12-1-2020

The ProcessJ C++ Runtime System and Code Generator

Alexander Christian Thomason

THE PROCESSJ C++ RUNTIME SYSTEM AND CODE GENERATOR

By

Alexander C. Thomason

Bachelor of Science — Computer Science
University of Nevada, Las Vegas
2018

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
December 2020

UNLV | GRADUATE COLLEGE

Thesis Approval

The Graduate College
The University of Nevada, Las Vegas

November 30, 2020

This thesis prepared by

Alexander C. Thomason

entitled

The ProcessJ C++ Runtime System and Code Generator

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

Master of Science in Computer Science
Department of Computer Science

Jan Pedersen, Ph.D.
Examination Committee Chair

Kathryn Hausbeck Korgan, Ph.D.
Graduate College Dean

Kazem Taghva, Ph.D.
Examination Committee Member

Laxmi Gewali, Ph.D.
Examination Committee Member

Sarah Harris, Ph.D.
Graduate College Faculty Representative

Abstract

ProcessJ is a modern Process-Oriented language that builds on previous work from other languages like occam and occam-pi. However, the only readily-available runtime system is built on top of the Java Virtual Machine (JVM). This is not a choice made intentionally, but simply out of a lack of other implementations – until now. This thesis introduces the new C++-based runtime system for ProcessJ, coupled with a new C++ code generator for the ProcessJ compiler. This thesis later examines the implementation details of the runtime system, including the components that make it up. We also examine the ability to cooperatively schedule many processes within the runtime environment, inside a separate scheduling system built on top of traditional operating system threading, rather than simply mapping processes one-to-one with threads. We later exemplify some of the cooperatively-schedulable code generated by the compiler, giving a complete rundown of the constituents and their various design choices. Lastly, we show the results of several tests that demonstrate the performance benefits of a bespoke C++-based runtime system, and discuss the future work and optimizations of this system.

Acknowledgements

"First and foremost I would like to thank my parents Leo and Annalloyd Thomason. Without their continuous love and support, I would not be anywhere near the man I am today, nor would I have thought I even had a chance at academia being my place in life.

I would also like to give a very serious thank you to Dr. Pedersen for his guidance and support throughout the entire process of going through with getting a Master's Degree, from the entry process to actually writing and finishing a Master's Thesis, and everything else in between. I also think I owe him another thank you for helping me realize that there are enough interesting things about compilers, CSP, and other related topics to keep me (happily) busy for a lifetime. ;^)

I want to thank my committee members: Dr. Taghva, Dr. Gewali, and Dr. Harris.

Finally, I want to thank all of the students that I have the pleasure of being a Graduate Assistant for, and especially all of the other Graduate Assistants that I work with for being some of the most wonderful friends that I think I will ever make in academia, as well as some of the best brains to pick when I'm stuck on something."

Alexander C. Thomason

University of Nevada, Las Vegas

December 2020

Table of Contents

List of Tables

List of Listings

Chapter 1

Introduction

When one considers the idea of modern computing – and by association hardware limitations – Moore's Law typically comes to mind. This law states that as time goes on, the number of transistors in any integrated circuit will double, but the price of said integrated circuit will halve. We have seen this hold true again and again in the past decades. This law is commonly associated with hardware limitations, not software. When we think of modern computers (at least at the time this paper was written), we probably only think about the vast amounts of RAM and high CPU speeds that are made possible (and affordable) by the phenomenon described in Moore's Law. However, another similarly-important but less-thought-of law better describes the improvement of software performance in terms of cores-per-CPU, and threads-per-core. This law is known as Amdahl's Law [WA05], and is mathematically given as:

$$S(p) = \frac{p}{1 + (p-1)f} \tag{1.1}$$

where p is the number of processors a machine has, and f is the percentage of any program that is not able to be divided into concurrent tasks. This equation describes the speedup factor of a program when parallelized. Think of the simple task of doing your laundry: you might place all your clothes in a washing machine, then wait for them to finish and then move them to a tumble dryer, then wait for them to finish drying before finally folding your clothes back up and putting them in your closet or dresser. We can describe these tasks are sequential, in that you cannot wash your clothes and dry them at the same time (unless you have some sort of cutting-edge laundry machine). However, if you wait for your clothes to finish washing by doing some other sort of chore before moving them to the dryer, and then do the same with your time spent waiting on the dryer, then you have made your whole task of laundry concurrent by *interleaving* other tasks in with

the whole process. In other words, you multiplex over several tasks in a sort of ordered system. Suppose you are doing several chores at one time: laundry, dishes, mopping, or other common household tasks. Suppose that your laundry has just been put in the washing machine and is just now starting to run, and your dryer is also running, so you don't need to check on that. Then you can move on to another task like putting the dishes into the dishwasher, or taking them out and putting them away. Then, as soon as you have put away all of the dishes, you can come back to your washing machine (or dryer) to see if the laundry is ready to be moved to the next stage of the full task of *doing* your laundry, and move the full process itself along as a result. Alternatively, the washing machine or dryer may still be running, which allows you to move on to some other chore, like mopping your floors. Now suppose you had another person around, and divided the work between the two of you (perhaps your friend could put clothes into the washer while you take clothes out of the dryer, for example). The two of you would be an example of parallel "processes" being performed at the same time by two different "processing units." Now we can observe the beauty of Amdahl's Law. Suppose that 15% of the entire process of doing laundry is sequential, like the time taken for one person to load laundry, move loads between machines, and finally remove and put away the laundry. If we also assume that there are at least 2 loads of laundry that need doing, this gives us a base speedup factor for one person doing laundry:

$$\begin{aligned} S(1) &= \frac{1}{1 + (1-1)0.15} \\ &= 1 \end{aligned}$$

and a speedup factor for two people doing laundry:

$$\begin{aligned} S(2) &= \frac{2}{1 + (2-1)0.15} \\ &= 1.739... \end{aligned}$$

This is a prime example of the beauty of concurrency and parallelism. Rather than just getting everything done in sequence, we can break up the entire process into parallel and sequential parts, distributing the parallel sections to several "workers" and getting the entire job done faster overall than if we had just left it purely sequential. However, with current language paradigms, tackling the issue of concurrency and parallelism is not an easy task. Several constructs like mutexes, semaphores, monitors, and more have been developed in the past as a sort of shim that protects against the issue of data race conditions. These tools come with their own set of problems, like deadlock, that pose incredibly complex problems for the programmers that use them. Knowing this,

it would be beneficial to use a language that builds on top of state-of-the-art programming tools to allow for concurrent and parallel programs to run correctly, as a layer of abstraction that simplifies the task of implementation without directly exposing the programmers utilizing the language to the aforementioned issues. This would certainly be a great problem to solve using ProcessJ, as it is a language that uses the process-oriented paradigm. This paradigm is such a good fit for our problem because it does exactly that: it abstracts the lower-level elements of ensuring proper concurrent behavior, and gives programmers the higher-order tools to aid them in their task of developing concurrent systems. Utilizing the ProcessJ language as developed thus far, we can further extend our solution to not just one target like the JVM, but also to others as needed. One such extension is towards C++, and is exactly the extension that has been designed and written to accompany this thesis.

Now that we have outlined a basic understanding of concurrency and parallelism, the question remains: "what can we do to implement software using concurrent and parallel concepts?" A naïve answer to that question would be to use threads and processes at the operating system level that use a shared memory pool for their objects and other data constructs. These threads and processes can also utilize mutexes to guard against data races. However, there are some problems with this model that should be considered.

The first of these is that threads and processes, as they currently stand in implementation, are too coarse-grain for the performance and scalability that we want to achieve. The most threads you could run on a single core of a processor is limited – perhaps around ten thousand or so at a time – which is quite shy of millions, or even billions. Another problem is that, as the number of threads and processes in a solution increases, the complexity of correct implementation using locking mechanisms also increases. In other words, the more threads you use, the more difficult it is to achieve the correct behavior while still guarding against race conditions. It would greatly benefit the programmer if there were some sort of easier way to reason about these abstract processes and their involvement and cooperation with other processes.

Another problem in the way of concurrency and parallelism today is that these concepts and object-oriented programming do not mix together as easily as one might think. Object-oriented programming relies on the concept of passive objects that are manipulated by the process or threads that need to use it. The objects themselves are not the "active" part of the software, and are not in control of their own code, or even their own data. Object-oriented programming, therefore, is not as analogous to modeling the real world as we would like. In the real world, "objects" have

their own autonomy, and operate themselves, instead of being operated on. In the world of object-oriented programming, an object can be passed around to many different threads of control, which in turn can operate on this object in many different ways. Even with the proper amount of locking to avoid race conditions, there is still a problem with too many external factors having access to an object at one point in time, with behavior that could potentially change the state of an object without all of the other entities knowing about this change. The result of this happening can be catastrophic at best, and would be better avoided altogether.

Enter CSP: Communicating Sequential Processes. While this concept is touched on in greater detail in a later chapter of this paper, one of the core ideas from this process algebra is clearly what we would like to include in our solution to the above problems. The idea of synchronous (in coordination), blocking (waiting), non-buffered message passing is more accurate in describing one of the many ways processes can interact with each other. More importantly, each procedure as described by CSP runs its own process itself, and has full autonomy over its execution and data, unlike the passive nature of objects. Processes, on the other hand, as described by CSP, are active themselves, and can decide whether or not they want to communicate with other processes, as well as *which* processes they want to communicate with. And, since communication is synchronous, we can formally reason about the program itself in terms of its behavior. This means that we can prove that a program under this model of thought can be free of deadlock or livelock. We can also formally prove if one process refines another, or if one process is closer to the exact behavior we want than its predecessor(s).

For this to be achieved, we need a few basic things. First, we need a language to be able to design and create programs that use this system. The language ProcessJ is just that language: a process-oriented one, rather than object-oriented. Next, we need a compiler that can take programs written using this language and produce executable forms that perform exactly to the specifications described above. Thankfully, the work done in [Cis19] gives us a powerful compiler that turns ProcessJ code into JARs that can be run using the Java Virtual Machine, that behave according to the concurrent and parallel semantics we want. Lastly, we need a runtime for these programs to utilize. It is not enough to build programs that utilize locking mechanisms and threads; a serious environment that introduces formal constructs of CSP in a way that both simplifies programming to the user and guarantees the behavior according to the aforementioned concepts is well-warranted. This has been formalized, proved correct, and modeled by [Cis19], [SP16], [Shr16], [PC19]. However, even this is not enough. Not only is a foundation for a runtime necessary, but

an actual implementation of this runtime, along with a powerful scheduling system, is required to fully realize and implement these components. The groundwork for this has also been done by [Cis19] for the Java Virtual Machine, but with some pitfalls and issues relating to performance. Thankfully, we have a new basis for this runtime system: the C++ programming language. Using C++ and its constructs for a runtime system, along with the ability to generate binary executables, we can potentially achieve performance greater than that of the Java Virtual Machine by removing the extra layer of abstraction that it is built on. This paper will introduce, explain, and explore in great detail the design, implementation, and implications of this new runtime system, along with the appropriate code generator that has been added to the aforementioned compiler.

1.1 Motivation

The motivation for this thesis comes from the future work described in [Cis19]. It is mentioned that having several different runtime systems written on top of different languages is a substantial benefit to developers because of the advantage of being able to use language-specific libraries without throwing out the idea of backward compatibility. This is also an obvious benefit because developers may want several different target languages and builds to fit their specific project needs. To be more specific the ProcessJ compiler's C++ code generation potentially allows us to create a very smooth way for ProcessJ programs to communicate and work together with other C++-based programs. The same can be said about other potential target languages, which will be described later in Chapter 7. Also, there are several performance benefits to having a runtime built on top of C++. The ability of ProcessJ's compiler to utilize a C++ compiler to produce binary executables gives us the potential ability to use C++ compiler optimizations on the generated code and therefore potentially improve the runtime performance of the program itself on top of the added benefit of concurrency.

One of the more pertinent motivations behind the C++ runtime system and code generator for ProcessJ is to build a runtime system that is not coupled to the pitfalls of utilizing the JVM and Java as the main underlying environment for ProcessJ. This difference in implementation opens up new opportunities for performance, specialized language features, and more. As will be shown later in this paper, the JVM implementation of ProcessJ's runtime system has been tested to show great performance increases over other runtime systems. However, the question posed in [SP16], "how many processes can be run on a single machine,", is yet to be answered with "billions." This new runtime system has been conceptualized (and now implemented) with the hopes that it will

be able to not only outperform its predecessors, but finally achieve the scalability that has been sought after. A new opportunity to further the performance of process-oriented languages has been opened by utilizing a different underlying language, and this effort has been made as a way to show that the current performance benefits and efficiency of the JVM implementation can be expanded upon.

To describe the work done on the ProcessJ C++ runtime and code generator, there are two main points to be made. First, an efficient and accurate port of the JVM runtime system needed to be made. Second, the proper code to utilize this new C++-based runtime system needed to be written to take full advantage of the many features introduced. This paper will go into appropriate detail as described in the next section.

1.2 Thesis Outline

In this thesis, Chapter 2 will lay some background knowledge to be able to understand the underlying ideas behind the language and its concepts, like CSP. Chapter 2 also touches on some of the history of process-oriented languages, runtime systems, and libraries, and their methods of tackling the problem of concurrency and parallelism in programming languages. Chapter 3 covers the implementation of the ProcessJ C++ runtime system in terms of constructs, operations, and Chapter 4 shows some examples of the code generated by the C++ code generator. Later, Chapter 5 shows some simple test programs written in ProcessJ, and then their execution results. Chapter 6 will conclude with a summary of all work done, and the results of this project's completion. Finally, Chapter 7 will outline any future work both introduced by this Thesis' completion and carried over from previous publications.

Chapter 2

Background Knowledge

The world of computer science is saturated with languages that incorporate the paradigm known as Object-Oriented Programming. This paradigm is particularly useful when describing objects within a system and how they interact alone or with each other. Some of these languages, like C++, build on top of another pre-existing set of operations and constructs, such as C. For example, in C++, you have the option to use the underlying constructs as if you were writing in C, because C++ was built as an extension of C. You have access directly to any C function so long as you include it in your program, as well as any of the functions that C++ offers itself. As another slightly different example, Python builds on top of several other languages such as C (CPython) [Fou20a], Java (Jython) [Fou20b], and more, while providing exclusively its own syntax and semantics: the C function strtok() is not available immediately to somebody using Python, but you can imagine that there are a multitude of functions written in CPython that use this function under the hood. The same can be said about other functions in Java that are used in the implementation of Jython, and so on. Both of these languages offer solutions to common problems from an object-oriented point of view using the base operations offered by the target language one way or another. ProcessJ differs from C++ and Java in that ProcessJ is a Process-Oriented Programming language, meaning that the basic elements are not objects, but processes themselves. If we look into C++ and Java specifically, we see that there are object-oriented solutions to concurrency and parallelism built-in, but several of their more dangerous lower-level constituents are bare and exposed to the programmer, leaving room for a lot of errors. This is the main problem that ProcessJ solves: instead of leaving the messy details of concurrent and parallel programming to the programmer, we provide an abstraction on top of these constructs, such as mutexes in C++ and Java monitors, and base the behavior of said abstraction on the process algebra known as CSP, Communicating

Sequential Processes.

2.1 Communicating Sequential Processes

Communicating Sequential Processes [Hoa85] is a process algebra designed by Sir Tony C.A.R. Hoare. It allows processes to be behaviorally described by events in sequence. It also allows those processes to be placed in sequence or in parallel with one another, and to communicate with one another via channels. This section will lay the background knowledge of CSP required to understand the inner workings of ProcessJ, and later on the C++ runtime system that implements several of the constructs of CSP. The examples given in the below subsections are taken from [Ros10].

2.1.1 Processes

In the world of CSP, the most basic building block we have is a single event. It may belong to a set of events called an alphabet. A single event in our alphabet is the opportunity for a process to engage in some sort of communication with its external environment. The process in question and its external environment must agree on this event before anything can be accomplished. That is, if the process is willing to engage in some event, the process can only proceed in doing so if the environment itself is also willing to engage in the same event. A process may also decide to simply not communicate or engage in anything at all. This process is called STOP, and does just that. Another process, called SKIP, is the process that successfully terminates execution.

2.1.2 Prefixing

We can combine processes in CSP with other processes by using the prefixing operator, $\rightarrow$, in order to create a process that engages on some event from our alphabet, and then stops. To demonstrate this operator, given some event α in our alphabet Σ, the process

$$P = \alpha \rightarrow STOP$$

is a process that will engage in the event α and then continue to behave like STOP. Here we see the simplest example of prefixing with STOP. We can go further to define processes that use prefixing to engage in several atomic events sequentially and then stop using this method. For instance, if we wanted to describe a simple daily routine using CSP, we could define a process as follows:

$$Day = getup \rightarrow breakfast \rightarrow work \rightarrow lunch \rightarrow play \rightarrow dinner \rightarrow tv \rightarrow gotobed \rightarrow STOP$$

As a side note, we can now better define SKIP. we can consider SKIP to be defined as engaging in a special event, which we may notate as ✓, that means that the process is finished with what it is doing, and then behaving like STOP. That is,

$$SKIP = \checkmark \rightarrow STOP$$

The events that processes engage in can also be used to describe communication in terms of reading or writing, with the use of channels. In CSP, given a process, P, a channel, c, a subset of our alphabet Σ, called A, and Processes P(x) defined for any event in A, the process

$$P = c?x : A \rightarrow P(x)$$

is a process which "reads" some event α restricted to subset A from some channel c, and then behaves like P(α). We can also remove the restriction of communicating *only* elements of A, by simply removing it, obtaining the notation:

$$P = c?x \rightarrow P(x)$$

Similarly, to "write" any arbitrary event over some channel c, we simply adjust our notation:

$$P = c!x \rightarrow P$$

2.1.3 Simple Recursion

The concept of prefixing allows us to place several events in sequence to describe some process' behavior. But what if we wanted to create a recursive process? The prefixing operator also allows for this to happen fairly easily. Given the event, α from our alphabet Σ, we can define a process,

$$P = \alpha \rightarrow P$$

That is, we can define a process that engages in the event α and then continues to behave like the process P. Thus the process P is a process which engages in events α infinitely.

2.1.4 External and Internal Choice

The ability to compound processes and events using the prefix operator is a very necessary part of CSP, but it can only get us so far. We still do not have a method of choosing between several

different communication events. This method is given to us as the operator $\Box$, known as external choice. Given two processes P and Q, perhaps defined to engage in events α, $\beta \in \Sigma$ as:

$$P = \alpha \rightarrow STOP$$

$$Q = \beta \rightarrow STOP$$

then the external choice between these two processes is notated as:

$$P \,\Box\, Q$$

This notation means that the process described by $P \,\Box\, Q$ will offer to engage on the first events of P or Q, namely α or β. More specifically, this process is obliged to engage on one of these events, whichever one the environment is also willing to engage on. This form of choice is often called deterministic choice, as we will see the second form of choice represents a nondeterministic version. If we define a process that incorporates internal choice between P and Q, written

$$P \sqcap Q$$

then we are defining a process that will engage on α or β, *or possibly neither of them.* In this case, contrary to external choice, the choice of which one to engage on (if at all!) is in the hands of the process itself.

2.1.5 Sequential Composition

It may also be beneficial to simply place processes in sequence with one another, unlike using prefixing. This operator is ;, and combines its argument processes in sequence with one another. That is, if we take our processes P and Q from above, we obtain

$$P; Q$$

known as the sequential composition of processes P and Q. This process will behave like P until P terminates, and then behave like Q until Q terminates, at which point the process will behave like SKIP.

2.1.6 Parallel Composition

In addition to sequential composition, we have another operator that places two processes in parallel with one another. This operator is called synchronous parallel composition, is notated as ||, and is

used as follows, redefining the P and Q processes once again:

$$P = \alpha \rightarrow Q$$

$$Q = ?x : \Sigma \rightarrow x \rightarrow Q$$

Thus the process $P \,||\, Q$ is defined as the process that behaves like P and Q in a parallel orientation. In other words, the process continually communicates α with itself. P will first communicate α since P and Q can certainly agree with each other, then Q will communicate α to P, and so on. However, it should be noted that synchronous parallel composition only works so long as the processes placed in parallel with one another agree on every single event that they perform. In other words, the processes placed in parallel with each other must "synchronize" on all events in the alphabet. For instance, If we redefine P and Q as

$$P = \alpha \rightarrow \delta \rightarrow STOP$$

$$Q = \beta \rightarrow \delta \rightarrow STOP$$

That is, if we define P as engaging in events α and δ, and then behaving like STOP, and if we define Q as engaging in events β and δ, and then behaving like STOP, we now have an event in our alphabet Σ that the two may synchronize on, but there will never be an agreement between the two since they deadlock on their respective first events. The solution to this problem is a specialization of the synchronous parallel composition operator, called alphabetized parallel composition. Again, with the two processes P and Q, this is notated:

$$P_{\{\delta\}}||_{\{\delta\}}Q$$

The bracketed δ indicates the events in our alphabet that the processes on which P and Q will synchronize. Otherwise these processes will run concurrently with each other. For instance, $P_{\{\delta\}}||_{\{\delta\}}Q$ may behave like the process $\alpha \rightarrow \beta \rightarrow \delta \rightarrow STOP$, since the two processes do not agree on their first events, but do agree on their shared event δ and then behave like $STOP$.

2.1.7 Interleaving

In another simpler way to solve this problem of required synchronization, we are given an operator, $|||$. This is the interleaving operator, which allows processes to be placed in a concurrent orientation without the process requiring a defined synchronization set between them. If we consider our P and Q from before, and interleave them:

$$P|||Q$$

then the interleaving of these two processes may behave like so: both engage in events α and β independently (one after the other), then engage in the δ events independently, and then both will reach $STOP$ independently. It should be noted that interleaving processes will alternate between each other's events in some order, whereas parallelized processes will perform all of these events together (in the case of synchronized parallel) or will perform synchronized events either independently or together, depending on the synchronization set defined (in the case of alphabetized parallel).

2.2 occam-π

The occam-π programming language [oK20] is a language maintained by the University of Kent as an extension of the occam language, with additional features influenced and inspired by CSP and π-calculus. This new and updated version of occam-π is specifically a continuation of occam version 2.1. Occam was developed by INMOS, for use on family of computer processors known as transputers. This language, unlike most languages today, have many built-in facilities for concurrent design, like abstract concepts of channels and processes, along with several other tools to prevent race conditions, deadlock, livelock, and other potential concurrent and parallel pitfalls.

The concepts used in ProcessJ are very similar to those used in occam-π, with the main differences being in syntax. ProcessJ is very heavily inspired by Java's syntax, whereas occam-π has an older style. For instance, many occam-π keywords are in an all-uppercase format. This is fairly far from the syntax of ProcessJ, where there is only an implied necessity of the use of camelCase for method names, field names, and other special members of the language.

Where occam-π and ProcessJ are the same is their hierarchical logic order. In other words, both languages offer concurrent and parallel design constructs (like channels, or barriers) at an atomic level, for users to interact with as basic building blocks of process communication and parallel/concurrent program creation. This is unlike most other high-level programming languages where most concurrency and parallelism is achieved using locks and condition variables, or other mutex-based solutions. Below in listing 2.1 is an example of occam-π code that doubles any number fed to it three times.

```
1
2 PROC double (CHAN INT in?, out!)
3   WHILE TRUE
4     INT x:
```

```
5       SEQ
6         in ? x
7         out ! 2*x
8 :
9
10 PROC octuple (CHAN INT in?, out!)
11   CHAN INT a, b:
12   PAR
13     double (in?, a!)
14     double (a?, b!)
15     double (b?, out!)
16 :
```

Listing 2.1: occam-π language example - double and octuple procs.

2.3 JCSP

JCSP [WBM+07] is a Java library written to provide a model of concurrency, as an amalgam of the CSP and π-calculus process calculi. JCSP was initially introduced in 1996 after a WoTUG "Java Threads Workshop", and since then has been developed and built out by numerous University of Kent students. The library itself is touted for its improvements over occam-π. To be more specific through some examples, JCSP allows true aliasing by Java's rules, unlike occam-π, and also lacks the archaic syntax design that occam and occam-π are so deeply coupled with. JCSP is a solid extension of the principles of process oriented design that occam-π was built for, and pushes the idea of process oriented design as a whole into the modern age. By bridging the gap between modern programming languages and time-tested concurrency models, the paradigm shift from simple locks and condition variables to a mathematical model of concurrency and parallelism is much closer than it may seem.

One of the biggest impacts on JCSP in terms of feature choice is that the library itself maps CSP processes to Java threads. This choice is convenient for development, because of the lack of necessity to provide a separate runtime scheduler on top of the already-existing schedule paradigms provided through the JVM. However, this choice drastically limits the scalability of JCSP when compared to the underlying runtime system for ProcessJ, JVMCSP. This is because the JVMCSP runtime system does not map CSP processes to threads, but rather to a higher-level object that is handled

by a hand-written cooperative runtime scheduler. This is the implementation difference between potentially hundreds of thousands of CSP processes, and potentially billions of CSP processes.

This difference between implementations is exemplified in the comparisons made in [SP16], where we can clearly see the performance differences between JCSP and ProcessJ's JVMCSP. To simplify this difference, the time between mere context switches was much lower for JVMCSP than JCSP, specifically by one order of magnitude on a Mac Pro running a quad-core Intel i7 Xeon processor, and two orders of magnitude on twin-sixteen-core AMD Opteron machine. One could easily attribute this (in part) to the difference between context switching in the realm of threads versus context switching in the realm of our smaller, more simplified "threading" system.

2.4 C++CSP2

While there is much to discuss about the pros and cons of JCSP's implementation, the ProcessJ *C++* runtime system would be better compared against a runtime that is also written in C++.

C++CSP2 [Bro07] is a C++-based CSP runtime library written and maintained by the University of Kent. This runtime library can be thought of as a C++ version of JCSP, with a few caveats. The most obvious one being that this is a C++ library, and thus does not suffer the performance penalties that Java incurs by running on a virtual abstraction above the hardware. C++CSP2 however, like its Java counterpart, makes the decision to map its abstract concept of "processes" (in the context of process-oriented design) to the concrete implementation of threads. This is a great benefit because it shifts the burden of scheduling alongside an operating system to the operating system itself. There is no need to attempt to circumvent the operating system scheduler, because the operating system scheduler is *exactly* what we need in order to manage how threads are run and terminated. However, this poses a problem as process-oriented programs scale upwards. Perhaps on some large-scale machines, the idea of having hundreds of thousands of threads running at one time seems like the upper limit of process-oriented design with respect to the current limitations of computer hardware. Once you pass a certain number of threads on a machine, performance begins to suffer greatly, especially as the number of threads grows onward from there. This is what could be considered as the Achilles heel of C++CSP2 versus the ProcessJ C++ runtime system, just like the difference between JCSP and the ProcessJ JVM runtime system. Instead of mapping processes to threads, we map processes to a smaller item that can be scaled over threads before threads must be scaled over a machine. That is, by treating threads as another layer of scheduling, we can achieve parallelism and concurrency that scales far beyond the

limitations of threads themselves.

In the later chapters of this paper, we will look at some tests of the ProcessJ C++ runtime system, and draw similar conclusions to those found in [SP16], as discussed previously in section 2.3.

2.5 MPI

MPI, also known as Message Passing Interface, is a standard of message-passing maintained by the MPI Forum. The goal of MPI is to create a uniform concept of message passing for high-performance parallel computing. This is achieved by standardizing the way messages are transmitted between processing units by requiring both the sending and receiving processing units to cooperate together in their communication. MPI defines several different methods of communication that can be used to assist in the implementation of distributed systems and the efficiency of distributed computing as a whole. The standard itself has had many implementations, perhaps the most popular and widely used of these being the OpenMPI implementation [Pro20].

The main concept by which MPI operates divides a distributed system into two categories of machines: master and slaves. The master processing unit is typically in charge of distributing the data being operated on to the slaves. The slaves then are then responsible for receiving this data, and doing whatever calculations are necessary with the data before returning the result to the master. The slaves may also communicate with each other, along with the master, in order to carry out a particular distributed task that the system must perform.

This poses the problem that MPI has been standardized to solve: how do machines with entirely separated address spaces communicate with each other uniformly? Through the specifications of the MPI standard, a system model has been designed that can send data in serial, whether it be a simple array of primitive values, or a complex data type, over a network. In other words, the MPI standard provides an organized and efficient way to move data from one machine's address space to another, even across several different machine architectures.

2.6 OpenMP

OpenMP, also known as Open Multi-Processing [DM98], is a grouping of libraries, along with some preprocessor and compiler directives, implemented for Fortran as well as C and C++ that are used to assist in the development of parallel computing solutions. In contrast to MPI, which is

geared towards distributed memory models of parallel computing, OpenMP is specifically made to simplify the shared memory model of parallel computing. That is, instead of distributed machines with separate address spaces, OpenMP operates on one machine with one address space.

OpenMP operates using what is called the fork/join model. This means that a program may start with one single thread of execution that collects and prepares data for calculation. The program may then use the facilities exposed to the developer to then fork other processes with multiple copies of the data distributed between them. These other processes will then run on the data given to them, and finally join with the original process after finishing and sending back their constituents of the solution to the original process.

The only problem that is left to the programmer in the case of using OpenMP is the protection of data that is shared between the many processes deployed to solve a problem. This is because compilers that recognize and handle OpenMP implementations do not have constructs in place that are able to handle or eliminate data dependency issues, let alone report them effectively.

2.7 C++ Coroutines

The C++20 standard defines a new set of tools for those looking for an easier way to think about concurrency. This set of tools is included in the coroutines library [CPP20], and outlines the formal syntax and semantics for coroutines within C++. This specification is not finalized yet, as C++20 itself has not been finalized at the time of writing, but we have a decent amount of solid draft work to understand the basic concept behind coroutines, along with their behavior within C++.

A coroutine in C++ is a function that has the ability to be suspended and resumed at a later time in execution. These functions utilize the new operators *co_await*, *co_yield*, or *co_return* to await resumption, yield execution to another function and return a value, or to return a value outright, respectively. Promise objects are utilized internally by coroutines to achieve their suspension and resumption abilities, as well as the method for returning values to the caller of the coroutine. Coroutines are inherently stack-less, and are contextually stored in a heap-allocated object (unless otherwise optimized out). This object is used in lieu of stack allocation, and contains the coroutine's promise object, along with any parameters (whether by-value or by-reference), the point of suspension (to know where to resume if the coroutine is able to do so), and any local variables that are in scope at the time of suspension.

The implications of coroutines are many, but for the purposes of this paper, will be limited to the context of ProcessJ's C++ runtime system. Using coroutines and their native yielding ability would

provide a large benefit over the way yielding and resuming is handled within the ProcessJ runtime, as we would be able to allow the language itself to handle the logic for "process" resumption and suspension as these features in ProcessJ are mapped to them. This also potentially makes short work of any local variables defined within a coroutine, since the scoping and variable declaration are also handled by the language, and do not, for instance, require all variables declared in a process to be static to the run() function itself. By mapping CSP processes to coroutines, the code generated by the ProcessJ Compiler's C++ code generator would be much simpler to understand in the context of a coroutine instead of a bespoke resumable "process." People can easily read up on what a coroutine is once they recognize its structure in the generated code, instead of having to read into internal documentation or source code for the transpiler.

Chapter 3

Runtime System & Code Generation

To achieve the goal of translating ProcessJ files to C++, and later compile these files to actual binary executables, the current ProcessJ compiler needed an additional code generator to produce the C++ code, as well as an actual runtime system to represent the constructs of ProcessJ in C++. In this chapter, some basic information about the implementation itself is given, and these two systems are explored in detail.

3.1 Runtime System Concepts and Background

3.2 Runtime System

The ProcessJ C++ Runtime System is a fully-implemented representation of the constructs defined and provided by ProcessJ. The runtime system itself is comprised of several header files that make up the C++ representations of ProcessJ. These files are all referenced in a source file for when a ProcessJ program is built. The build system *make* is used to automate this build process. This runtime system is structured file-wise as follows:

```
/
├── include
│   ├── runtime
│   │   ├── pj_alt.hpp
│   │   ├── pj_barrier.hpp
│   │   ├── pj_channel.hpp
│   │   ├── pj_channel_type.hpp
│   │   ├── pj_inactive_pool.hpp
│   │   ├── pj_many2many_channel.hpp
│   │   ├── pj_many2one_channel.hpp
│   │   ├── pj_one2many_channel.hpp
│   │   ├── pj_one2one_channel.hpp
│   │   ├── pj_par.hpp
│   │   ├── pj_process.hpp
│   │   ├── pj_protocol.hpp
│   │   ├── pj_record.hpp
│   │   ├── pj_run_queue.hpp
│   │   ├── pj_runtime.hpp
│   │   ├── pj_scheduler.hpp
│   │   ├── pj_timer.hpp
│   │   └── pj_timer_queue.hpp
│   └── utilities
│       ├── delay_queue.hpp
│       ├── pj_utilities.hpp
│       └── rtti.hpp
├── lib
│   └── libprocessj.a (generated when the runtime is built)
├── src
│   ├── runtime
│   │   └── pj_runtime.cpp
│   └── utilities
│       └── pj_utilities.cpp
└── Makefile
```

3.2.1 *pj_scheduler.hpp*

The C++ runtime scheduler, similar to the JVM runtime scheduler, is built on the principle of *non-preemptive cooperative scheduling* [Cis19]. The C++ runtime scheduler is built around a construct called a run queue, which is used as an organizational container for the processes that are currently running in the context of a ProcessJ program. These processes are operated on by the runtime scheduler through interacting with the run queue, which is detailed later in this section. Processes themselves are always aware of the scheduler that they are in through a member pointer to the scheduler, which allows them to insert other processes (in the case of the main process, or any par

blocks that a process may have). The scheduler itself is structured and has members given in this subsection.

A Naïve Non-Preemptive Cooperative Multi-core Scheduler

One of the more interesting features of the C++ runtime scheduler is the addition of a naïve implementation of a multicore scheduler. This implementation itself is not yet finished, and still has a lot of work to be done, but the initial idea, as well as a functional prototype, has been introduced as of the time of writing. This subsection, in addition to a detailed overview of the scheduler, will give an overview of the features added to the runtime scheduler to enable proper multicore behavior. In addition, code to exemplify the functionality as well as the performance of said implementation will follow in Chapter 5, along with a description of future work needed on the implementation in Chapter 7.

Listing 3.1 is the beginning of the header file defining the *pj_scheduler* class. A scheduler instance is created in the *main*() function of a ProcessJ program, and at the end of the program's runtime is then destructed according to the constructor(s) and destructor given.

```
1  #ifndef PJ_SCHEDULER_HPP
2  #define PJ_SCHEDULER_HPP
3
4  #include <runtime/pj_inactive_pool.hpp>
5
6  #include <runtime/pj_run_queue.hpp>
7  #include <runtime/pj_process.hpp>
8  #include <runtime/pj_timer.hpp>
9
10 #include <mutex>
11 #include <string>
12
13 #include <sched.h>
14 #include <pthread.h>
15 #include <sys/types.h>
16
17 namespace pj_runtime
18 {
19     class pj_scheduler
```

```
20      {
21
22      public:
23          pj_inactive_pool ip;
24
25          // default ctor with 1 scheduler on cpu 0
26          pj_scheduler()
27          : cpu(0), cpus(std::thread::hardware_concurrency())
28          {
29
30          }
31
32          // specialized ctor for use when multiple schedulers
33          // are desired: cpu indicates the cpu the scheduler thread
34          // is isolated to
35          pj_scheduler(uint32_t cpu)
36          : cpu(cpu), cpus(std::thread::hardware_concurrency())
37          {
38
39          }
40
41          // clean up the scheduler thread and the timer queue
42          ~pj_scheduler()
43          {
44              if(this->sched_thread.joinable())
45              {
46                  this->sched_thread.join();
47              }
48
49              /* timer queue should only be killed once */
50              tq.kill();
51
52              std::cerr << "[Scheduler " << cpu << "] Total Context Switches
    : "
53                        << context_switches
54                        << "\n[Scheduler " << cpu << "] Max RunQueue Size: "
```

```
                    << max_rq_size
                    << std::endl;
        }

        // insert a process into the scheduler's run queue
        void insert(pj_process* p)
        {
            std::lock_guard<std::mutex> lk(mutex);
            rq.insert(p);
        }

        // insert a timer into the scheduler's timer queue
        void insert(pj_timer* t)
        {
            std::lock_guard<std::mutex> lk(mutex);
            tq.insert(t);
        }

        // start the scheduler
        void start()
        {
            /* only need to start the timer queue once */
            tq.start();
            this->sched_thread = std::thread(&pj_scheduler::run, this);

            if(this->sched_thread.joinable())
            {
                this->sched_thread.join();
            }
        }

        ...

    protected:
        // query the scheduler's size
        int size()
```

```
91          {
92              std::lock_guard<std::mutex> lk(mutex);
93              return rq.size();
94          }
95
96          // increment context switches
97          void inc_context_switches()
98          {
99              std::lock_guard<std::mutex> lk(mutex);
100             context_switches++;
101         }
102
103         // increment rq max
104         void inc_max_rq_size(size_t size)
105         {
106             std::lock_guard<std::mutex> lk(mutex);
107             if(size > max_rq_size)
108             {
109                 max_rq_size = size;
110             }
111         }
112
113     ...
114
115     };
116 }
117
118 #endif
```

Listing 3.1: The *pj_scheduler* class definition.

Listing 3.2 shows the private data members of the *pj_scheduler* class. We can see both the run queue (rq) and timer queue (tq), which are both used to store processes that have yet to run again, and for timers that will be used by processes, respectively. We also see each instance has a pair of mutexes for guarding different critical sections of the scheduler's behavior and I/O calls, as well as some other members that assist in calculating stats about the program's execution. One thing to note is that each instance of a scheduler object has a thread object, which is the thread

of execution for the scheduler itself. When the scheduler is started, this thread is handed the code that handles the scheduler's behavior and the thread is then executed. The code for the scheduler itself is given in listing 3.3.

```
private:
    pj_timer_queue tq;
    pj_run_queue   rq;

    std::mutex mutex;
    std::mutex iomutex;

    uint64_t start_time       = 0;
    int32_t  context_switches = 0;
    size_t   max_rq_size      = 0;

    std::thread sched_thread;
    uint32_t             cpu;
    uint32_t            cpus;
```

Listing 3.2: The *pj_scheduler* class definition – private data members.

In Listing 3.3 we can see how the scheduler works, noting that it is indeed a near-exact port of the JVM runtime scheduler [Shr16]. In short, the scheduler will continuously remove processes from the run queue, and will hand over control to the *run*() method of each process, as long as they are ready to be run. Otherwise, it will place the process back into the queue, and move onto the next process. If the process has terminated execution, then the process will be destroyed appropriately, instead of being placed back into the run queue.

```
// scheduler behavior
void run(void)
{
    // isolate the scheduler thread to its cpu
    this->isolate_thread();

    // while we have more processes to run
    while(rq.size() > 0)
    {
        // grab max run queue size for stats
```

```
        if(static_cast<size_t>(rq.size()) > max_rq_size)
        {
            max_rq_size = rq.size();
        }

        // grab our next process
        pj_process* p = rq.next();

        // if it's ready to be run
        if(p->is_ready())
        {
            // run it
            p->run();
            // we switched contexts
            context_switches++;
            // if it's not terminated yet
            if(!p->is_terminated())
            {
                // insert the process back into the run queue
                rq.insert(p);
            }
            else
            {
                // finalize the process
                p->finalize();
                // delete the process
                delete p;
            }
        }
        else
        {
            // if it's not ready, just place it back in the queue
            rq.insert(p);
        }
    }
```

```
}
```

Listing 3.3: The *pj_scheduler* class definition – *run*() member function.

Note line 5 of listing 3.3: a call to a member function *isolate_thread*(). This function reassigns the scheduler's thread of execution to a single core of a multicore CPU. While this is not very useful in the case of single core scheduling, it is much more effective in the case of multicore scheduling when we want to ensure that one scheduler is running on one core at any given moment – thus allowing for guaranteed parallel behavior. This function is given in listing 3.4.

```
// isolate the thread of execution for the scheduler behavior to a
// cpu, defined by the cpu variable.
// ---
// this only currently works on posix-compliant machines
void isolate_thread(void)
{
    std::unique_lock<std::mutex> lock(this->iomutex, std::defer_lock);

    // grab thread id
    std::thread::id th_id = this->sched_thread.get_id();

    lock.lock();
    pj_logger::log("isolating thread ", th_id, " to cpu ", cpu);
    lock.unlock();

  // current/new cpu sets to indicate what cpus the thread will
  // be run on and what cpus we want the thread to be run on
    cpu_set_t cur_set;
    cpu_set_t new_set;
    pthread_t    p_th;
    uint32_t        i;

  // zero out the new cpu set before we set it correctly
    CPU_ZERO(&new_set);

  // error sanity check on cpu count argument
    if(!this->cpus)
```

```
28      {
29          lock.lock();
30          std::cerr << "error: hardware_concurrency not set/determinable\n";
31          lock.unlock();
32          abort();
33      }
34
35  // array of integers to track current/new cpu set
36      uint8_t arr_cur_set[this->cpus];
37      uint8_t arr_new_set[this->cpus];
38
39  // zero out both
40      for(i = 0; i < this->cpus; ++i)
41      {
42          arr_cur_set[i] = 0;
43          arr_new_set[i] = 0;
44      }
45
46  // grab native thread handle for setting cpu set
47      p_th = this->sched_thread.native_handle();
48      lock.lock();
49      pj_logger::log("the native_handle is ", p_th);
50      lock.unlock();
51
52  // error condition on not getting thread back
53      if(!p_th)
54      {
55          lock.lock();
56          std::cerr << "error: native_handle() returned null\n";
57          lock.unlock();
58          abort();
59      }
60
61      lock.lock();
62      pj_logger::log("getting thread cpu_set...");
63      lock.unlock();
```

```
64
65  // zero out the current cpu set
66    CPU_ZERO(&cur_set);
67
68  // get the current cpu set for the thread
69    if(pthread_getaffinity_np(p_th,
70                              sizeof(cpu_set_t),
71                              &cur_set))
72    {
73        lock.lock();
74        perror("pthread_getaffinity_np");
75        lock.unlock();
76        abort();
77    }
78
79  // print out which cpus this thread will run on
80    lock.lock();
81    for(i = 0; i < this->cpus; ++i)
82    {
83        if(CPU_ISSET(i, &cur_set))
84        {
85            pj_logger::log("cpu ", i, " is in thread ", th_id, "'s current
    cpu set");
86        }
87    }
88
89    pj_logger::log("now setting thread ", th_id, "'s cpu_set to ", cpu);
90    lock.unlock();
91
92  // set the new cpu set to contain only the cpu this scheduler
93  // sould run on
94    CPU_SET(cpu, &new_set);
95    arr_new_set[cpu] = 1;
96
97    lock.lock();
98    pj_logger::log("new cpu_set is:");
```

```
99      for(i = 0; i < cpus; ++i)
100     {
101         std::cout << static_cast<uint32_t>(arr_new_set[i]) << " ";
102     }
103     pj_logger::log("which implies:");
104     for(i = 0; i < cpus; ++i)
105     {
106         if(CPU_ISSET(i, &new_set))
107         {
108             pj_logger::log(i);
109         }
110     }
111     std::cout << std::endl;
112     lock.unlock();
113
114   // if we don't set the new cpu set for the thread correctly,
115   // error out
116     if(pthread_setaffinity_np(p_th,
117                               sizeof(cpu_set_t),
118                               &new_set))
119     {
120         lock.lock();
121         perror("pthread_setaffinity_np");
122         lock.unlock();
123         abort();
124     }
125
126     lock.lock();
127     pj_logger::log("verifying thread ", th_id, "'s cpu_set...");
128     lock.unlock();
129
130   // double-check our new cpu set is what we want it to be
131   // as a sanity check
132     CPU_ZERO(&cur_set);
133     if(pthread_getaffinity_np(p_th,
134                               sizeof(cpu_set_t),
```

```
135                                 &cur_set))
136     {
137         lock.lock();
138         perror("pthread_getaffinity_np");
139         lock.unlock();
140         abort();
141     }
142 
143     lock.lock();
144     for(i = 0; i < cpus; ++i)
145     {
146         if(CPU_ISSET(i, &cur_set))
147         {
148             pj_logger::log("cpu ", i, " is in new current cpu set");
149             arr_cur_set[i] = 1;
150         }
151     }
152 
153   // sanity check on whether or not the cpu set was modified correctly
154     for(i = 0; i < cpus; ++i)
155     {
156         if(arr_cur_set[i] != arr_new_set[i])
157         {
158             std::cerr << "error: cpu " << i << " is in thread " << th_id
159                       << "'s cpu_set\n";
160             lock.unlock();
161             abort();
162         }
163     }
164     lock.unlock();
165 
166     lock.lock();
167     pj_logger::log("thread ", th_id, "'s cpu_set successfully modified\n")
    ;
168     lock.unlock();
```

```
169 }
```

Listing 3.4: The *pj_scheduler* class definition – *isolate_thread*() member function.

The only current downside to utilizing *isolate_thread*() this way is that the implementation is coupled to the POSIX thread standard, used on several UNIX-like operating systems. It is untested (and probably does not work altogether) on Windows. The resolution of this problem will be left as future work for the time being.

3.2.2 *pj_process.hpp*

Listing 3.5 shows us the contents of *pj_process.hpp*. That is, this shows us the runtime representation of a process. A process is the method by which every invocation or process is executed. Each process has a *run*() method that processes will overload with their own code by extending the *pj_process* class. Additionally, a process may overload *finalize*() in the event that there are barriers or par blocks that require a process to resign from or decrement the counter of, respectively. We return to these two cases in Section 3.2.6 and 3.2.3.

```
1  #ifndef PJ_PROCESS_HPP
2  #define PJ_PROCESS_HPP
3
4  #include <iostream>
5  #include <ostream>
6  #include <mutex>
7
8  #include <sys/types.h>
9
10 namespace pj_runtime
11 {
12     class pj_process
13     {
14     public:
15         pj_process()
16         {
17
18         }
19
```

```
20          virtual ~pj_process()
21          {
22
23          }
24
25          // ready flag getter
26          bool is_ready()
27          {
28              return ready;
29          }
30
31          // ready flag setter
32          void set_ready()
33          {
34              std::unique_lock<std::mutex> lock(this->mtx);
35              if(!ready)
36              {
37                  ready = true;
38              }
39          }
40
41          // ready flag setter
42          void set_not_ready()
43          {
44              std::unique_lock<std::mutex> lock(this->mtx);
45              if(ready)
46              {
47                  ready = false;
48              }
49          }
50
51          // terminated flag setter
52          void terminate()
53          {
54              terminated = true;
55          }
```

```
56
57          // terminated flag getter
58          bool is_terminated()
59          {
60              return terminated;
61          }
62
63          // process behavior member function, overloaded by any process
64          virtual void run()
65          {
66
67          }
68
69          // process finalization member function, overloaded by any
70          // process resigning from a barrier or decrementing a par counter
71          virtual void finalize()
72          {
73
74          }
75
76          // run_label setter
77          virtual void set_label(uint32_t label)
78          {
79              run_label = label;
80          }
81
82          // run_label getter
83          virtual uint32_t get_label()
84          {
85              return run_label;
86          }
87
88          friend std::ostream& operator<<(std::ostream& o, pj_process& p)
89          {
90              return o << "base process operator<< called (nothing
    overwritten)";
```

```
91          }
92
93      protected:
94          std::mutex mtx;
95
96      private:
97          uint32_t run_label      = 0;
98          bool     ready        = true;
99          bool     terminated = false;
100     };
101 }
102
103 #endif
```

Listing 3.5: The *pj_process* class definition.

3.2.3 *pj_par.hpp*

The code given in listing 3.6 describes the runtime representation of a par block in ProcessJ. A par block in ProcessJ describes a block of code to be executed in parallel. For example, listing 3.7 below gives an example of a simple ProcessJ file that utilizes a par block.

```
1  #ifndef PJ_PAR_HPP
2  #define PJ_PAR_HPP
3
4  #include <runtime/pj_process.hpp>
5
6  #include <sys/types.h>
7
8  namespace pj_runtime
9  {
10     class pj_par
11     {
12     public:
13         pj_par() = delete;
14
15         // initialize with the process declaring a par
```

```
16          // (p) and the number of processes in the par
17          pj_par(int process_count, pj_process* p)
18          : process_count(process_count)
19          {
20              this->process = p;
21          }
22
23          ~pj_par() = default;
24
25          // proc count setter
26          void set_process_count(int32_t count)
27          {
28              this->process_count = count;
29          }
30
31          // decrement the number of processes still executing
32          // that were invoked from inside the par. if there
33          // are no more, then set the process declaring the par
34          // ready
35          void decrement()
36          {
37              std::lock_guard<std::mutex> lock(this->mtx);
38
39              this->process_count--;
40
41              if(this->process_count == 0)
42              {
43                  this->process->set_ready();
44              }
45          }
46
47          // returns true if there are processes in the par,
48          // setting the par process not ready, and false otherwise
49          bool should_yield()
50          {
51              std::lock_guard<std::mutex> lock(this->mtx);
```

```
52
53              if(this->process_count == 0)
54              {
55                  return false;
56              }
57
58              this->process->set_not_ready();
59              return true;
60          }
61
62      protected:
63      private:
64          pj_process* process;
65          int32_t process_count;
66          std::mutex mtx;
67      };
68  }
69
70  #endif
```

Listing 3.6: The *pj_par* class definition.

```
1  // import io library
2  import std.io;
3
4  // foo process
5  public void foo() { println("Hello from foo!"); }
6
7  // bar process
8  public void bar() { println("Hello from bar!"); }
9
10 // main process
11 public void main(string[] args) {
12   // run the following in parallel
13   par {
14     println("Print from Hello");
15     foo();
```

```
16      bar();
17    }
18  }
```

Listing 3.7: *Hello.pj*.

On line 8 of listing 3.7, we see the declaration of a par block for calls to *println*() (a member of the io library), and the two methods *foo*() and *bar*() defined on lines 3 and 5, respectively. Note that these are non-yielding procedures, and are not treated as processes themselves, but are treated as static functions by the C++ runtime system, and handed off to an anonymous procedure that will then invoke the static function. The runtime handles this as follows: the main process creates a par object with an argument of 3 passed to it, indicating the number of processes in the par. Then, the main process creates three anonymous non-yielding processes that invoke *println*() with the specified string (on line 9), *foo*() with no arguments (line 10), and *bar*() with no arguments (line 10). These processes are then scheduled one after the other. Upon their termination, *finalize*() is invoked, which has been overloaded to include a call to *decrement*() on the par that the process has been declared inside. This way, the par object will know that the processes declared inside of it have completed, and the process declaring the par can then resume execution.

3.2.4 *pj_channel.hpp*

The file *pj_channel.hpp*, shown entirely below in listing 3.8, describes the representation and behavior of a channel in ProcessJ. A channel is a construct used as a medium by which processes can communicate with each other. Anything can be sent through these channels to other processes, including processes themselves. The concept of mobile processes utilizes these channels to accomplish its goal, though process mobility itself is still left for future work.

```
1  #ifndef PJ_CHANNEL_HPP
2  #define PJ_CHANNEL_HPP
3
4  #include <runtime/pj_process.hpp>
5  #include <runtime/pj_channel_type.hpp>
6
7  #include <mutex>
8
```

```
namespace pj_runtime
{
  class pj_channel
  {
  public:
    // default constructor with default NONE channel type
    pj_channel()
    : type(pj_channel_types::NONE)
    {

    }

    // specialized constructor with a type as an argument
    pj_channel(pj_channel_types t)
    : type(pj_channel_type(t))
    {

    }

    // specialized constructor with a type as an argument
    pj_channel(pj_channel_type t)
    : type(t)
    {

    }

    // move constructor
        pj_channel(pj_channel&& other)
        {
        this->reader = other.reader;
        this->writer = other.writer;
        }

    // move assignment operator
        pj_channel& operator=(pj_channel&& other)
        {
```

```
45          this->reader = other.reader;
46          this->writer = other.writer;
47          return *this;
48          }
49
50      virtual ~pj_channel()
51      {
52              reader = nullptr;
53              writer = nullptr;
54      }
55
56      // returns the type of the channel
57          pj_channel_type get_channel_type()
58          {
59            return type;
60          }
61
62          // if we already have a writer, register the alt process p
63          // as the reader, and return the writer pointer
64          pj_process* alt_get_writer(pj_process* p)
65          {
66              std::lock_guard<std::mutex> lock(this->mtx);
67              if(!writer)
68              {
69                  reader = p;
70              }
71
72              return writer;
73          }
74
75      // sets the reader to the argument process and returns
76      // a pointer to the writing process
77          pj_process* set_reader_get_writer(pj_process* p)
78          {
79              reader = p;
80              return writer;
```

```
81        }
82
83        // following four member functions are overloaded by child classes
    ,
84        // namely one2one, many2one, one2many, and many2many channels
85        virtual bool claim_read(pj_process* p)
86        {
87          return false;
88        }
89
90        virtual void unclaim_read(pj_process* p)
91        {
92
93        }
94
95        virtual bool claim_write(pj_process* p)
96        {
97          return false;
98        }
99
100       virtual void unclaim_write(pj_process* p)
101       {
102
103       }
104
105       friend std::ostream& operator<<(std::ostream& o, pj_channel& c)
106       {
107         o << c.type;
108         return o;
109       }
110
111 protected:
112   pj_channel_type type;
113   std::mutex mtx;
114       pj_process* writer = nullptr;
115       pj_process* reader = nullptr;
```

```
116   };
117 }
118
119 #endif
```

Listing 3.8: The *pj_channel* class definition.

A channel in ProcessJ's C++ runtime system can have one of four specific channel types, which are all described by the *pj_channel_type* class. Specifically, these four types of channels are one-to-one, one-to-many, many-to-one, and many-to-many. To accomplish the implementation of these specialized channels, the *pj_channel* class is extended by four new classes, each used to describe and facilitate these different behavior specifications. These four specialized channel types, as well as their C++ class representation will be described in the following sections.

pj_channel_type.hpp

The class *pj_channel_type* defines an internal bookkeeping class for the C++ runtime system to identify what kind of channel is being used or referenced from the viewpoint of a generic *pj_channel* object instead of a specialized one. This class defines an enumeration of the different channel types, and includes a string representation of the channel type as well.

```
#ifndef PJ_CHANNEL_TYPE_HPP
#define PJ_CHANNEL_TYPE_HPP
#include <string>

namespace pj_runtime
{
  // enumeration of channel types
  enum pj_channel_types
  {
    NONE,
    ONE2ONE,
    ONE2MANY,
    MANY2ONE,
    MANY2MANY
  };

  class pj_channel_type
  {
  public:
    // default constructor with default NONE type
    pj_channel_type()
    : type(pj_channel_types::NONE)
    {

    }

    // specialized constructor with channel type as argument
    pj_channel_type(pj_channel_types t)
    : type(t)
    {
      switch(t)
      {
        case pj_channel_types::ONE2ONE:
        type_str = "one-to-one channel for use by one writer and one
   reader";
        break;
```

```
36        case pj_channel_types::ONE2MANY:
37        type_str = "one-to-many channel for use by one writer and many
    readers";
38        break;
39        case pj_channel_types::MANY2ONE:
40        type_str = "many-to-one channel for use by many writers and one
    reader";
41        break;
42        case pj_channel_types::MANY2MANY:
43        type_str = "many-to-many channel for use by many writers and many
    readers";
44        break;
45        default:
46        type_str = "bad channel type";
47        break;
48      }
49    }
50
51    virtual ~pj_channel_type()
52    {
53
54    }
55
56    // channel type getter function
57    pj_channel_types get_type()
58    {
59      return type;
60    }
61
62    // channel type as-a-string getter function
63    std::string get_type_string()
64    {
65      return type_str;
66    }
67
68    friend std::ostream& operator<<(std::ostream& o, pj_channel_type& t)
```

```
69      {
70        o << t.type_str;
71        return o;
72      }
73
74    private:
75      pj_channel_types type;
76      std::string type_str = "";
77    };
78  }
79
80  #endif
```

Listing 3.9: The *pj_channel_type* class definition.

pj_one2one_channel.hpp

The *pj_one2one_channel* class is the C++ runtime representation of a one-to-one channel in ProcessJ. This type of channel has **one** reader, and **one** writer. The file *pj_one2one_channel.hpp* is given in Listing 3.10, and a brief description of behavior is given afterwards.

```
1  #ifndef PJ_ONE2ONE_CHANNEL_HPP
2  #define PJ_ONE2ONE_CHANNEL_HPP
3
4  #include <runtime/pj_channel.hpp>
5
6  namespace pj_runtime
7  {
8    // template-generic channel class definition
9    template <typename T>
10   class pj_one2one_channel : public pj_runtime::pj_channel
11   {
12   public:
13     // default constructor with default NONE type
14     pj_one2one_channel()
15     {
16       this->type = pj_channel_types::ONE2ONE;
```

```
17      }
18
19      ~pj_one2one_channel()
20      {
21
22      }
23
24      // write data to the channel, and become not ready
25      // to run until the reader wakes us (the writer) up
26      // again
27      void write(pj_process* p, T data)
28      {
29        std::lock_guard<std::mutex> lock(this->mtx);
30        this->data = data;
31        writer = p;
32        writer->set_not_ready();
33        if(reader)
34        {
35          reader->set_ready();
36        }
37      }
38
39      // read from the channel, set the writer ready
40      // again, and clear our reader/writer pointers
41      T read(pj_process* p)
42      {
43        std::lock_guard<std::mutex> lock(this->mtx);
44        writer->set_ready();
45        writer = nullptr;
46        reader = nullptr;
47        return this->data;
48      }
49
50      // returns true if there is a writer registered
51      // on the channel already, and false otherwise,
52      // also setting the reader not ready until the
```

```
53      // writer comes along and wakes us (the reader) up
54      bool is_ready_to_read(pj_process* p)
55      {
56        std::lock_guard<std::mutex> lock(this->mtx);
57        if(writer)
58        {
59          return true;
60        }
61        else
62        {
63          reader = p;
64          reader->set_not_ready();
65        }
66        return false;
67      }
68
69      // a one2one channel is always ready to be written to
70      bool is_ready_to_write(pj_process* p)
71      {
72        return true;
73      }
74
75      // rendezvous code for before a channel read
76      T pre_read_rendezvous(pj_process* p)
77      {
78        T data = this->data;
79        this->data = reinterpret_cast<T>(0);
80        return data;
81      }
82
83      // rendezvous code for after a channel read
84      void post_read_rendezvous(pj_process* p)
85      {
86        writer->set_ready();
87        writer = nullptr;
88        reader = nullptr;
```

```
89      }
90
91    protected:
92    T data;
93    };
94  }
95
96  #endif
```

Listing 3.10: The *pj_one2one_channel* class definition.

There are two general initial cases for this channel type: either a process claims the channel to write something to a potential reading process, or a process claims the channel to read something from a potential writing process. In the first of these cases, the writing process calls *is_ready_to_write*(), which always returns true in the case of a one-to-one channel. This fact may seem surprising, but is in fact verified by the findings of [PC19]. To put it simply, a one-to-one channel is *always* ready to be written to whenever a writing process gets to a write statement, because there is only ever one writer using this channel. A write statement would only be delayed if a reader is not present to read the data from the channel, and in that case, the reader is responsible for setting the writer ready after reading.

The writer then calls *write*() to place data into the channel. Afterwards, the writer will be set as not ready and wait for the reading process to read the data out of the channel. Then, a reading process will (hopefully!) claim the channel and check if it is ready to be read from by calling *is_ready_to_read*(), which returns true if there is already a writing process, or sets the reading process not ready and returns false if a writing process is not present. If this returns true, the reading process then calls *read*() which returns the data from the channel, and then both reading and writing processes are set as ready, and the program continues.

In the latter case where a process wants to read before a writer has written something to the channel, the reverse of the previous case happens. In other words, the reading process checks if the channel is ready, and becomes not ready as it waits for a writing process. A writing process claims the channel, writes some data to it, and waits for the reader to receive the data. The reading process then reads from the channel, the writing process is set ready, and execution of the program continues.

pj_one2many_channel.hpp

The *pj_one2many_channel* class is the C++ runtime representation of a one-to-many channel in ProcessJ. This type of channel has one writer and many readers. The file *pj_one2many_channel.hpp* is given in listing 3.11, and a brief description of behavior is given afterwards.

```
#ifndef PJ_ONE2MANY_CHANNEL_HPP
#define PJ_ONE2MANY_CHANNEL_HPP

#include <runtime/pj_channel.hpp>
#include <runtime/pj_one2one_channel.hpp>

#include <queue>

namespace pj_runtime
{
  template <typename T>
  class pj_one2many_channel : public pj_runtime::pj_one2one_channel<T>
  {
  public:
    // default constructor with null read_claim pointer
    pj_one2many_channel()
    : read_claim(nullptr)
    {
      this->type = pj_channel_types::ONE2MANY;
    }

    ~pj_one2many_channel()
    {
      read_claim = nullptr;
    }

    // claim the read end of the channel, optionally being set
        // not ready to run and enqueued on the read queue if
        // there is already an active reading process
    bool claim_read(pj_process* p)
    {
```

```
32      std::lock_guard<std::mutex> lock(this->mtx);
33      if(!read_claim || read_claim == p)
34      {
35        read_claim = p;
36        return true;
37      }
38      else
39      {
40        p->set_not_ready();
41        read_queue.push(p);
42      }
43
44      return false;
45    }
46
47    // deregister from the read end of the channel, optionally
48        // grab the next reader from the queue and set them ready
49    void unclaim_read()
50    {
51      std::lock_guard<std::mutex> lock(this->mtx);
52      if(read_queue.empty())
53      {
54        read_claim = nullptr;
55      }
56      else
57      {
58        pj_process* p = read_queue.front();
59        read_queue.pop();
60        read_claim = p;
61        p->set_ready();
62      }
63    }
64
65  protected:
66    pj_process* read_claim = nullptr;
67    std::queue<pj_process*> read_queue;
```

```
68    };
69 }
70
71 #endif
```

Listing 3.11: The *pj_one2many_channel* class definition.

For a one-to-many channel, the way that a process accomplishes writing to a channel is the same as a one-to-one channel. The difference comes from having many readers at the other end of the channel. This channel type, instead of utilizing a single reader, has instead a *std* :: *queue* of potential readers, stored in the order in which they committed to a read on the channel itself. A potential reading process calls *claim_read*(), which will set them as the current active reader if there are no readers in the channel's read queue, and add them to the queue if otherwise. Then, the current active reader will call *read*() to extract the written data, or wait until there is data to be read. Then, assuming the current reading process has completed a *read*() call, the current reader then calls *unclaim_read*() which removes them from being the current reading process, and dequeues the next reading process from the queue as the new current reading process.

To consider the correctness of this channel type in a naïve manner, consider a program with one writing process, and, arbitrarily, four reading processes. If the writing process arrives at the channel write first, it will of course write its data to the channel, and yield to wait for a reader to come by and read the data. Otherwise, if one or more of the reading processes get to their channel read first, they will first check if the channel read head can be claimed via a call to *claim_read*(), in which case they will be set as the current reading process, or in the event that the read head has already been claimed, will be set not ready to run and placed in the potential reader queue. Once a reading process is woken up because something is ready to be read from the channel, the channel then calls *is_ready_to_read*(), which returns true if there is a writer present, and false if there is not, also setting the reading process back to not being ready. This specific sequence of calls has a very important feature to it: by checking if the read head has been claimed, and *then* checking if the channel is ready to be read from, or in other words, if the channel already has a writer committed on the writing end of the channel, then there is no case where a reader will be able to invoke *read*() without a writer being present. This aforementioned sequence of function calls is a safety net to be sure that there is always data to be read when a *read*() call comes around.

pj_many2one_channel.hpp

The *pj_many2one_channel* class is the C++ runtime representation of a one-to-many channel in ProcessJ. This type of channel has many writers and one reader. The file *pj_many2one_channel.hpp* is given in listing 3.12, and a brief description of behavior is given afterwards.

```
#ifndef PJ_MANY2ONE_CHANNEL_HPP
#define PJ_MANY2ONE_CHANNEL_HPP

#include <runtime/pj_one2one_channel.hpp>

#include <queue>

namespace pj_runtime
{
  template <typename T>
  class pj_many2one_channel : public pj_runtime::pj_one2one_channel<T>
  {
  public:
    pj_many2one_channel()
    {

    }

    ~pj_many2one_channel()
    {

    }

    // claim the write end of the channel, optionally being set
        // not ready to run and enqueued on the write queue if
        // there is already an active writing process
    bool claim_write(pj_process* p)
    {
      std::lock_guard<std::mutex> lock(this->mtx);
      if(!write_claim || write_claim == p)
      {
```

```
32          write_claim = p;
33          return true;
34        }
35        else
36        {
37          p->set_not_ready();
38          write_queue.push(p);
39        }
40
41        return false;
42      }
43
44      // deregister from the write end of the channel, optionally
45          // grab the next writer from the queue and set them ready
46      void unclaim_write()
47      {
48        std::lock_guard<std::mutex> lock(this->mtx);
49        if(write_queue.empty())
50        {
51          write_claim = nullptr;
52        }
53        else
54        {
55          pj_process* p = write_queue.front();
56          write_queue.pop();
57          write_claim = p;
58          p->set_ready();
59        }
60      }
61
62    protected:
63      pj_process* write_claim = nullptr;
64      std::queue<pj_process*> write_queue;
65    };
66  }
67
```

```
68 #endif
```

Listing 3.12: The *pj_many2one_channel* class definition.

The way that a process writes over a many-to-one channel is similar to the way that a process reads from a one-to-many channel. A many-to-one channel utilizes a *std* :: *queue* on the writer end to handle having several writing processes. Like the reading method for a one-to-many channel, a process that wants to write over a many-to-one channel will call *claim_write*() and be set as the current writing process, or be placed into the write queue. Then, the current writing process will write its data to the channel, and wait for a process to read the data. Once a reading process has taken the data from the channel, the writer will be released from the channel by calling *unclaim_write*() and a new current writing process will be dequeued from the write queue. This type of channel benefits from the same guarding sequence of function calls as the one-to-many channel, but in a different manner. A writer will always yield after a write to wait for a reader to read the data out of the channel, which will then wake up the writing process again. The writing process will then relinquish the writing head and set the next queued writer as the writing process, and so on.

pj_many2many_channel.hpp

The *pj_many2many_channel* class is the C++ runtime representation of a many-to-many channel in ProcessJ. This type of channel has many writers and many readers, making it a sort of amalgam of the many-to-one and one-to-many channels. The file *pj_many2many_channel.hpp* is given in listing 3.13, and a brief description of behavior is given afterwards.

```
1  #ifndef PJ_MANY2MANY_CHANNEL_HPP
2  #define PJ_MANY2MANY_CHANNEL_HPP
3
4  #include <runtime/pj_one2one_channel.hpp>
5
6  #include <queue>
7
8  namespace pj_runtime
9  {
10     template <typename T>
11     class pj_many2many_channel : public pj_runtime::pj_one2one_channel<T>
```

```
12      {
13      public:
14          pj_many2many_channel()
15          {
16
17          }
18
19          ~pj_many2many_channel()
20          {
21
22          }
23
24          // claim the read end of the channel, optionally being set
25          // not ready to run and enqueued on the read queue if
26          // there is already an active reading process
27          bool claim_read(pj_process* p)
28          {
29              std::lock_guard<std::mutex> lock(this->mtx);
30              if(!read_claim || read_claim == p)
31              {
32                  read_claim = p;
33                  return true;
34              }
35              else
36              {
37                  p->set_not_ready();
38                  read_queue.push(p);
39              }
40
41              return false;
42          }
43
44          // deregister from the read end of the channel, optionally
45          // grab the next reader from the queue and set them ready
46          void unclaim_read()
47          {
```

```
48              std::lock_guard<std::mutex> lock(this->mtx);
49              if(read_queue.empty())
50              {
51                  read_claim = nullptr;
52              }
53              else
54              {
55                  pj_process* p = read_queue.front();
56                  read_queue.pop();
57                  read_claim = p;
58                  p->set_ready();
59              }
60          }
61
62          // claim the write end of the channel, optionally being set
63          // not ready to run and enqueued on the write queue if
64          // there is already an active writing process
65          bool claim_write(pj_process* p)
66          {
67              std::lock_guard<std::mutex> lock(this->mtx);
68              if(!write_claim || write_claim == p)
69              {
70                  write_claim = p;
71                  return true;
72              }
73              else
74              {
75                  p->set_not_ready();
76                  write_queue.push(p);
77              }
78
79              return false;
80          }
81
82          // deregister from the write end of the channel, optionally
83          // grab the next writer from the queue and set them ready
```

```
84          void unclaim_write()
85          {
86              std::lock_guard<std::mutex> lock(this->mtx);
87              if(write_queue.empty())
88              {
89                  write_claim = nullptr;
90              }
91              else
92              {
93                  pj_process* p = write_queue.front();
94                  write_queue.pop();
95                  write_claim = p;
96                  p->set_ready();
97              }
98          }
99
100     protected:
101         pj_process* read_claim = nullptr;
102         pj_process* write_claim = nullptr;
103         std::queue<pj_process*> read_queue;
104         std::queue<pj_process*> write_queue;
105     };
106 }
107
108 #endif
```

Listing 3.13: The *pj_many2many_channel* class definition.

As stated previously, the behavior of a many-to-many channel is a combination of the many-to-one and one-to-many channels. This channel type utilizes both a read queue and a write queue. As reading processes or writing processes attempt to read or write to the channel, the first for each will be placed as the current active reading or writing processes, or enqueued in the read or write queue, respectively. This is accomplished by the readers calling *claim_read*(), or the writers calling *claim_write*(). Then, the current reading process or writing process will read or write to the channel, and call *unclaim_read*() or *unclaim_write*() to remove themselves from the channel and allow the next dequeued reader or writer to continue on with communication.

3.2.5 *pj_alt.hpp*

The file *pj_alt.hpp*, shown in part in listing 3.14, describes the representation of an alt construct in ProcessJ. An alt itself is an object of *pj_alt* type that is maintained by the process in which the alt was declared. This object stores all of the important information and other elements that an alt needs. Line 18 defines the type of alt guards, which is a std::variant of the three possible constructs: a string (specifically the SKIP string defined on line 25), a channel read end, or a timeout. lines 34 through 36 define the elements of an alt, including the process declaring the alt, a *std* :: *vector* of alt guards, and a *std* :: *vector* of Boolean guards. The *pj_alt* class defines operations specific to an alt, namely *set_guards*(), *enable*(), and *disable*(). The first of these member functions has an obvious function: it is responsible for setting the guards of the alt based on the guards that are locally declared and initialized from the alternating process. The last other two member functions are touched on in more detail later on in this section.

```
1  #ifndef PJ_ALT_HPP
2  #define PJ_ALT_HPP
3
4  #include <runtime/pj_process.hpp>
5  #include <runtime/pj_channel.hpp>
6  #include <utilities/rtti.hpp>
7
8  #include <iostream>
9  #include <vector>
10 #include <variant>
11 #include <string>
12
13 #include <sys/types.h>
14
15 namespace pj_runtime
16 {
17     // typedef for the kinds of alt guards we can expect
18     typedef std::variant<std::string, pj_channel*, pj_timer*>
       pj_alt_guard_type;
19
20     // pj_alt class def
21     class pj_alt
```

```
22      {
23      public:
24          // c++ implementation of SKIP
25          inline static const std::string SKIP = "skip";
26
27          pj_alt() = delete;
28
29      // specialized constructor with count of processes in the alt
30      // and the process in which the alt was declared itself as arguments
31          pj_alt(uint64_t count, pj_process* p)
32          {
33              this->process = p;
34          }
35
36          ~pj_alt()
37          {
38              pj_logger::log("pj_alt destructor called");
39          }
40
41          // set the boolean and object guards of the alt
42          bool set_guards(std::vector<bool>            b_guards,
43                          std::vector<pj_alt_guard_type> guards)
44          {
45              this->b_guards = b_guards;
46              this->guards   = guards;
47
48              for(uint32_t i = 0; i < this->b_guards.size(); ++i)
49              {
50                  if(b_guards[i])
51                  {
52                      return true;
53                  }
54              }
55
56              return false;
57          }
```

```
58
59          ...
60
61      private:
62          pj_process* process;
63          std::vector<pj_alt_guard_type> guards;
64          std::vector<bool> b_guards;
65      };
66  }
67
68  #endif
```

Listing 3.14: The *pj_alt* class definition.

In the ProcessJ program in Listing 3.15, we see the processes *writer*1, *writer*2, and *reader* defined, with *reader* alternating on a sequence of guards, with the first two guards being channel reads from *writer*1 and *writer*2, and the default guard being skip. The code that is generated for an alt statement uses two specific member functions to operate, namely *enable*() and *disable*(). These functions are detailed in listings 3.16 and 3.17, and work with each other to ensure that the correct guard is selected out of the alt.

```
1   // import io library
2     import std.io;
3
4     // writer process
5     public void writer1(chan<int>.write out) {
6         out.write(42);
7     }
8
9     // writer process
10    public void writer2(chan<int>.write out) {
11        out.write(43);
12    }
13
14    // reader process, with alt between reading from in1 or in2,
15    // optionally skipping if none are ready to be read from
16    public void reader(chan<int>.read in1, chan<int>.read in2) {
```

```
17          int v;
18          alt {
19              v = in1.read() : { println("Got " + v + " from writer 1."); }
20              v = in2.read() : { println("Got " + v + " from writer 2."); }
21              skip : { println("in skip"); }
22          }
23      }
24
25      // main process, with 2 channels (c1 and c2) declared,
26      // and the two writers/a reader set in parallel with each other
27      public void main(string[] args) {
28          chan<int> c1, c2;
29          par {
30              writer1(c1.write);
31              writer2(c2.write);
32              reader(c1.read, c2.read);
33          }
34      }
```

Listing 3.15: *alttest.pj*.

In listing 3.16 and 3.17, we can see that *enable*() simply iterates through the alt guards in sequence, selecting the first one whose preguard is ready. Note that the boolean guard (being accessed on line 8) is checked first. This is because a boolean preguard is considered before the actual alt guard itself is checked. If a boolean preguard is set as true, then that alt is considered. Otherwise, the alt guard is skipped over using a continue statement on line 10. In the case that a boolean preguard is true, and thus the alt guard must be considered, "readiness" is defined in terms of the alt guard we are looking at. A channel must have a valid writing process set as its writer, that is still waiting to write some data to the channel. For a timeout statement to be "ready," the timer simply must have timed out. A skip guard is always ready, and is thus treated as a "default" case. Following the code for *enable*() and *disable*(), the alternating process first invokes *enable*() to select a guard that is ready. In this particular implementation of alts, we have generalized the alt to behave as a form of prioritized alt. That is, once the guard that is ready is selected (noting that the guards are visited and considered in order, from top to bottom), the alternating process then yields, and invokes *disable*() when it is woken up. In the case of a non-prioritized alt, *disable*()

then moves back up from the selected alt (the index of which is returned from *enable*()) to the first alt guard, making sure that the alternating process is not still registered on some channel from another alt guard, or that a timer has been expired and is not still running for the alternating process. However, in the case that the keyword *pri* is used, indicating that a prioritized alt is indeed wanted, then *disable*() ensures that the first guard that is ready is chosen, regardless of the returned index from *enable*(), all while still ensuring that appropriate deregistration from any timers or channels happens.

```
// first pass _down_ the alt guards
int32_t enable(void)
{
    // iterate through all the object guards
    for(uint32_t i = 0; i < this->guards.size(); ++i)
    {
        // if the boolean guard is false, skip this guard entirely
        if(!this->b_guards[i])
        {
            continue;
        }

        // if it's a string (a SKIP)
        if(std::holds_alternative<std::string>(this->guards[i]))
        {
            // check if it's literally SKIP
            if(std::get<std::string>(this->guards[i]) == SKIP)
            {
                // set the process ready and return the index of our selected alt
                this->process->set_ready();
                return static_cast<int32_t>(i);
            }
        }
        // if it's a channel read
        else if(std::holds_alternative<pj_runtime::pj_channel*>(this->guards[i]))
        {
```

```
27            // if the channel has a writer already (is ready to be read
    from)
28            if(std::get<pj_runtime::pj_channel*>(this->guards[i])->
    alt_get_writer(this->process) != nullptr)
29            {
30                // set the process ready and return the index of our
    selected alt
31                this->process->set_ready();
32                return static_cast<int32_t>(i);
33            }
34        }
35        // if it's a timeout statement
36        else if(std::holds_alternative<pj_runtime::pj_timer*>(this->guards
    [i]))
37        {
38            // if the timer has timed out
39            if(std::get<pj_runtime::pj_timer*>(this->guards[i])->
    get_real_delay() <= std::chrono::time_point_cast<std::chrono::
    milliseconds>(std::chrono::system_clock::now()))
40            {
41                // set the process ready, expire the timer, and return the
     index
42                this->process->set_ready();
43                std::get<pj_runtime::pj_timer*>(this->guards[i])->expire()
    ;
44                return static_cast<int32_t>(i);
45            }
46            else
47            {
48                // start the timer if not started already
49                std::get<pj_runtime::pj_timer*>(this->guards[i])->start();
50            }
51        }
52    }
53
54    // default case -- no guards ready
```

```
55      return -1;
56  }
```

Listing 3.16: The *pj_alt* class definition – *enable*() member function.

```
1   // second pass _up_ the alt guards
2   int32_t disable(int32_t i)
3   {
4       int32_t selected = -1;
5
6       // if we previously did not have a ready guard, start from the very
        bottom
7       if(i == -1)
8       {
9           i = this->guards.size() - 1;
10      }
11
12      // iterate back up the guards
13      for(int32_t j = i; j >= 0; --j)
14      {
15          // if the boolean flag for this guard is false, skip it
16          if(!this->b_guards[i])
17          {
18              continue;
19          }
20
21          // if it's a SKIP, select it
22          if(std::holds_alternative<std::string>(this->guards[j]))
23          {
24              if(std::get<std::string>(this->guards[j]) == SKIP)
25              {
26                  selected = j;
27              }
28          }
29          // if it's a channel read and we have a writer, select it
30          else if(std::holds_alternative<pj_runtime::pj_channel*>(this->
        guards[j]))
```

```
31          {
32              if(std::get<pj_runtime::pj_channel*>(this->guards[j])->
    set_reader_get_writer(nullptr) != nullptr)
33              {
34                  selected = j;
35              }
36          }
37          // if it's a timer that has expired (timed out), select it,
    otherwise kill it
38          else if(std::holds_alternative<pj_runtime::pj_timer*>(this->guards
    [j]))
39          {
40              if(std::get<pj_runtime::pj_timer*>(this->guards[j])->expired()
    )
41              {
42                  selected = j;
43              }
44              else
45              {
46                  std::get<pj_runtime::pj_timer*>(this->guards[j])->kill();
47              }
48          }
49      }
50      return selected;
51  }
```

Listing 3.17: The *pj_alt* class definition – *disable*() member function.

3.2.6 *pj_barrier.hpp*

The file *pj_barrier.hpp*, shown entirely below in listing 3.18, describes the representation of a barrier in ProcessJ. A barrier is a construct used to synchronize one or more processes before proceeding onward. The barrier accomplishes this in the C++ runtime by using the member functions *enroll*(), *resign*(), and *sync*().

```
1 #ifndef PJ_BARRIER_HPP
2 #define PJ_BARRIER_HPP
```

```
3
4  #include <runtime/pj_process.hpp>
5
6  #include <iostream>
7  #include <vector>
8  #include <queue>
9
10 #include <sys/types.h>
11
12 namespace pj_runtime
13 {
14     class pj_barrier
15     {
16     public:
17         // vector of synced processes on this barrier
18         std::vector<pj_process*> synced;
19         // number enrolled on the barrier
20         uint32_t enrolled = 0;
21
22         // init with one enrolled
23         pj_barrier()
24         : enrolled(1)
25         {
26
27         }
28
29         ~pj_barrier()
30         {
31
32         }
33
34         // enroll n - 1, where n is the number of processes that wish to
35         // synchronize on the barrier
36         void enroll(uint32_t proc_count)
37         {
38             std::lock_guard<std::mutex> lock(this->mtx);
```

```
39              this->enrolled += (proc_count - 1);
40          }
41
42          // resign from the barrier by decr'ing enrolled
43          void resign()
44          {
45              std::lock_guard<std::mutex> lock(this->mtx);
46              if(this->enrolled > 1)
47              {
48                  --this->enrolled;
49              }
50          }
51
52          // synchronize on the barrier. if every process is synchronized,
53          // then set them ready and clear the vector.
54          void sync(pj_process* process)
55          {
56              std::lock_guard<std::mutex> lock(this->mtx);
57              process->set_not_ready();
58              this->synced.push_back(process);
59              if(this->synced.size() == enrolled)
60              {
61                  for(uint32_t i = 0; i < this->synced.size(); ++i)
62                  {
63                      this->synced[i]->set_ready();
64                  }
65                  synced.clear();
66              }
67          }
68
69      protected:
70      private:
71          std::mutex mtx;
72      };
73  }
74
```

```
75 #endif
```

Listing 3.18: The *pj_barrier* class definition.

As we can see, the *enroll*() function, defined on line 36, takes an argument of the number of processes that we are enrolling on the barrier in question. When *sync*() is invoked by some process enrolled on the barrier, that process is set not ready and placed in a *std* :: *vector* of processes that have synchronized on the barrier. In other words, all of the processes that call *sync*() will wait until *every process* that is enrolled on the barrier has synchronized with it. Then, lastly, the final process to enroll on a barrier triggers the resumption of all processes synchronized thus far, with the condition on line 59 that allows the processes to be set ready by iterating through the synced processes and setting them all ready, as seen on lines 61 thru 65. Once these processes finish running, the scheduler will invoke their *finalize*() functions. Inside of the *finalize*() function for each of the processes that enroll on a barrier, a call to *resign*() is made to remove the process from the barrier, such that the barrier can keep track of the processes still potentially using the barrier. Note that many processes can enroll on a barrier at several points in time during the execution of a program, and thus the barrier must know how many enrolled processes to expect before setting them all ready upon receiving the correct number of *sync*() invocations.

Consider the ProcessJ code in listing 3.19:

```
1  // import io library
2  import std.io;
3
4  // foo process, sets a variable and syncs on a barrier
5  public void foo(barrier b) {
6      int a = 5;
7      b.sync();
8  }
9
10 // bar process, syncs on a barrier
11 public void bar(barrier b) {
12     b.sync();
13 }
14
15 // main process, declares barrier b and enrolls 2 processes
16 // in parallel on it
```

```
17 public void main(string args[]) {
18     barrier b;
19     par enroll b {
20         foo(b);
21         bar(b);
22     }
23 }
```

Listing 3.19: *barrierEx.pj*.

In this example, we see that a par block on line 14 is created and enrolls in a barrier, *b*, declared on line 13. The main process declares a *pj_barrier* and a *pj_par*, with both objects expecting two processes to use them. The par block is created with 2 processes inside of it, namely *foo* and *bar*, and enrolls on the barrier *b*. In this case, the processes themselves are created, their *finalize*() function is overloaded to include a *resign*() call for barrier *b*, and they are then scheduled. Each process runs until the *sync*() invocation is made, in which the process that gets to its *sync*() call first will wait for the other to also invoke *sync*(). Then the two are set ready by the barrier, their execution is terminated, *finalize*() is invoked on them, and they are deregistered from the barrier. The usefulness of this barrier object is clearly seen by considering the possibility of processes needing to step forward with each other. In other words, the necessity of barriers is clearly seen in the case of a program that is written to solve a problem where processes must communicate with one another to solve their next chunk of the problem set given to them. We will see a larger example of this in some of the larger tests written in chapter **??**, where the benefits of a barrier in concurrent programming and ProcessJ as a whole will be made more clear.

3.2.7 *pj_record.hpp*

A *pj_record* in the C++ runtime system is the internal representation of a ProcessJ record. A record in ProcessJ is a user-defined data type, similar to a *struct* in C++. In fact, a record in ProcessJ is directly mapped to a *struct* in C++, and as such, records in ProcessJ follow the inheritance structure of C++ *struct*s.

```
1 #ifndef PJ_RECORD_HPP
2 #define PJ_RECORD_HPP
3
4 namespace pj_runtime
```

```
{
    // every pj_record inherits from this struct
  struct pj_record
  {

  };
}

#endif
```

Listing 3.20: The *pj_record* class definition.

3.2.8 *pj_protocol.hpp*

The *pj_protocol* class is used to define a ProcessJ data structure, similar to a record, but with the added feature of enclosed tags that can be used to describe different forms of data. The *pj_protocol* class definition describe both the base struct that protocols in ProcessJ are mapped to in C++, along with the *pj_protocol_case* struct that is used inside a protocol to define the specific data elements within.

```
#ifndef PJ_PROTOCOL_HPP
#define PJ_PROTOCOL_HPP

namespace pj_runtime
{
    // Every protocol case inherits from this struct
    struct pj_protocol_case
    {
    public:
        // tag value set in the code generator to be able
        // to switch on protocol cases
        int tag;
    };
}

#endif
```

Listing 3.21: The *pj_protocol* and *pj_protocol_case* struct definitions.

This definition, given in listing 3.21, shows exactly what was previously stated: protocols and protocol cases in ProcessJ are mapped simply to structs. A *pj_protocol* is used to define a protocol in ProcessJ, and it contains one or more *pj_protocol_case*s that are members. The base struct allows for inheritance in ProcessJ to simply be mapped to the inheritance system of C++, again similar to the *pj_record* class.

3.2.9 *pj_run_queue.hpp*

The *pj_run_queue* class is an internal part of the scheduler for the C++ runtime that manages the organization of processes, running or not, within the runtime system. In the current system structure, each scheduler instance shares one run queue to pull from and run as needed. In future revisions to the scheduler system itself, each scheduler instance will have its own run queue that holds any processes that the scheduler instance is tasked with running.

```
1  #ifndef PJ_RUN_QUEUE_HPP
2  #define PJ_RUN_QUEUE_HPP
3
4  #include <runtime/pj_process.hpp>
5
6  #include <queue>
7  #include <thread>
8  #include <mutex>
9  #include <iostream>
10
11 #include <sys/types.h>
12
13 namespace pj_runtime
14 {
15     class pj_run_queue
16     {
17     public:
18
19         pj_run_queue()
20         {
21
22         }
```

```
23
24          ~pj_run_queue()
25          {
26
27          }
28
29          // insert a process into the run queue
30          void insert(pj_process* p)
31          {
32              std::lock_guard<std::mutex> lk(rq_mutex);
33              queue.push(p);
34          }
35
36          // grab the next process from the run queue
37          pj_process* next()
38          {
39              std::lock_guard<std::mutex> lk(rq_mutex);
40              pj_process* next = queue.front();
41              queue.pop();
42              return next;
43          }
44
45          // query the size of the run queue
46          size_t size()
47          {
48              std::lock_guard<std::mutex> lk(rq_mutex);
49              size_t size = queue.size();
50              return size;
51          }
52
53      protected:
54          std::queue<pj_process*> queue;
55
56      private:
57          std::mutex rq_mutex;
58      };
```

```
59 }
60
61 #endif
```

Listing 3.22: The *pj_run_queue* class definition.

As shown in listing 3.22, the run queue is simply a wrapper class around a *std* :: *queue* of *pj_process*es, with a mutex to guard against concurrent access. It is a simple wrapper that allows for processes to be inserted using *insert*(), or removed by using *next*(). These functions, along with *size*() are used by *pj_scheduler* instances' *run*() functions to assist in scheduling and running processes in a ProcessJ program.

3.2.10 *pj_runtime.hpp*

The *pj_runtime.hpp* header is simply a header used that includes all of the runtime elements discussed in this chapter. This header is included at the top of each program generated by the ProcessJ compiler so that all the appropriate elements can be included in a program via one single include guard.

```
1  #ifndef PJ_RUNTIME_HPP
2  #define PJ_RUNTIME_HPP
3
4  /* include guards for the runtime types */
5  #include <runtime/pj_logger.hpp>
6  #include <runtime/pj_process.hpp>
7  #include <runtime/pj_timer.hpp>
8  #include <runtime/pj_timer_queue.hpp>
9  #include <runtime/pj_run_queue.hpp>
10 #include <runtime/pj_inactive_pool.hpp>
11 #include <runtime/pj_scheduler.hpp>
12 #include <runtime/pj_channel.hpp>
13 #include <runtime/pj_one2one_channel.hpp>
14 #include <runtime/pj_one2many_channel.hpp>
15 #include <runtime/pj_many2one_channel.hpp>
16 #include <runtime/pj_many2many_channel.hpp>
17 #include <runtime/pj_barrier.hpp>
18 #include <runtime/pj_alt.hpp>
19 #include <runtime/pj_record.hpp>
```

```
#include <runtime/pj_protocol.hpp>
#include <runtime/pj_par.hpp>
#include <runtime/pj_array.hpp>
#include <runtime/pj_string.hpp>

#endif
```

Listing 3.23: The *pj_runtime.hpp* header file.

3.2.11 *pj_timer.hpp*

The *pj_timer* class defines the internal representation of a ProcessJ Timer object in C++. Timers are quite simple to understand: they are timers that can be set to a specific timeout value (given in milliseconds by default), and can also be attached to processes waiting for them to timeout.

```
#ifndef PJ_TIMER_HPP
#define PJ_TIMER_HPP

#include <runtime/pj_process.hpp>

#include <iostream>
#include <chrono>
#include <ostream>

namespace pj_runtime
{

    class pj_timer
    {
        friend class pj_timer_queue;
        friend class pj_alt;

    public:
        bool m_started;
        bool m_expired;

        // default ctor, immediate timeout, no process (the kill timer)
```

```
23        pj_timer()
24        : m_started(false),
25          m_expired(false),
26          m_delay(0),
27          m_real_delay(std::chrono::time_point_cast<std::chrono::
    milliseconds>(std::chrono::system_clock::time_point::min())),
28          m_killed(false),
29          m_process(static_cast<pj_process*>(0))
30        {
31
32        }
33
34        // specialized ctor, with specified timeout value
35        pj_timer(long timeout)
36        : m_started(false),
37          m_expired(false),
38          m_delay(timeout),
39          m_real_delay(std::chrono::time_point_cast<std::chrono::
    milliseconds>(std::chrono::system_clock::time_point::min())),
40          m_killed(false),
41          m_process(static_cast<pj_process*>(0))
42        {
43
44        }
45
46        // specialized ctor, specified process and timeout value
47        pj_timer(pj_process* process, long timeout)
48        : m_started(false),
49          m_expired(false),
50          m_delay(timeout),
51          m_real_delay(std::chrono::time_point_cast<std::chrono::
    milliseconds>(std::chrono::system_clock::time_point::min())),
52          m_killed(false),
53          m_process(process)
54        {
55
```

```
56        }
57
58        ~pj_timer() = default;
59
60        // calculate the timeout point and mark the timer as started
61        void start()
62        {
63            m_real_delay = std::chrono::system_clock::time_point(std::
    chrono::milliseconds(pj_timer::read() + this->get_delay()));
64            m_started = true;
65        }
66
67        // timeout value setter
68        void timeout(long timeout)
69        {
70            m_delay = timeout;
71        }
72
73        // read the current time
74        static long read()
75        {
76            auto now = std::chrono::system_clock::now();
77            auto now_ms = std::chrono::time_point_cast<std::chrono::
    milliseconds>(now);
78            auto now_epoch = now_ms.time_since_epoch();
79            auto value = std::chrono::duration_cast<std::chrono::
    milliseconds>(now_epoch);
80            return static_cast<long>(value.count());
81        }
82
83        // kill flag setter
84        void kill()
85        {
86            m_killed = true;
87        }
88
```

```
89          // kill flag getter
90          bool killed()
91          {
92              return m_killed;
93          }
94
95          // expired flag setter
96          void expire()
97          {
98              m_expired = true;
99          }
100
101         // expired flag getter
102         bool expired()
103         {
104             return m_expired;
105         }
106
107         // delay getter
108         long get_delay()
109         {
110             return m_delay;
111         }
112
113         // timer process pointer setter
114         void set_process(pj_process* p)
115         {
116             m_process = p;
117         }
118
119         // timer process pointer getter. returns the process pointer
120         // iff the timer is not killed yet, else a nullptr is returned
121         pj_process* get_process()
122         {
123             return (m_killed) ? static_cast<pj_process*>(0) : m_process;
124         }
```

```
125
126         friend std::ostream& operator<<(std::ostream& o, pj_timer& t)
127         {
128             return o << "Process: " << t.m_process;
129         }
130
131 protected:
132         // _real_ delay value getter for alt
133         std::chrono::system_clock::time_point get_real_delay()
134         {
135             return std::chrono::time_point_cast<std::chrono::milliseconds
    >(m_real_delay);
136         }
137
138     private:
139         long m_delay;
140         std::chrono::system_clock::time_point m_real_delay;
141         long m_timeout;
142         bool m_killed;
143         pj_process* m_process;
144     };
145 }
146
147 #endif
```

Listing 3.24: The *pj_timer* class definition.

In the class definition in listing 3.24, we can see the functions that facilitate a timer being created, used, and cleaned up by a process that declares one. A *pj_timer* is constructed with either empty arguments (implying immediate timeout), a specified timeout value as the argument, or a *pj_process* pointer and a specified timeout value as arguments. Then, a timer can be polled for its timeout point by using *get_delay*(). Along with some other internal functions like *set_process*() and *get_process*(), the timer will timeout and whatever process that is waiting (if any) to resume operation of the process that was waiting for the timer. More examples of this behavior and its many forms will be detailed int section 3.3.

3.2.12 *pj_timer_queue.hpp*

The *pj_timer_queue* class is another internal organization object for use by *pj_scheduler* instances to manage timers as they are made and used by processes within a ProcessJ program runtime. Like the *pj_run_queue* object, the timer queue is an element that facilitates proper behavior and organization of timers, but within its own thread. That is, a timer queue has its own thread of execution separate from a scheduler that facilitates all timer operations while a ProcessJ program is running.

```
#ifndef PJ_TIMER_QUEUE_HPP
#define PJ_TIMER_QUEUE_HPP

#include <runtime/pj_timer.hpp>
#include <utilities/delay_queue.hpp>

#include <thread>
#include <atomic>

namespace pj_runtime
{
    class pj_timer_queue
    {

    friend class pj_timer;

    public:
        // default ctor, exit_value is always set
        pj_timer_queue()
        : exit_value(1)
        {

        }

        // clean up the timer thread, and the kill timer
        // if necessary
        ~pj_timer_queue()
        {
```

```
29              if(timer_thread.joinable())
30              {
31                  timer_thread.join();
32              }
33
34              /* make sure we delete our kill_timer sanely */
35              if(kill_timer)
36              {
37                  delete kill_timer;
38              }
39          }
40
41          // insert a timer into the timer queue
42          void insert(pj_timer* timer)
43          {
44              std::lock_guard<std::mutex> lock(this->mtx);
45              dq.enqueue(timer, timer->get_real_delay());
46          }
47
48          // timer queue behavior
49          void start()
50          {
51              timer_thread = std::thread([this]()
52              {
53                  while(1)
54                  {
55                      // get a timer out of the delay queue
56                      // ---
57                      // this blocks until the timer is ready
58                      // to be removed (the timer has timed out)
59                      pj_timer* timer = dq.dequeue();
60
61                      // expire the timer
62                      timer->expire();
63
64                      // get the timer's process
```

```
65                      pj_process* p = timer->get_process();
66
67                      // check if we can safely exit as a thread
68                      if(!p && exit_value)
69                      {
70                          // this is the kill timer, we're done
71                          return;
72                      }
73
74                      if(p)
75                      {
76                          // wake up the process waiting on the timer
77                          p->set_ready();
78                      }
79
80                      // delete the timer object
81                      delete timer;
82                  }
83              });
84          }
85
86          // kill the timer queue by placing a special timer in,
87          // to let the timer thread know that we're done
88          void kill()
89          {
90              // we're ready to end execution
91              kill_flag.exchange(true);
92
93              // make our kill_timer and place it in the queue
94              kill_timer = new pj_timer();
95
96              // drop the kill timer in so the timer queue knows
97              // it's time to stop
98              this->insert(kill_timer);
99          }
100
```

```
101         // query the size of the timer queue's delay queue
102         size_t size()
103         {
104             return dq.size();
105         }
106
107     private:
108         pj_utilities::delay_queue<pj_timer*> dq;
109         std::thread                timer_thread;
110         std::mutex                           mtx;
111         std::atomic<int32_t> exit_value;
112         std::atomic<bool>    kill_flag;
113
114         pj_timer* kill_timer = nullptr;
115     };
116 }
117
118 #endif
```

Listing 3.25: The *pj_timer_queue* class definition.

As shown in listing 3.25, the timer queue object is a wrapper around a specialized queue called a delay queue, similar in implementation to a Java delay queue. This wrapper has its own thread – as mentioned before, which handles the execution of the timer queue's *run*() function, and a mutex to guard against concurrent access problems. The timer queue has functions similar to the scheduler and run queue, such as *insert*(), that allow timers to be placed in and waited on by the runtime system as a whole. The function *start*() is invoked by the scheduler on startup to initiate the thread of execution for the timer, before moving on to starting the actual ProcessJ program.

When execution of a ProcessJ program has stopped successfully, and it is time for the runtime to begin the process of ending its own execution, the function *kill*() is called by the scheduler to start the process of killing the timer queue. A *std* :: *atomic* value and a special timer called *kill_timer* are used to signal the timer queue that it is time to stop working and clean up for the runtime to gracefully stop. The timer is created, and placed into the timer queue. When the timer queue receives this timer from the delay queue, it checks to see if it is indeed the kill timer, which will be signified by the attached process being null, and the *exit_value* being set appropriately, as seen

on line 72. Then, if this condition is true, the timer queue thread returns and ends its execution, allowing the scheduler and other elements of the runtime system to clean up and finally terminate as a whole.

3.3 Code Generation

In this section, we will examine some of the generated code for the constructs of ProcessJ, and give some explanation as to the reasoning behind it.

3.3.1 Alt

To examine the code generation of an alt, we turn again to the example code in *alttest.pj*. An excerpt of the generated code for the alt used in the reader process can be seen below in Listing 3.26.

```
1  // alt object
2  _ld$alt3 = new pj_runtime::pj_alt(2, this);
3
4  // boolean guards as a vector
5  boolean_guards = { true, true };
6
7  // object guards as a vector
8  object_guards = { _pd$in13, pj_runtime::pj_alt::SKIP };
9
10 // ready flag for the alt, returned by set_guards()
11 alt_ready = _ld$alt3->set_guards(boolean_guards, object_guards);
12
13 // selected value, initialized to 0
14 selected = static_cast<int>(0);
15
16 // local index variable, initialized to 0
17 _ld$index2 = static_cast<int>(0);
18
19 // if the alt guards were not set properly, abort
20 if (!alt_ready) {
21     std::cout << "RuntimeError: One of the boolean pre-guards must be true
   !" << std::endl;
```

```
22      abort();
23  }
24
25  // set the process not ready to be run
26  this->set_not_ready();
27
28  // index of the selected alt guard from the enable()
29  _ld$index2 = _ld$alt3->enable();
30
31  // we want to resume from the next label
32  this->set_label(3);
33  return;
34
35  _proc$reader2$682474708L3:
36
37  // selected contains the guard that was ready on the disable()
38  selected = _ld$alt3->disable(_ld$index2);
39
40  // switch-case statement on the selected guard, and the code
41  // generated for that guard's behavior
42  switch(selected)
43  {
44      case 0:
45        // channel read
46          if (!_pd$in13->is_ready_to_read(this)) {
47              this->set_label(1);
48              return;
49          }
50
51          _proc$reader2$682474708L1:
52          _ld$v1 = _pd$in13->read(this);
53
54          _proc$reader2$682474708L2:
55          io::println("Got ", _ld$v1, " from writer 1.");
56          break;
57      case 1:
```

```
58        // skip case
59          io::println("in skip");
60          break;
61      default:
62          break;
63  }
```

Listing 3.26: *alttest.pj* - generated alt code.

In this example, we see the alt object constructed on line 1. Next, the boolean guards and object guards are constructed as *std* :: *vector*s on lines 2 and 3, respectively. The alt is passed the guards on line 4 with the invocation of *set_guards*(), and we later see *enable*() invoked, which selects the guard that is ready. Then, we see an invocation of *disable*() when the reader process is woken back up. Once all of this has been done, we see the switch-case statement for the *selected* variable, which is returned and set by *disable*() on line 19. This switch case facilitates the actual choice between handling the different alt guards. The code for each alt guard may have been generated, but the selection of the guard and final behavior of the alternating process is dictated by the *enable*() and *disable*() result.

3.3.2 Barrier

Listing 3.27 shows us the program *barrierEx.pj*. This is a simple test program that places two processes, *foo* and *bar*, in parallel with each other, and enrolls them on a barrier *b*. Then, these processes are passed the barrier, and both invoke *sync*() on the barrier, which causes them to wait until they have both reached this *sync*() invocation. This code is shown for *bar* in Listing 3.28.

```
1  import std.*;
2
3  public void foo(barrier b) {
4      int a = 5;
5      b.sync();
6  }
7
8  public void bar(barrier b) {
9      b.sync();
10 }
11 texmaker red underline
```

```
12 public void main(string args[]) {
13     barrier b;
14     par enroll b {
15         foo(b);
16         bar(b);
17     }
18 }
```

Listing 3.27: *barrierEx.pj*.

```
1  virtual void run()
2  {
3      switch (get_label())
4      {
5          case 0: goto _proc$bar11198L0; break;
6          case 1: goto _proc$bar11198L1; break;
7      }
8
9      _proc$bar11198L0:
10     _pd$b2->sync(this);
11     this->set_label(1);
12     return;
13     _proc$bar11198L1:
14     terminate();
15     return;
16 }
```

Listing 3.28: *barrierEx.pj* - generated *bar* :: *run*() method.

The invocation of *sync*() on line 10 places the process in the barrier's sync set, and then sets the process not ready to run. The process yields after this, and does not get set ready to run again until all other processes enrolled on the barrier have also synchronized.

The processes that are within the par block of this test program need to have code generated specially so that they have the ability to resign from the barrier on termination, and decrement the par's process counter. This code is shown below in 3.29 on lines 35-39.

```
1 _ld$_par1 = new pj_runtime::pj_par(2, this);
2 _ld$b2->enroll(2);
```

```
3
4  // local class definition to extend foo, and give it its finalize() code
5  class _proc$foo01198_overload_finalize_0 : public _proc$foo01198
6  {
7  public:
8      _proc$main21169311* parent;
9
10     _proc$foo01198_overload_finalize_0(pj_runtime::pj_scheduler* sched,
       pj_runtime::pj_barrier* b, _proc$main21169311* parent)
11     : _proc$foo01198{sched, b}, parent(parent)
12     {
13     }
14
15     virtual void finalize()
16     {
17         parent->_ld$_par1->decrement();
18         parent->_ld$b2->resign();
19     }
20 };
21
22 this->sched->insert(new _proc$foo01198_overload_finalize_0(this->sched,
       _ld$b2, this));
23
24 // same thing for bar
25 class _proc$bar11198_overload_finalize_1 : public _proc$bar11198
26 {
27 public:
28     _proc$main21169311* parent;
29
30     _proc$bar11198_overload_finalize_1(pj_runtime::pj_scheduler* sched,
       pj_runtime::pj_barrier* b, _proc$main21169311* parent)
31     : _proc$bar11198{sched, b}, parent(parent)
32     {
33     }
34
35     virtual void finalize()
```

```
36      {
37          parent->_ld$_par1->decrement();
38          parent->_ld$b2->resign();
39      }
40  };
41
42  this->sched->insert(new _proc$bar11198_overload_finalize_1(this->sched,
        _ld$b2, this));
43
44  if (_ld$_par1->should_yield()) {
45      this->set_label(1);
46      return;
47  }
48
49  _proc$main21169311L1:
```

Listing 3.29: *barrierEx.pj* - generated barrier code.

The code generated in 3.29 shows the actual construction of the par object, the barrier object, the enrollment of the processes in the par on the barrier, and even the *finalize*() code that is generated in the case of processes needing to decrement the par counter, or resign from a barrier after execution.

It should also be noted that, since this barrier is part of a par block, the code generated in 3.29 also exemplifies the code generated for a par, where anonymous classes (or in this case, extensions of previously-declared classes for processes) are generated on-the-fly, given a *finalize*() that includes the par block and barriers (if any) the processes are enrolled on, and then instantiated to be passed to the scheduler.

3.3.3 Channels

This subsection will delve into the four different types of channels, in a grouping of non-shared and shared channels.

One2One

For a one-to-one channel in ProcessJ, the test program *channelWR.pj* (shown in Listing 3.30) gives a basic example of the code generated for a channel read, and a channel write.

```
import std.*;

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

public void bar(chan<int>.read in) {
    int x = in.read();
    println("x: " + x);
}

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

Listing 3.30: *channelWR.pj*.

```
virtual void run()
{
  // switch-case block for process resumption points
    switch (get_label())
    {
        case 0: goto _proc$foo01171377L0; break;
        case 1: goto _proc$foo01171377L1; break;
    }

  // first label
    _proc$foo01171377L0:

  // write the literal value 4 to the out channel
    _pd$out1->write(this, 4);
    // set our next resumption label
    this->set_label(1);
```

```
    // yield
    return;

  // second label
    _proc$foo01171377L1:
    // let the scheduler know we're finished running
    terminate();
    // process end
    return;
}
```

Listing 3.31: *channelWR.pj* - $foo :: run()$ method.

```
virtual void run()
{
  // switch-case block for process resumption points
    switch (get_label())
    {
        case 0: goto _proc$bar11171222L0; break;
        case 1: goto _proc$bar11171222L1; break;
        case 2: goto _proc$bar11171222L2; break;
    }

  // first label
    _proc$bar11171222L0:

  // initialize x local declaration
    _ld$x1 = static_cast<int>(0);
    // if we're not ready to read, then yield until we're woken up
    if (!_pd$in2->is_ready_to_read(this)) {
        this->set_label(1);
        return;
    }

    _proc$bar11171222L1:
    // read from the in channel into x
    _ld$x1 = _pd$in2->read(this);
```

```
25
26      _proc$bar11171222L2:
27      // invoke println()
28      io::println("x: ", _ld$x1);
29      // let the scheduler know we're done running
30      terminate();
31      // process end
32      return;
33  }
```

Listing 3.32: *channelWR.pj* - *bar* :: *run*() method.

The code in Listing 3.31 shows us the code generated by the *foo* process, namely a write call on line 11, which sets the process as not ready until the channel has been read from by *bar*. Likewise, in Listing 3.32, we see the resulting code generated for *bar* and its channel read. In this code, line 13 invokes *is_ready_to_read*(), which returns true if there is a writer already present on the channel, and false otherwise, setting the reader of the channel to the process that called it. Then, the invocation of *read*() on line 19 actually returns the data in the channel to the reader, and wakes up the writer process to continue onward when its turn to run comes around.

Shared (One2Many/Many2One/Many2Many)

To exemplify the code generated by the remaining three channels, we can consider the code generated for the test program *sharedchan.pj*, shown in Listing 3.33.

```
1  // import io library
2  import std.io;
3
4  // reader process that reads from the in channel and prints
5  // what it receives forever
6  public void reader(int id, shared chan<int>.read in) {
7    while (true) {
8      int v; v = in.read();
9      println(id + ": " + v);
10   }
11 }
12
13 // writer process that writes the values 0...n to the channel out
```

```
14 public void writer(shared chan<int>.write out) {
15   int v = 0;
16   while (true) {
17     out.write(v);
18     v = v + 1;
19   }
20 }
21
22 // main process
23 public void main(string args[]) {
24   // shared channel
25   shared chan<int> c;
26   // localdecl
27   int a = 129;
28   // run 3 readers and 2 writers in parallel
29   par {
30     reader(1, c.read);
31     reader(2, c.read);
32     reader(3, c.read);
33     writer(c.write);
34     writer(c.write);
35   }
36 }
```

Listing 3.33: *sharedchan.pj*.

```
1  // init local v
2  _ld$v2 = static_cast<int>(0);
3
4  // if we are in the read queue (not immediately the reader), wait
5  // until we are woken up to read
6  if(!_pd$in2->claim_read(this))
7  {
8      set_label(1);
9      return;
10 }
11
```

```
_proc$reader01099237691L1:
// if we're not ready to read, yield until we are woken up to read
if(!_pd$in2->is_ready_to_read(this))
{
    set_label(2);
    return;
}

_proc$reader01099237691L2:
// read from in into v
_ld$v2 = _pd$in2->read(this);
set_label(3);

// release our claim on the read head of the channel
_pd$in2->unclaim_read();
return;
_proc$reader01099237691L3:
// print our value
io::println(": ", _ld$v2);
```

Listing 3.34: *sharedchan.pj* - shared channel read code.

```
// if we are in the write queue (not immediately the writer), wait
// until we are woken up to write
if(!_pd$out3->claim_write(this))
{
    set_label(1);
    return;
}
_proc$writer11171377L1:
// write local declaration v to the channel out
_pd$out3->write(this, _ld$v3);
set_label(2);
return;

_proc$writer11171377L2:
// release our claim on the write head of the channel
```

```
16 _pd$out3->unclaim_write();
17
18 // increment v
19 _ld$v3 = _ld$v3 + 1;
```

Listing 3.35: *sharedchan.pj* shared channel write code.

In Listing 3.34, we see the code generated for a read call. This is the code that is generated for any read on a one-to-many or a many-to-many channel. The invocation on line 3 to *claim_read*() sets the reader of the channel, and optionally enqueues the process attempting to claim the read end of the channel into the channel's read queue. The process is then set not ready to run, and yields until it reaches the front of the read queue. Then the read call behaves like a normal one-to-one channel read, the only difference being the additional call to *unclaim_read*() on line 20, which deregisters the current active reader from the channel, and wakes up the next reader in the read queue, if there are any present.

In Listing 3.35, the code generated for a write call can be seen. This code in particular is generated for all calls to *write*() to a many-to-one or many-to-many channel. The invocation to *write*() is guarded by an invocation to *claim_write*(), which behaves the same as *claim_read*() for writers, and uses a write queue instead. It is followed by a call to *unclaim_write*(), which does the same thing as *unclaim_read*() as well, but for writers instead.

3.3.4 Processes

While there is only one real kind of process in the runtime, there are technically three different ways to generate them. The first of which is a normal process, defined in the global scope as a C++ class. This kind of process is shown in generated form in Listing 3.36. The second of these is the anonymous process, which is defined within the scope of an enclosing par block (in the case of a single line of code), which is defined and scheduled on-the-fly. This process is shown in ProcessJ form and in generated form in Listing 3.37. The third and final is an extension of the normal process, which is redefined with an overloaded *finalize*() function, in the case of a normal process being invoked from within a par, a par that enrolls on a barrier, or being invoked with a barrier passed as an argument. Listing 3.38 shows some code that will generate this sort of process, and shows this type of process as generated by the compiler.

```
// processes are classes that extend pj_runtime::pj_process
class _proc$reader01099237691 : public pj_runtime::pj_process
{

public:
  // local declarations
    bool _ld$foreverLoop0$1;
    int _ld$v2;

    _proc$reader01099237691() = delete;
    _proc$reader01099237691(pj_runtime::pj_scheduler* sched,
  int _pd$id1,
  pj_runtime::pj_many2many_channel<int32_t>* _pd$in2)
    {
        this->sched = sched;
        this->_pd$id1 = _pd$id1;
        this->_pd$in2 = _pd$in2;
    }

    virtual ~_proc$reader01099237691() = default;

    virtual void run()
    {
        // switch-case block to handle process resumption points
        switch (get_label())
        {
            case 0: goto _proc$reader01099237691L0; break;
            case 1: goto _proc$reader01099237691L1; break;
            case 2: goto _proc$reader01099237691L2; break;
            case 3: goto _proc$reader01099237691L3; break;
        }

        _proc$reader01099237691L0:
    // initialize variables
        _ld$foreverLoop0$1 = static_cast<bool>(0);
        _ld$foreverLoop0$1 = true;
```

```
37          // infinite loop begin
38          while (_ld$foreverLoop0$1) {
39            // init local v
40              _ld$v2 = static_cast<int>(0);
41
42        // if we are in the read queue (not immediately the reader),
43        // then yield until we're woken up to read
44              if(!_pd$in2->claim_read(this))
45              {
46                  set_label(1);
47                  return;
48              }
49
50              _proc$reader01099237691L1:
51              // if the channel isn't ready to be read from, yield until
52              // we're woken up to read
53              if(!_pd$in2->is_ready_to_read(this))
54              {
55                  set_label(2);
56                  return;
57              }
58
59              _proc$reader01099237691L2:
60              // read from in into v
61              _ld$v2 = _pd$in2->read(this);
62              set_label(3);
63        // release our claim on the read head
64              _pd$in2->unclaim_read();
65              return;
66              _proc$reader01099237691L3:
67              // print v
68              io::println(": ", _ld$v2);
69              (void)0;
70            }
71            terminate();
72            return;
```

```
73      }
74
75  protected:
76      // formal variables
77      int _pd$id1;
78      pj_runtime::pj_many2many_channel<int32_t>* _pd$in2;
79  private:
80      pj_runtime::pj_scheduler* sched;
81  };
```

Listing 3.36: A standard process definition.

```
1   // main process
2   public void main(string[] args) {
3     // non-shared channel b
4     chan<int> b;
5     // local ints
6     int q, r;
7     // write and read in parallel
8       par {
9         c.write(q)
10        r = c.read();
11      }
12  }
13
14  ...
15
16  // anonymous process that wraps the 'c.write(q)' invocation above
17  class _proc$Anonymous250 : public pj_runtime::pj_process
18  {
19  public:
20      _proc$main151169311* parent;
21
22      _proc$Anonymous250() = delete;
23      _proc$Anonymous250(pj_runtime::pj_scheduler* sched,
24    pj_runtime::pj_array<std::string>* _pd$args63,
25    _proc$main151169311* parent)
```

```
26      {
27          this->sched = sched;
28          this->parent = parent;
29          this->_pd$args63 = _pd$args63;
30      }
31
32      virtual ~_proc$Anonymous250() = default;
33
34      virtual void run()
35      {
36          switch (get_label())
37          {
38              case 0: goto _proc$Anonymous250L0; break;
39              case 1: goto _proc$Anonymous250L1; break;
40          }
41
42          _proc$Anonymous250L0:
43      // anonymous process wraps around this channel write
44          parent->_ld$b646->write(this, parent->_ld$q672);
45          this->set_label(1);
46          return;
47
48          _proc$Anonymous250L1:
49          terminate();
50      }
51      // finalize() is overloaded with a decrement of the par counter
52      // because this is an anonymous process within a par block
53      virtual void finalize()
54      {
55          parent->_ld$_par9->decrement();
56
57      }
58  protected:
59      pj_runtime::pj_array<std::string>* _pd$args63;
60
61  private:
```

```
62      pj_runtime::pj_scheduler* sched;
63  };
64
65  this->sched->insert(new _proc$Anonymous250(this->sched, _pd$args63, this))
        ;
```

Listing 3.37: An 'anonymous' process definition.

```
1   public void main(string args[]) {
2     shared chan<int> c;
3     int a = 129;
4     par {
5       // these readers are overloaded below
6       reader(1, c.read);
7       reader(2, c.read);
8       reader(3, c.read);
9       writer(c.write);
10      writer(c.write);
11    }
12  }
13
14  ...
15
16  // overloaded class for reader proc. since reader() is a defined
17  // process, all we have to do is extend the reader class itself
18  // and overload its finalize() member function for decrementing the
19  // par that it is a member of
20  class _proc$reader01099237691_overload_finalize_0 : public
        _proc$reader01099237691
21  {
22  public:
23      _proc$main21169311* parent;
24
25      _proc$reader01099237691_overload_finalize_0(pj_runtime::pj_scheduler*
        sched, int id, pj_runtime::pj_many2many_channel<int32_t>* in,
        _proc$main21169311* parent)
26      : _proc$reader01099237691{sched, id, in}, parent(parent)
```

```
27      {
28      }
29
30      virtual void finalize()
31      {
32          parent->_ld$_par1->decrement();
33      }
34
35      this->sched->insert(new _proc$reader01099237691_overload_finalize_0(
        this->sched, 1, _ld$c5, this));
36  };
```

Listing 3.38: An 'overload and finalize' process definition.

3.3.5 Protocol

To show the generated code for a protocol and all other code associated, we will examine the test *protocolSwitch.pj*, given below in Listing 3.39.

```
1  // wildcard import of std library
2  import std.*;
3
4  // protocol P declaration
5  public protocol P {
6    // protocol case request
7      request : { int number; double amount; }
8      // protocol case reply
9      reply: { boolean status; }
10 }
11
12 // main process
13 public void main(string args[]) {
14   // instantiate new P protocol, with protocol case reply
15     P p = new P { reply: status = true };
16     // protocols can be switched on their cases as shown below
17     switch (p) {
18     case request:
```

```
19          println("number: " + p.number + ", amount: " + p.amount);
20          break;
21      case reply:
22          println("status = " + p.status);
23          break;
24      }
25  }
```

Listing 3.39: *protocolSwitch.pj*.

There are two protocols given here: protocol P. and protocol X. The *main*() process then creates a new P object, and switches on its type, behaving one way or the other depending on the protocol case stored within p. In this way, one can think of a protocol as a distant cousin of a union, or variant. However, the inheritance structure for protocols is *backwards*, compared to the typical inheritance structure of classes.

```
1   class P
2   {
3   public:
4       class request : public pj_runtime::pj_protocol_case
5       {
6       public:
7           int number;
8           double amount;
9
10          request(int number, double amount)
11          {
12              this->number = number;
13              this->amount = amount;
14              this->tag = 0;
15          }
16      };
17
18      class reply : public pj_runtime::pj_protocol_case
19      {
20      public:
21          bool status;
```

```
22
23          reply(bool status)
24          {
25              this->status = status;
26              this->tag = 1;
27          }
28      };
29  };
```

Listing 3.40: *protocolSwitch.pj* - protocol P.

The code generated in Listing 3.40 describes the protocol P. This is simply a class with a nested class within it. In particular, pay attention to the *tag* variable, which is used for the switch-case block seen earlier in the test program. Listing 3.41 shows us how the switch determines which protocol case to select.

```
1  _ld$p1 = new P::reply(true);
2  switch(_ld$p1->tag) {
3  case 0:
4      io::println("number: ", reinterpret_cast<P::request*>(_ld$p1)->number,
       ", amount: ", reinterpret_cast<P::request*>(_ld$p1)->amount);
5      break;
6  case 1:
7      io::println("status = ", reinterpret_cast<P::reply*>(_ld$p1)->status);
8      break;
9  }
```

Listing 3.41: *protocolSwitch.pj* - protocol switch-case.

As seen above, the new protocol case *reply* is allocated and stored. Then, the *tag* variable of the protocol case is switched on. The value given to this variable is generated by the compiler as it visits each protocol case.

3.3.6 Record

Records in ProcessJ can be thought of as structs, and that is exactly what they are mapped to within the C++ representation. Listing 3.42 shows us a few example records, and their generated counterparts. Listing 3.43 shows some code that creates and accesses members within these structs.

```
public record T {
    int a;
}

public record K {
    int z;
    T t;
}

public record X extends T {
    int p;
    string b;
}

...

struct T : public pj_runtime::pj_record
{
public:
    T() = default;
    T(int a)
    {
        this->a = a;
    }

    ~T()
    {
    }
    int a;

};

struct K : public pj_runtime::pj_record
{
public:
    K() = default;
```

```
37      K(int z, T* t)
38      {
39          this->z = z;
40          this->t = t;
41      }
42
43      ~K()
44      {
45          delete t;
46      }
47      int z;
48      T* t;
49
50  };
51
52  // extends records T
53  struct X : public T
54  {
55  public:
56      X() = default;
57      X(int a, int p, std::string b)
58      {
59          this->a = a;
60          this->p = p;
61          this->b = b;
62      }
63
64      ~X()
65      {
66      }
67      int a;
68      int p;
69      std::string b;
70
```

```
71 };
```

Listing 3.42: *pj_record* declarations.

As we see below in Listing 3.43, the writer process creates a number of new records on lines 2-4. Then the writer synchronizes on a barrier, and writes the record object l to its *out* channel. We see the code generated by the compiler on lines 12-37.

```
1  public void writer(chan<L>.write out, barrier b1) {
2    K k = new K { z = 3, t = new T { a = 45 } };
3    X x = new X { b = "Ben", p = 300, a = 20 };
4    L l = new L { k = new K { z = 4, t = new T { a = 65 }}, str = "Benjamin"
       };
5    b1.sync();
6
7    out.write(l);
8  }
9
10 ...
11
12 virtual void run()
13 {
14     switch (get_label())
15     {
16         case 0: goto _proc$writer0536917446L0; break;
17         case 1: goto _proc$writer0536917446L1; break;
18         case 2: goto _proc$writer0536917446L2; break;
19     }
20
21     _proc$writer0536917446L0:
22
23     _ld$k1 = new K {.z = 3, .t = new T {.a = 45}};
24     _ld$x2 = new X {.a = 20, .p = 300, .b = "Ben"};
25     _ld$l3 = new L {.k = new K {.z = 4, .t = new T {.a = 65}}, .str = "
      Benjamin"};
26     _pd$b12->sync(this);
27     this->set_label(1);
28     return;
```

```
29      _proc$writer0536917446L1:
30      _pd$out1->write(this, _ld$13);
31      this->set_label(2);
32      return;
33
34      _proc$writer0536917446L2:
35      _ld$13= nullptr;
36      terminate();
37      return;
38  }
```

Listing 3.43: *pj_record* usage.

Note that although the L defined on line 25 is allocated in one process, the pointer holding it is set null instead of being deleted. This is because objects sent over channels like records or protocols are considered "owned" by the process they are sent to, and thus the recipient is in charge of deallocating the object.

3.3.7 Timer

To demonstrate the code generated for a timer, the test program *timer.pj* will be examined in partial generated form, shown below in Listing 3.44.

```
1  import std.*;
2
3  public void main(string[] args) {
4      timer t;
5      t.timeout(100);
6      long a = t.read();
7      println("> " + a);
8  }
```

Listing 3.44: *timer.pj*.

This test shows both the code for a timeout statement, as well as the code for a timer read. Both of these generated forms are shown below in Listing 3.45.

```
1  virtual void run()
2  {
3      switch (get_label())
```

```
4     {
5         case 0: goto _proc$main01169311L0; break;
6         case 1: goto _proc$main01169311L1; break;
7     }
8
9     _proc$main01169311L0:
10
11    _ld$t1 = static_cast<pj_runtime::pj_timer*>(0);
12    _ld$t1 = new pj_runtime::pj_timer(this, 100);
13    _ld$t1->set_process(this);
14    _ld$t1->timeout(100);
15    _ld$t1->start();
16    this->sched->insert(_ld$t1);
17    this->set_not_ready();
18    this->set_label(1);
19    return;
20
21    _proc$main01169311L1:
22    _ld$a2 = static_cast<long>(0);
23    _ld$a2 = pj_runtime::pj_timer::read();
24    io::println("> ", _ld$a2);
25    terminate();
26    return;
27 }
```

Listing 3.45: *timer.pj* - generated *timeout*() and *read*() code.

As we can see above, the *timeout*() invocation generates a number of lines of code. First, the timer has its process pointer set to the process invoking *timeout*(). The timeout value is set at 100, then the timer is started and inserted into the schedulers timer queue. The process then yields and waits to be set ready to run again by the timer object once it has been dequeued from the timer queue.

The *read*() invocation, on the other hand, generates one line of code: a call to a static member function of the *pj_timer* class that simply returns the current system time in milliseconds, as a long value.

Chapter 4

Results

In this chapter, a few larger examples will be considered to show some of the more interesting problems that can be solved using ProcessJ with the new C++ Runtime System and Code Generator.

4.1 Maximum Running Process Benchmarks

It has been a long-term goal of the ProcessJ group to achieve the fabled "one billion process" runtime goal. As mentioned in [SP16] and [Shr16], we have achieved at maximum 480,900,001 running processes on a single machine. While the machine tested on was quite powerful for the time (having a whopping 128 Gigabytes of RAM, and a 32-core Xeon CPU), the University of Nevada, Las Vegas has acquired a considerably more powerful cluster of machines. One such machine, nicknamed "snorlax" within the "superkitty" computer cluster, has Two AMD EPYC Rome 7452 CPUs, clocked at 2.35GHz (turbo up to 3.35GHz), for a whopping 64 cores, 128 threads total. Also, this machine has 512 GB (in a 16-by-32-GB configuration) 288 pin DDR4 SDRAM clocked at 2666MHz, with ECC.

With the acquisition of this machine, and now being armed with 128 threads and 512GB of unadulterated computational power, it is only natural that one interested in the power of concurrency and parallelism would want to take this machine to the max in terms of ProcessJ's process scheduling system. With this in mind, a test similar to that run by [SP16] and [Shr16] was compiled and run on this machine. The test code in ProcessJ can be seen in Appendix A. The following table shows the results of this testing.

Looking at the results of Table 4.1, it is easy to see that we have now achieved one of the main goals of ProcessJ for many years: we have successfully run over one billion processes on a single

Table 4.1: Process Benchmarking Results

Process Count	**Context Switches**	**Time (seconds)**	**RAM usage (gigabytes)**
6,000,001	11,000,002	5.712	1.119
12,000,0001	22,000,002	17.543	2.367
60,000,001	110,000,002	57.762	11.6
240,000,001	440,000,002	236.923	45.96
360,000,001	660,000,002	365.117	68.4
540,000,001	990,000,002	556.697	102
750,000,001	1,375,000,002	766.755	142
1,002,000,001	1,837,000,002	1,044.424	191
2,502,000,001	4,587,000,002	2,755.303	475

core. While it is also obvious that being able to scale to even six million processes is a great feat over the traditional thread implementations for parallel programming, it is much more exciting to see such massive scalability. However, this is not the most interesting thing to know about the runtime itself. While scaling up to billions of processes is indeed good, it would be much more interesting to know the sizes of the runtime constructs like processes, alternations, and more to see the memory constraints and how they relate to the runtime of ProcessJ itself. Further tests will be done in a future paper to truly grasp the power of this new runtime system.

4.2 CommsTime

To test the efficiency of channel communication in the C++ runtime system, the test CommsTime was used. This test is given in Appendix B. This test consists of a network of processes that communicate one million numbers, starting from 1 and incrementing to 1000000. This test was run on a machine with an Intel i7-6700k clocked at 4.20GHz, with 4 cores and 8 threads available. This machine also has 16 GB (in a 2-by-8-GB configuration) of 288 pin DDR4 SDRAM clocked at 4000MHz. The resulting calculations of this test are in the following table.

Table 4.2: CommsTime Benchmarking Results

μs / iteration	**μs / communication**	**μs / context switch**
0.5308	1.32725	5.309

As an added comparison, though on a different machine, the CommsTime test results from [Shr16] have been included as well in Table 4.3.

Table 4.3: Previous CommsTime Benchmarking Results

	Mac / OS X			AMD / Linux		
	LiteProc	JCSP	PJProcess	LiteProc	JCSP	PJProcess
μs / iteration	9.26	27.00	8.30	13.56	136.00	7.52
μs / communication	2.31	6.00	2.08	3.90	35.00	1.88
μs / context switch	1.32	3.00	0.69	1.94	17.00	0.63

4.3 The Santa Claus Problem

In addition to the previous two conformity tests, we would also benefit from seeing some more real-world-friendly examples. One of these examples is the Santa Claus Problem. This problem, described in detail by [Tro94], has been solved using ProcessJ, as the constructs within the language lend themselves to an elegant solution. The code for this solution is in Appendix C.

One of the more important things to note about the implementation of the Santa Claus problem is that it is notedly difficult to implement in a non-process-oriented language, such as C++. However, as tested here, the ProcessJ C++ code generator is successful in translating the high-level ProcessJ description of the problem into lower-level C++ code that behaves correctly. This is precisely what this test set out to prove about the new code generator and runtime system in terms of conformity at both compiler and runtime environment level.

4.4 Full Adder Implementation

Another interesting real-world-friendly problem is the implementation of an 8-bit full adder in ProcessJ. This shows an interesting mapping of ProcessJ to chip design. The code for this problem is given in Appendix D. To look into this test in more detail, we can use this as an opportunity to see the translation of ProcessJ into C++ in a more effective view. That is, we can use the fact that this program has a wide variety of processes that all do different things to compare and contrast the number of lines in a ProcessJ program versus the number of lines in the generated C++ code for ProcessJ constructs. These comparisons are given in the following table.

This test in particular shows us a different architecture of program in the context of ProcessJ. While the Santa Claus Problem was a single layer of different processes working together, the Full Adder implementation shows a multi-layered system of processes. For example, the 8-bit adder process is made of 2 4-bit adder processes, and those are each made of 4 1-bit adder processes, and so on. The channels used for communication between these lower-level processes are passed

Table 4.4: Full Adder LOC Comparison Table

Full Adder Component	**ProcessJ LOC Count**	**C++ LOC Count**
notGate	5	60
orGate	8	187
andGate	8	187
nandGate	8	96
muxGate	8	180
xorGate	12	219
oneBitAdder	14	271
fourBitAdder	9	174
eightBitAdder	7	164

down from the main process to the very bottom layer of the system itself, which led to a number of implementation detail revisions, such as the placement of process-local variables when translated down to C++ (to satiate any curiosity, these local variables live as member variables within each process' class definition).

Chapter 5

Conclusion

In this thesis, we have introduced a new C++-based runtime system, and a new code generator for the ProcessJ compiler. We have demonstrated the form that ProcessJ's basic constructs take in C++, and shown the facilities of the new runtime system that handle the scheduling and running of processes. To demonstrate the code generated by the compiler, We have also shown the code generated for constructs like alts, barriers, and other CSP concepts that are at the core of ProcessJ. To do this, we have given several test programs and their generated code. To demonstrate the correctness of this runtime, several problems such as the Santa Claus Problem and the implementation of a Full Adder circuit in ProcessJ were examined, along with their ProcessJ solutions.

In addition, the performance of the scheduler was demonstrated with a simple program that enabled us to test the maximum number of processes (2,502,000,001) that could be run in a single scheduler instance within the runtime. The results of these tests have shown us the fully-featured runtime system is capable of handling much more than the JVM-based runtime system is, as the C++ runtime was both faster and consumed much less memory than its JVM counterpart.

In conclusion, the C++ runtime system for ProcessJ is a very powerful, efficient, and performance-oriented system that is an improvement over its predecessors, and the C++ code generator for the ProcessJ compiler correctly generates code that utilizes this runtime system. These two additions to the ProcessJ "family" have not only outperformed their C++ predecessors, such as C++CSP2, but have also outperformed their JVM counterparts within ProcessJ. Thus, the goal of this thesis has been achieved.

Chapter 6

Future Work

A number of possible improvements should be investigated for the C++ runtime system. These possibilities are touched upon here.

6.1 Coroutine-based Implementation

With the addition of C++20, the ProcessJ C++ runtime system would perhaps greatly benefit from the use of the new coroutine library. This library, as discussed earlier in this paper, effectively makes yielding and resumption of functions language-intrinsic, which could possibly eliminate the need for keeping track of a resumption label, as well as provide the ability to use compiler optimizations introduced with this library to improve runtime performance of ProcessJ further.

6.2 Multi-Core Scheduling

The ProcessJ C++ runtime system has a semi-functional prototype of a multi-core scheduler, which is currently in experimental development. We expect another paper to come shortly that expands on this prototype, and introduces an efficient, load-balancing runtime scheduler that can run on not only one, but several cores at once. The performance implications to this on top of the results of maximum process benchmarking are many, and will be expanded upon at a later time.

6.3 Runtime Version 2.0

One of the greater challenges in development of this runtime system was the manual memory management inherent to C++. Without a garbage collector like the JVM runtime system, the question of ownership, and responsibility of deletion is a non-trivial mess. A rewritten runtime

that utilizes smart pointers introduced in later C++ versions (11, 14, etc.) would greatly benefit from compiler optimizations, and the code generation for this runtime would also potentially benefit from simplification of delete statement generation. The use of move and copy semantics from C++ would also benefit the passing of non-primitive data over channels from process to process. This topic will hopefully be expanded upon in a future paper.

Chapter 7

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 1625677. (https://www.nsf.gov/awardsearch/showAward?AWD_ID=1625677)

Appendix A

Billions of Processes Benchmark Test for ProcessJ

```
import std.io;

public void foo(chan<int>.read c1, chan<int>.write c2) {
    int x;
    par {
        x = c1.read();
        c2.write(10);
    }
}

public void bar(chan<int>.write c1, chan<int>.read c2) {
    int y;
    par {
        y = c2.read();
        c1.write(20);
    }
}

public void main(string[] args) {
    par for(int i = 0; i < 1000000; ++i) {
        chan<int> c1, c2;
        foo(c1.read, c2.write);
```

```
23          bar(c1.write, c2.read);
24      }
25  }
```

Listing A.1: *proctest.pj*

Appendix B

CommsTime Conformity Test for ProcessJ

```
1  import std.io;
2
3  public void prefix(long n, chan<long>.read in, chan<long>.write out) {
4    out.write(n);
5    long l = 0;
6    while (l < 1000000) {
7      l = in.read();
8      out.write(l);
9    }
10 }
11
12 public void succ(chan<long>.read in, chan<long>.write out) {
13   long l = 0;
14   while (l < 999999) {
15     l = in.read();
16     out.write(l+1);
17   }
18 }
19
20 public void delta(chan<long>.read in, chan<long>.write out1, chan<long>.
       write out2) {
21   long l = 0;
```

```
22    while (l < 1000000) {
23      l = in.read();
24      par {
25        out1.write(l);
26        if (l != 1000000)
27          out2.write(l);
28      }
29    }
30  }
31
32  public void consume(chan<long>.read in) {
33    long l = 0;
34    while (l < 1000000) {
35      l = in.read();
36      println(l);
37    }
38  }
39
40  public void main(string args[]) {
41    chan<long> a,b,c,d;
42    par {
43      delta(d.read, a.write, b.write);
44      succ(b.read, c.write);
45      prefix(0, c.read, d.write);
46      consume(a.read);
47    }
48  }
```

Listing B.1: *commstime.pj*.

Appendix C

Santa Claus Problem Implementation in ProcessJ

```
import std.*;

const int N_REINDEER = 9;
const int G_REINDEER = N_REINDEER;
const int N_ELVES = 10;
const int G_ELVES = N_ELVES;
const int HOLIDAY_TIME = 100000;
const int WORKING_TIME = 200000;
const int DELIVERY_TIME = 100000;
const int CONSULTATION_TIME = 200000;

protocol Reindeer_msg {
    holiday: { int id; }
    deer_ready: { int id; }
    deliver: { int id; }
    deer_done: { int id; }
}

protocol Elf_msg {
    working: { int id; }
    elf_ready: { int id; }
    waiting: { int id; }
```

```
23      consult: { int id; }
24      elf_done: { int id; }
25  }
26
27  protocol Santa_msg {
28      reindeer_ready: { }
29      harness: { int id; }
30      mush_mush: { }
31      woah: { }
32      unharness: { int id; }
33      elves_ready: { }
34      greet: { int id; }
35      consulting: { }
36      santa_done: { }
37      goodbye: { int id; }
38  }
39
40  protocol Message extends Reindeer_msg, Elf_msg, Santa_msg;
41
42  public void random_wait(long max_wait, long seed) {
43      timer t;
44      long wait;
45      initRandom(seed);
46      wait = longRandom();
47      wait = wait % 250;
48      t.timeout(wait);
49  }
50
51  public void display(chan<Message>.read in) {
52      Message msg;
53      while (true) {
54          msg = in.read();
55          switch (msg) {
56          case holiday:
57              println("                                    Reindeer-" + msg.
    id + ": on holiday ... wish you were here");
```

```
58              break;
59          case deer_ready:
60              println("                                        Reindeer-" + msg.
    id + ": back from holiday ... ready for work");
61              break;
62          case deliver:
63              println("                                        Reindeer-" + msg.
    id + ": delivering toys ... la-di-da-di-da-di-da");
64              break;
65          case deer_done:
66              println("                                        Reindeer-" + msg.
    id + ": all toys delivered ... want a holiday");
67              break;
68          case working:
69              println("                    Elf-" + msg.id + ": working");
70              break;
71          case elf_ready:
72              println("                    Elf-" + msg.id + ": need to
    consult Santa");
73              break;
74          case waiting:
75              println("                    Elf-" + msg.id + ": in the waiting
     room...");
76              break;
77          case consult:
78              println("                    Elf-" + msg.id + ": about these
    toys...??");
79              break;
80          case elf_done:
81              println("                    Elf-" + msg.id + ": OK ... we'll
    built it, bye...");
82              break;
83          case reindeer_ready:
84              println("Santa: Ho-ho-ho ... the reindeer are back!");
85              break;
86          case harness:
```

```
87              println("Santa: harnessing reindeer: " + msg.id);
88              break;
89          case mush_mush:
90              println("Santa: mush mush...");
91              break;
92          case woah:
93              println("Santa: woah... we're back home!");
94              break;
95          case unharness:
96              println("Santa: un-harnessing reindeer: " + msg.id);
97              break;
98          case elves_ready:
99              println("Santa: Ho-ho-ho... some elves are here!");
100             break;
101         case greet:
102             println("Santa: hello elf: " + msg.id);
103             break;
104         case consulting:
105             println("Santa: consulting with elves...");
106             break;
107         case santa_done:
108             println("Santa: OK, all done -- thanks!");
109             break;
110         case goodbye:
111             println("Santa: goodbye elf: " + msg.id);
112             break;
113         }
114     }
115 }
116
117 public void p_barrier_knock(const int n, chan<boolean>.read a,
118                             chan<boolean>.read b,
119                             chan<boolean>.write knock) {
120     while (true) {
121         for (int i = 0; i < n; i++) {
122             boolean any;
```

```
123             any = a.read();
124         }
125 
126         knock.write(true);
127         for (int i = 0; i < n; i++) {
128             boolean any;
129             any = b.read();
130         }
131     }
132 }
133 
134 public void p_barrier(const int n,
135                       chan<boolean>.read a,
136                       chan<boolean>.read b) {
137     while (true) {
138         for (int i = 0; i < n; i++) {
139             boolean any;
140             any = a.read();
141         }
142         for (int i = 0; i < n; i++) {
143             boolean any;
144             any = b.read();
145         }
146     }
147 }
148 
149 public void syncronize(shared chan<boolean>.write a,
150                        shared chan<boolean>.write b) {
151     a.write(true);
152     b.write(true);
153 }
154 
155 public void reindeer(const int id,
156                      const long seed,
157                      barrier just_reindeer,
158                      barrier santa_reindeer,
```

```
159                     shared chan<int>.write to_santa,
160                     shared chan<Reindeer_msg>.write report) {
161     long my_seed = seed;
162     long wait = HOLIDAY_TIME;
163     long t;
164     timer tim;
165 
166     while (true) {
167         report.write(new Reindeer_msg { holiday: id = id });
168         random_wait(wait, my_seed);
169         report.write(new Reindeer_msg { deer_ready: id = id });
170         just_reindeer.sync();
171         to_santa.write(id);
172         santa_reindeer.sync();
173         report.write(new Reindeer_msg { deliver: id = id });
174         santa_reindeer.sync();
175         report.write(new Reindeer_msg { deer_done: id = id });
176         to_santa.write(id);
177     }
178 }
179 
180 public void elf(const int id,
181                 const long seed,
182                 shared chan<boolean>.write elves_a,
183                 shared chan<boolean>.write elves_b,
184                 shared chan<boolean>.write santa_elves_a,
185                 shared chan<boolean>.write santa_elves_b,
186                 shared chan<int>.write to_santa,
187                 shared chan<Elf_msg>.write report) {
188 
189     long my_seed = seed;
190     long wait = WORKING_TIME;
191 
192     while (true) {
193         report.write(new Elf_msg{ working: id = id });
194         random_wait(wait, my_seed);
```

```
        report.write(new Elf_msg{ elf_ready: id = id });
        syncronize(elves_a, elves_b);
        to_santa.write(id);
        syncronize(santa_elves_a, santa_elves_b);
        report.write(new Elf_msg{ consult: id = id });
        syncronize(santa_elves_a, santa_elves_b);
        report.write(new Elf_msg{ elf_done: id = id });
        to_santa.write(id);
    }
}

public void santa(const long seed,
                  chan<boolean>.read knock,
                  chan<int>.read from_reindeer,
                  chan<int>.read from_elf,
                  barrier santa_reindeer,
                  shared chan<boolean>.write santa_elves_a,
                  shared chan<boolean>.write santa_elves_b,
                  shared chan<Santa_msg>.write report) {

    long my_seed = seed;
    timer tim;
    long t, wait;

    while (true) {
        int id;
        boolean answer;
        pri alt {
            id = from_reindeer.read() : {
                report.write(new Santa_msg{ reindeer_ready: });
                report.write(new Santa_msg{ harness: id = id });
                for (int i = 0; i < G_REINDEER-1; i++) {
                    id = from_reindeer.read();
                    report.write(new Santa_msg{ harness: id = id });
                }
                report.write(new Santa_msg{ mush_mush: });
```

```
231                 santa_reindeer.sync();
232                 t = tim.read();
233                 tim.timeout(100);
234                 report.write(new Santa_msg{woah: });
235                 santa_reindeer.sync();
236                 for (int i = 0; i < G_REINDEER; i++) {
237                     id = from_reindeer.read({ report.write(new Santa_msg{
    unharness: id = id }); });
238                 }
239             }
240             answer = knock.read() : {
241                 report.write(new Santa_msg{ elves_ready: });
242                 for (int i = 0; i < G_ELVES; i++) {
243                     id = from_elf.read();
244                     report.write( new Santa_msg{ greet: id = id });
245                 }
246                 syncronize(santa_elves_a, santa_elves_b);
247                 report.write(new Santa_msg{ consulting: });
248                 t = tim.read();
249                 tim.timeout(100);
250                 report.write(new Santa_msg{ santa_done: });
251                 syncronize(santa_elves_a, santa_elves_b);
252                 for (int i = 0; i < G_ELVES; i++) {
253                     id = from_elf.read({ report.write(new Santa_msg{
    goodbye: id = id }); });
254                 }
255             }
256         }
257     }
258 }
259
260 public void main(string[] args) {
261     timer tim;
262     long seed;
263     seed = tim.read();
264     seed = (seed >> 2) + 42;
```

```
265
266     barrier just_reindeer, santa_reindeer;
267
268     shared write chan<boolean> elves_a, elves_b;
269     chan<boolean> knock;
270     shared write chan<boolean> santa_elves_a, santa_elves_b;
271     shared write chan<int> reindeer_santa, elf_santa;
272     shared write chan<Message> report;
273
274     println("SANTA OUTPUT       ELF OUTPUT      REINDEER OUTPUT");
275     println("
    --------------------------------------------------------------------------------
    ");
276
277     par {
278         par enroll santa_reindeer {
279             santa(seed + (N_REINDEER + N_ELVES),
280                   knock.read, reindeer_santa.read,
281                   elf_santa.read, santa_reindeer,
282                   santa_elves_a.write, santa_elves_b.write,
283                   report.write);
284             par enroll just_reindeer, santa_reindeer {
285                 reindeer(0, seed, just_reindeer, santa_reindeer,
286                     reindeer_santa.write, report.write);
287                 reindeer(1, seed + 1, just_reindeer, santa_reindeer,
288                     reindeer_santa.write, report.write);
289                 reindeer(2, seed + 2, just_reindeer, santa_reindeer,
290                     reindeer_santa.write, report.write);
291                 reindeer(3, seed + 3, just_reindeer, santa_reindeer,
292                     reindeer_santa.write, report.write);
293                 reindeer(4, seed + 4, just_reindeer, santa_reindeer,
294                     reindeer_santa.write, report.write);
295                 reindeer(5, seed + 5, just_reindeer, santa_reindeer,
296                     reindeer_santa.write, report.write);
297                 reindeer(6, seed + 6, just_reindeer, santa_reindeer,
298                     reindeer_santa.write, report.write);
```

```
299                 reindeer(7, seed + 7, just_reindeer, santa_reindeer,
300                     reindeer_santa.write, report.write);
301                 reindeer(8, seed + 8, just_reindeer, santa_reindeer,
302                     reindeer_santa.write, report.write);
303             }
304         }
305 
306         par {
307             elf(0, N_REINDEER + seed, elves_a.write, elves_b.write,
    santa_elves_a.write,
308             santa_elves_b.write, elf_santa.write, report.write);
309             elf(1, N_REINDEER + (seed + 1), elves_a.write, elves_b.write,
    santa_elves_a.write,
310             santa_elves_b.write, elf_santa.write, report.write);
311             elf(2, N_REINDEER + (seed + 2), elves_a.write, elves_b.write,
    santa_elves_a.write,
312             santa_elves_b.write, elf_santa.write, report.write);
313             elf(3, N_REINDEER + (seed + 3), elves_a.write, elves_b.write,
    santa_elves_a.write,
314             santa_elves_b.write, elf_santa.write, report.write);
315             elf(4, N_REINDEER + (seed + 4), elves_a.write, elves_b.write,
    santa_elves_a.write,
316             santa_elves_b.write, elf_santa.write, report.write);
317             elf(5, N_REINDEER + (seed + 5), elves_a.write, elves_b.write,
    santa_elves_a.write,
318             santa_elves_b.write, elf_santa.write, report.write);
319             elf(6, N_REINDEER + (seed + 6), elves_a.write, elves_b.write,
    santa_elves_a.write,
320             santa_elves_b.write, elf_santa.write, report.write);
321             elf(7, N_REINDEER + (seed + 7), elves_a.write, elves_b.write,
    santa_elves_a.write,
322             santa_elves_b.write, elf_santa.write, report.write);
323             elf(8, N_REINDEER + (seed + 8), elves_a.write, elves_b.write,
    santa_elves_a.write,
324             santa_elves_b.write, elf_santa.write, report.write);
325             elf(9, N_REINDEER + (seed + 9), elves_a.write, elves_b.write,
```

```
    santa_elves_a.write,
326             santa_elves_b.write, elf_santa.write, report.write);
327         }
328
329         display(report.read);
330         p_barrier_knock(G_ELVES, elves_a.read, elves_b.read, knock.write);
331         p_barrier(G_ELVES + 1, santa_elves_a.read, santa_elves_b.read);
332     }
333 }
```

Listing C.1: *santa.pj*.

Appendix D

Full Adder Implementation in ProcessJ

```
1  import std.*;
2
3  public void notGate(chan<boolean>.read in, chan<boolean>.write out) {
4      boolean x = false;
5      x = in.read();
6      out.write(!x);
7  }
8
9  public void orGate(chan<boolean>.read in1, chan<boolean>.read in2, chan<
       boolean>.write out) {
10     boolean x = false, y = false;
11     par{
12         x = in1.read();
13         y = in2.read();
14     }
15     out.write(x || y);
16 }
17
18 public void andGate(chan<boolean>.read in1, chan<boolean>.read in2,chan<
       boolean>.write out) {
19     boolean x = false, y = false;
20     par {
```

```
21          x = in1.read();
22          y = in2.read();
23      }
24      out.write(x && y);
25  }
26
27  public void nandGate(chan<boolean>.read in1, chan<boolean>.read in2,
28                       chan<boolean>.write out) {
29      chan<boolean> a;
30      par {
31          andGate(in1, in2, a.write);
32          notGate(a.read, out);
33      }
34      return;
35  }
36
37  public void muxGate(chan<boolean>.read in, chan<boolean>.read out1,
38                      chan<boolean>.write out2) {
39      boolean x = false; x = in.read();
40      par {
41          out1.write(x);
42          out2.write(x);
43      }
44      return;
45  }
46
47  public void xorGate(chan<boolean>.read in1, chan<boolean>.read in2,
48                      chan<boolean>.write out) {
49      chan<boolean> a, b, c, d , e, f, g, h, i;
50      par {
51          muxGate(in1, a.read, b.write);
52          muxGate(in2, c.read, d.write);
53          nandGate(b.read, d.read, e.write);
54          muxGate(e.read, f.read, g.write);
55          nandGate(a.read, f.read, h.write);
56          nandGate(c.read, g.read, i.write);
```

```
57          nandGate(h.read, i.read, out);
58      }
59  }
60
61  public void oneBitAdder(chan<boolean>.read in1, chan<boolean>.read in2,
62                          chan<boolean>.read in3, chan<boolean>.write result
    ,
63                          chan<boolean>.write carry) {
64      chan<boolean> a, b, c, d, e, f, g, h, i, j, k;
65          par{
66              muxGate(in1, a.read, b.write);
67              muxGate(in2, c.read, d.write);
68              xorGate(a.read, c.read, e.write);
69              muxGate(e.read, f.read, g.write);
70              muxGate(in3, h.read, i.write);
71              xorGate(f.read, h.read, result);
72              andGate(g.read, i.read, j.write);
73              andGate(b.read, d.read, k.write);
74              orGate(j.read, k.read, carry);
75          }
76  }
77
78  public void fourBitAdder(chan<boolean>.read inA0, chan<boolean>.read inA1,
79                           chan<boolean>.read inA2, chan<boolean>.read inA3,
80                           chan<boolean>.read inB0, chan<boolean>.read inB1,
81                           chan<boolean>.read inB2, chan<boolean>.read inB3,
82                           chan<boolean>.read inCarry, chan<boolean>.write
    result0,
83                           chan<boolean>.write result1, chan<boolean>.write
    result2,
84                           chan<boolean>.write result3, chan<boolean>.write
    carry) {
85      chan<boolean> a, b, c;
86      par {
87          oneBitAdder(inA0, inB0, inCarry, result0, a.write);
88          oneBitAdder(inA1, inB1, a.read, result1, b.write);
```

```
89          oneBitAdder(inA2, inB2, b.read, result2, c.write);
90          oneBitAdder(inA3, inB3, c.read, result3, carry);
91      }
92  }
93
94  public void eightBitAdder(chan<boolean>.read inA0, chan<boolean>.read inA1
        ,
95                            chan<boolean>.read inA2, chan<boolean>.read inA3
        ,
96                            chan<boolean>.read inA4, chan<boolean>.read inA5
        ,
97                            chan<boolean>.read inA6, chan<boolean>.read inA7
        ,
98                            chan<boolean>.read inB0, chan<boolean>.read inB1
        ,
99                            chan<boolean>.read inB2, chan<boolean>.read inB3
        ,
100                           chan<boolean>.read inB4, chan<boolean>.read inB5
        ,
101                           chan<boolean>.read inB6, chan<boolean>.read inB7
        ,
102                           chan<boolean>.read inCarry, chan<boolean>.write
       result0,
103                           chan<boolean>.write result1, chan<boolean>.write
        result2,
104                           chan<boolean>.write result3, chan<boolean>.write
        result4,
105                           chan<boolean>.write result5, chan<boolean>.write
        result6,
106                           chan<boolean>.write result7, chan<boolean>.write
        outCarry) {
107     chan<boolean> a;
108     par {
109         fourBitAdder(inA0, inA1, inA2, inA3,
110                      inB0, inB1, inB2, inB3,
111                      inCarry, result0, result1,
```

```
                    result2, result3, a.write);
        fourBitAdder(inA4, inA5, inA6, inA7,
                    inB4, inB5, inB6, inB7,
                    a.read,
                    result4, result5, result6,
                    result7, outCarry);
    }
}

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;
```

```
148     q3 = true;
149     q4 = false;
150     q5 = true;
151     q6 = false;
152     q7 = true;
153 
154     par {
155         // First number
156         a7.write(p7);
157         a6.write(p6);
158         a5.write(p5);
159         a4.write(p4);
160         a3.write(p3);
161         a2.write(p2);
162         a1.write(p1);
163         a0.write(p0);
164 
165         // Second number
166         b7.write(q7);
167         b6.write(q6);
168         b5.write(q5);
169         b4.write(q4);
170         b3.write(q3);
171         b2.write(q2);
172         b1.write(q1);
173         b0.write(q0);
174 
175         // Initial carry
176         inCarry.write(inC);
177 
178         eightBitAdder(a0.read, a1.read, a2.read, a3.read,
179                       a4.read, a5.read, a6.read, a7.read,
180                       b0.read, b1.read, b2.read, b3.read,
181                       b4.read, b5.read, b6.read, b7.read,
182                       inCarry.read, r0.write, r1.write,
183                       r2.write, r3.write, r4.write, r5.write,
```

```
184                         r6.write, r7.write, outCarry.write);
185
186         f0 = r0.read();
187         f1 = r1.read();
188         f2 = r2.read();
189         f3 = r3.read();
190         f4 = r4.read();
191         f5 = r5.read();
192         f6 = r6.read();
193         f7 = r7.read();
194
195         c = outCarry.read();
196     }
197
198     println("  " + p7 + " " + p6 + " " + p5 + " " + p4 + " " + p3 + " " +
    p2 + " " + p1 + " " + p0 + " (InCarry:"  + inC + ")");
199     println("+ " + q7 + " " + q6 + " " + q5 + " " + q4 + " " + q3 + " " +
    q2 + " " + q1 + " " + q0);
200     println("----------");
201     println(" " + f7 + " " + f6 + " " + f5 + " " + f4 + " " + f3 + " " +
    f2 + " " + f1 + " " + f0);
202     println("Carry was: " + c);
203 }
```

Listing D.1: *fulladder.pj*.

Bibliography

[Bro07] Neil C. C. Brown. C++CSP2: A Many-to-Many Threading Model for Multicore Architectures. In Alistair A. McEwan, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, pages 183–205, jul 2007.

[Cis19] Benjamin Cisneros. ProcessJ: The JVMCSP Code Generator. Master's thesis, 2019.

[CPP20] CPPReference. Coroutines (c++20), October 2020.

[DM98] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry Standard for Shared-Memory Programming. In *IEEE Computational Science and Engineering*, pages 46–55, jan-march 1998.

[Fou20a] Python Software Foundation. CPython Github Repository, October 2020.

[Fou20b] Python Software Foundation. Jython, October 2020.

[Hoa85] Tony C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[oK20] University of Kent. The occam-π programming language, October 2020.

[PC19] Jan Pedersen and Kevin Chalmers. Verifying Channel Communication Correctness for a Multi-Core Cooperatively Scheduled Runtime Using CSP. In *2019 IEEE/ACM 7th International Conference on Formal Methods in Software Engineering (FormaliSE)*, pages 65–74, may 2019.

[Pro20] The OpenMPI Project. OpenMPI: Open Source High Performance Computing, October 2020.

[Ros10] A.W. Roscoe. *Understanding Concurrent Systems.* Springer, 2010.

[Shr16] Cabel Shrestha. The JVMCSP Runtime and Code Generator for ProcessJ in Java. Master's thesis, 2016.

[SP16] Cabel Shrestha and Jan Pedersen. JVMCSP - Approaching Billions of Processes on a Single-Core JVM. In *Communicating Process Architectures 2016*, 2016.

[Tro94] J Trono. A New Exercise in Concurrency. volume 26, pages 3, 8–10, 1994.

[WA05] Barry Wilkinson and Michael Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers.* Prentice-Hall, Upper Saddle River, NJ, 2005.

[WBM+07] Peter Welch, Neil Brown, James Moores, Kevin Chalmers, and Bernhard Sputh. *Integrating and Extending JCSP*. 2007.

Curriculum Vitae

Graduate College

University of Nevada, Las Vegas

Alexander C. Thomason (athoma@protonmail.com)

Degrees:

Bachelor of Science in Computer Science 2018

University of Nevada Las Vegas

Thesis Title: The ProcessJ C++ Runtime System and 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. Sarah Harris, Ph.D.