings of the 23rd international conference on Supercomputing*, pages 316–325. ACM, 2009.

[41] J. Davies. Setting real-time CSP. *Computing Laboratory, Oxford University*, 1994.

[42] I. East, J. Martin, P. Welch, D. Duce, M. Green, et al. Communicating Mobile Processes.

[43] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *2011 38th Annual international symposium on computer architecture (ISCA)*, pages 365–376. IEEE, 2011.

[44] J. R. Fonseca Cacho. Engaging assignments increase performance. 2019.

[45] J. R. Fonseca Cacho and K. Taghva. Reproducible research in document analysis and recognition. In *Information Technology-New Generations*, pages 389–395. Springer, 2018.

[46] J. R. Fonseca Cacho and K. Taghva. Aligning ground truth text with ocr degraded text, 2019. Paper Presented at Computing Conference, London, UK.

[47] J. R. Fonseca Cacho, K. Taghva, and D. Alvarez. Using the google web 1t 5-gram corpus for ocr error correction. In *Information Technology-New Generations*. Springer, 2019. in press.

[48] D. J. Frank, R. H. Dennard, E. Nowak, P. M. Solomon, Y. Taur, and H.-S. P. Wong. Device scaling limits of si mosfets and their application dependencies. *Proceedings of the IEEE*, 89(3):259–288, 2001.

[49] E. Gamma. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education India, 1995.

[50] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, nov 1994.

[51] J. Gibbons. Design Patterns as Higher-Order Datatype-Generic Programs. In *Proceedings of the 2006 ACM SIGPLAN workshop on Generic programming*, pages 1–12. ACM, 2006.

[52] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Pearson Education, 2006.

[53] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface (Scientific and Engineering Computation)*. The MIT Press, nov 2014.

[54] W. D. Gropp. Learning from the Success of MPI. In *International Conference on High-Performance Computing*, pages 81–92. Springer, 2001.

[55] P. B. Hansen. Structured Multiprogramming. *Communications of the ACM*, 15(7):574–578, 1972.

[56] O. Hernandez, F. Song, B. Chapman, J. Dongarra, B. Mohr, S. Moore, and F. Wolf. Performance Instrumentation and Compiler Optimizations for MPI/OpenMP Applications. In *International Workshop on OpenMP*, pages 267–278. Springer, 2005.

[57] C. Hewitt. Viewing Control Structures as Patterns of Passing Messages. *Artificial intelligence*, 8(3):323–364, 1977.

[58] C. Hewitt. Actor Model of Computation: Scalable Robust Information Systems. *arXiv preprint arXiv:1008.1459*, 2010.

[59] G. Hilderink, J. Broenink, W. Vervoort, and A. Bakkers. Communicating Java Threads. In *Parallel Programming and Java, Proceedings of WoTUG*, volume 20, pages 48–76, 1997.

[60] G. H. Hilderink, J. F. Broenink, and A. Bakkers. Communicating Threads for Java. In *Proceedings of WoTUG-22: Architectures, Languages and Techniques for Concurrent Systems*, pages 243–261, 1999.

[61] C. Hinton and C. R. Ivimey. Legacy of the transputer. *emergence*, 80186:19.

[62] C. A. R. Hoare. Monitors: An Operating System Structuring Concept. In *The origin of concurrent programming*, pages 272–294. Springer, 1974.

[63] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*,

21(8):666–677, 1978.

[64] C. A. R. Hoare, S. D. Brookes, and A. W. Roscoe. *A theory of Communicating Sequential Processes*. Oxford University Computing Laboratory, Programming Research Group, 1981.

[65] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.

[66] R. Love. *Linux Kernel Development (3rd Edition)*. Addison-Wesley Professional, jul 2010.

[67] D. Malik. *C++ Programming*. Thomson, 2011.

[68] D. May. CSP, occam and Transputers. In *Communicating Sequential Processes. The First 25 Years*, pages 75–84. Springer, 2005.

[69] D. May and R. Taylor. Occam-an overview. *Microprocessors and Microsystems*, 8(2):73–79, 1984.

[70] R. Milner. *Communicating and Mobile Systems: the pi calculus*. Cambridge university press, 1999.

[71] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, i. *Information and computation*, 100(1):1–40, 1992.

[72] J. Moores. Ccsp-a portable csp-based run-time system. In *Architectures, languages and techniques for concurrent systems: WoTUG-22, proceedings of the 22nd World Occam and Transputer User Group Technical Meeting, 11-14 April 1999, Keele, United Kingdom*, volume 57, page 147. IOS Press, 1999.

[73] P. Niemeyer and D. Leuck. *Learning Java: A Bestselling Hands-On Java Tutorial*. O'Reilly Media, 4 edition, 2013.

[74] S. Oaks and H. Wong. *Java Threads: Understanding and Mastering Concurrent Programming*. " O'Reilly Media, Inc.", 2004.

[75] G. Oprean. An implementation of active objects in java. 2007.

[76] G. Oprean and J. B. Pedersen. Asynchronous Active Objects in Java. In *CPA*, pages 237–254, 2008.

[77] T. J. Parr. Enforcing Strict Model-View Separation in Template Engines. In *Proceedings of the 13th international conference on World Wide Web*, pages 224–233. ACM, 2004.

[78] J. Pedersen and K. Chalmers. Verifying Channel Communication Correctness for a Multi-Core Cooperatively Scheduled Runtime Using CSP. In *Proceedings of the 2019 FormaliSE workshop at ICSE*, May 2019.

[79] J. B. Pedersen and B. Kauke. Resumable Java Bytecode-Process Mobility for the JVM. In *CPA*, pages 159–172, 2009.

[80] J. B. Pedersen and M. L. Smith. ProcessJ: A Possible Future of Process-Oriented Design. *Communicating Process Architectures*, pages 133–156, 2013.

[81] J. B. Pedersen and M. Sowders. Static Scoping and Name Resolution for Mobile Processes with Polymorphic Interfaces. In *CPA*, pages 71–85, 2011.

[82] J. B. Pedersen and A. Stefik. Towards Millions of Processes on the JVM. *Communicating Process Architectures*, 71, 2014.

[83] R. Rabenseifner, G. Hager, and G. Jost. Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-core SMP Nodes. In *2009 17th Euromicro international conference on parallel, distributed and network-based processing*, pages 427–436. IEEE, 2009.

[84] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Bell System Technical Journal*, 57(6):1905–1929, 1978.

[85] C. G. Ritson, A. T. Sampson, and F. R. Barnes. Multicore scheduling for lightweight communicating processes. In *International Conference on Coordination Languages and Models*, pages 163–183. Springer, 2009.

[86] C. G. Ritson, A. T. Sampson, and F. R. Barnes. Multicore scheduling for lightweight communicating processes. *Science of Computer Programming*, 77(6):727–740, 2012.

[87] C. G. Ritson and P. H. Welch. A Process-Oriented Architecture for Complex System Modelling. *Concurrency and Computation: Practice and Experience*, 22(8):965–980, 2010.

[88] B. Roscoe. A Theory of Communicating Sequential Processes. *Journal of the ACM*, (3), 1984.

[89] D. Sangiorgi and D. Walker. *The pi-calculus: a Theory of Mobile Processes*. Cambridge university press, 2003.

[90] N. C. Schaller, G. H. Hilderink, and P. H. Welch. Using java for parallel computing: Jcsp versus ctj, a comparison. *Communicating process architectures*, 2000:205–226, 2000.

[91] R. R. Schaller. Moore's law: past, present and future. *IEEE spectrum*, 34(6):52–59, 1997.

[92] H. Schildt. *Java The Complete Reference, 8th Edition*. McGraw-Hill Education, 8 edition, 2011.

[93] S. Schneider, A. Cavalcanti, H. Treharne, and J. Woodcock. A Layered Behavioural Model of Platelets. In *11th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'06)*, pages 9–pp. IEEE, 2006.

[94] C. Shrestha and J. B. Pedersen. JVMCSP-Approaching Billions of Processes on a Single-Core JVM. 2016.

[95] C. D. Shrestha. The JVMCSP Runtime and Code Generator for ProcessJ in Java. 2016.

[96] L. Smith. Mixed mode mpi/openmp programming. *UK High-End Computing Technology Report*, pages 1–25, 2000.

[97] L. Smith and M. Bull. Development of mixed mode MPI/OpenMP applications. *Scientific Programming*, 9(2-3):83–98, 2001.

[98] M. Sowders. ProcessJ: A Process-Oriented Programming Language. 2011.

[99] M. Sowders and J. B. Pedersen. Mobile Process Resumption in Java Without Bytecode Rewriting. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, page 1. The Steering Committee of The World Congress in Computer Science, Computer , 2011.

[100] S. Stepney, F. A. Polack, and H. R. Turner. Engineering Emergence. In *11th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'06)*, pages 9–pp. IEEE, 2006.

[101] R. Thakur and W. Gropp. Open issues in MPI implementation. In *Asia-Pacific Conference on Advances in Computer Systems Architecture*, pages 327–338. Springer, 2007.

[102] B. Vinter, J. M. Bjørndalen, and R. M. Friborg. PyCSP Revisited. In *Cpa*, pages 263–276, 2009.

[103] P. H. Welch. Java Threads in the Light of occam/CSP. In *Architectures, Languages and Patterns for Parallel and Distributed Applications, Proceedings of WoTUG*, volume 21, pages 259–284, 1998.

[104] P. H. Welch. Process Oriented Design for Java: Concurrency for All. *Lecture Notes in Computer Science*, pages 687–687, 2002.

[105] P. H. Welch. Life of occam-pi. *Communicating Process Architectures 2013*, pages 293–318, 2013.

[106] P. H. Welch and F. R. Barnes. Communicating Mobile Processes. In *Communicating Sequential Processes. The First 25 Years*, pages 175–210. Springer, 2005.

[107] P. H. Welch, N. C. Brown, J. Moores, K. Chalmers, and B. H. Sputh. Integrating and extending JCSP. 2007.

[108] P. H. Welch, K. Wallnau, A. T. Sampson, and M. Klein. To boldly go: an occam-π mission to engineer emergence. *Natural Computing*, 11(3):449–474, 2012.

[109] J. Wexler. *Concurrent programming in OCCAM 2*. Ellis Horwood New York, 1989.

Curriculum Vitae

Graduate College

University of Nevada, Las Vegas

Benjamin Cisneros Merino

Email: benjcisneros@gmail.com

Degrees:

Bachelor of Science in Computer Science 2017

University of Nevada Las Vegas

Thesis Title: ProcessJ: The JVMCSP Code Generator

Thesis Examination Committee:

Chairperson, Dr. Jan Bækgaard Pedersen, Ph.D.

Committee Member, Dr. Kazem Taghva, Ph.D.

Committee Member, Dr. Laxmi Gewali, Ph.D.

Graduate Faculty Representative, Dr. Emma E. Regentova, Ph.D.