# The JVMCSP Runtime and Code Generator for ProcessJ in Java

Cabel Dhoj Shrestha
*University of Nevada, Las Vegas*, shresc1@unlv.nevada.edu

# THE JVMCSP RUNTIME AND CODE GENERATOR FOR PROCESSJ IN JAVA

by

Cabel Dhoj Shrestha

Bachelor of Science in Computer Engineering (B.Sc.CE)
University of Cebu, Cebu, Philippines
2007

A thesis submitted in partial fulfillment of
the requirements for the

Master of Science in Computer Science

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

University of Nevada, Las Vegas
May 2016

UNLV | GRADUATE COLLEGE

**Thesis Approval**

The Graduate College
The University of Nevada, Las Vegas

May 13, 2016

This thesis prepared by

Cabel Dhoj Shrestha

entitled

The JVMCSP Runtime and Code Generator for ProcessJ in Java

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*

Kazem Taghva, Ph.D.
*Examination Committee Member*

Andreas Stefik, Ph.D.
*Examination Committee Member*

Hualiang Teng, Ph.D.
*Graduate College Faculty Representative*

Kathryn Hausbeck Korgan, Ph.D.
*Graduate College Interim Dean*

# Abstract

The modern day advancements in multi-core technologies require programmers to use the right tools and languages to fully harness their potentials. On that front, our endeavor lies in developing a new multiprocessing programming language. Concurrent or parallel programs can be hard to get right because of locks, monitors, mutexes, etc. One solution is using a CSP based process-oriented language. Process-oriented programming alleviates many of the problems found in thread and lock programming by proper encapsulation of data, explicit synchronous message passing, and the ability to verify code to be free of deadlocks and livelocks by using tools like FDR. Therefore, we have developed a new language called ProcessJ (CSP semantics and Java-like syntax) as a way to modernize languages like occam/occam-$\pi$ which are outdated and only run on certain Linux distributions. ProcessJ is a multi-backend language with a compiler written in Java; and in this thesis, we focus on the JVM backend, which we call the JVMCSP; in particular, we consider code generation, the necessary runtime classes to support concurrency constructs on the JVM, and a simple cooperative non-preemptive scheduler. We also show how to translate ProcessJ source into Java source that makes use of the runtime classes that we have developed.

# Acknowledgements

As much as I looked forward to writing about what I accomplished in this thesis, I have looked forward to writing this section equally as I have much to be thankful for.

First and foremost, I would like to thank my advisor, Dr. Pedersen, without whom this thesis would have never seen the light of day. His continued supervision with the design of this thesis implementation made it more concrete and his patience and encouragement helped me stay motivated. I would like to thank him for keeping me on my toes and inspiring me always to think and do more.

I would also like to thank Dr. Kazem Taghva, Dr. Andreas Stefik, and Dr. Hualiang (Harry) Teng for being a part of my defense committee.

Thank you to all my friends who supported me during this thesis with words of encouragement and/or by just being there when I needed to unwind.

Finally, I need to thank my family and my girlfriend, Dr. Yelungka Thapa - my dad and mom who are my heroes; my two brothers who keep me in check; and my girlfriend for being just the way she is. Without these people, I wouldn't have made it this far with this thesis or in life.

CABEL DHOJ SHRESTHA

*University of Nevada, Las Vegas*

*May 2016*

# Table of Contents

**Abstract** iii

**Acknowledgements** iv

**Table of Contents** v

**List of Tables** ix

**List of Figures** x

**List of Listings** xi

**Chapter 1 Introduction** 1

1.1 Objective . . . 2

1.2 Outline . . . 2

**Chapter 2 Background** 4

2.1 Communicating Sequential Processes (CSP) . . . 4

2.2 Scheduling . . . 6

2.2.1 Preemptive . . . 6

2.2.2 Non-preemptive or Cooperative . . . 6

2.3 The ASM Library . . . 7

2.4 The StringTemplate Templating Engine . . . 7

2.5 Process-oriented Design in ProcessJ . . . 9

2.5.1 Targets . . . 11

**Chapter 3 Related Work** 13

3.1 Process resumption and cooperative scheduling on the JVM . . . 13

3.2 Other Process Oriented Languages or Libraries . . . . . . . . . . . . . . . . . . . . . . 15

3.2.1 occam / occam-$\pi$ and the CCSP runtime . . . . . . . . . . . . . . . . . . . 15

3.2.2 JCSP . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15

3.3 Other approaches to concurrency . . . . . . . . . . . . . . . . . . . . . . . . . 16

3.3.1 MPI . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

3.3.2 Threads/Locks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18

**Chapter 4 Implementation** **20**

4.1 The JVMCSP Runtime . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20

4.1.1 PJProcess . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21

4.1.2 PJChannel . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23

4.1.3 PJOne2OneChannel . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26

4.1.4 PJMany2OneChannel . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

4.1.5 PJOne2ManyChannel . . . . . . . . . . . . . . . . . . . . . . . . . . . 31

4.1.6 PJMany2ManyChannel . . . . . . . . . . . . . . . . . . . . . . . . . . 32

4.1.7 PJPar . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35

4.1.8 PJTimer . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36

4.1.9 PJAlt . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37

4.1.10 PJBarrier . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41

4.1.11 PJProtocolCase . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42

4.1.12 The Scheduler and its Components . . . . . . . . . . . . . . . . . . . . 43

4.1.12.1 The Run Queue . . . . . . . . . . . . . . . . . . . . . . . . . 43

4.1.12.2 The Timer Queue . . . . . . . . . . . . . . . . . . . . . . . . 43

4.1.12.3 Inactive Pool . . . . . . . . . . . . . . . . . . . . . . . . . . 45

4.1.12.4 The Scheduler . . . . . . . . . . . . . . . . . . . . . . . . . . 45

4.2 Language Constructs and Code Generation . . . . . . . . . . . . . . . . . . . 47

4.2.1 Basic Statements, Expressions and Loop/Condition Controls . . . . . . . . . . . . . 47

4.2.2 Process . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47

4.2.3 Par Block . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50

4.2.4 Channels . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54

4.2.5 Claim . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57

4.2.6 Timers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59

4.2.7 Alternative (ALT) . . . . . 61
4.2.8 Barrier . . . . . 65
4.2.9 Records . . . . . 68
4.2.10 Protocol . . . . . 70
4.3 State Management and Rewriting for Resumption . . . . . 75
4.3.1 State Management . . . . . 75
4.3.2 Bytecode Instrumentation with ASM for Resumption . . . . . 75

**Chapter 5 Results** **79**
5.1 The Runtime Object Sizes . . . . . 79
5.2 ProcessJ Benchmarking for Max Number of Processes . . . . . 80
5.3 Timing (CommsTime) . . . . . 80
5.4 Conformity Tests . . . . . 81
5.4.1 Mandelbrot Generation . . . . . 82
5.4.2 The Santa Claus Problem . . . . . 83
5.4.3 Full Adder . . . . . 84

**Chapter 6 Conclusion** **89**

**Chapter 7 Future Work** **91**
7.1 Multi-Core Scheduler . . . . . 91
7.2 Libraries . . . . . 91
7.3 Blocking I/O Calls . . . . . 91
7.4 Mobile Procedure and Polymorphic Resumption . . . . . 92
7.5 Alternative (alt) . . . . . 92
7.6 Claim . . . . . 92
7.7 Run Queue . . . . . 92

**Appendix A Billions of Processes Benchmark Test For ProcessJ** **94**

**Appendix B Parallel MandelBrot in ProcessJ** **95**

**Appendix C Sequential MandelBrot in Java** **97**

**Appendix D Parallel Mandelbrot in C using Open MPI library.** **99**

| | |
|---|---|
| **Appendix E The Santa Claus Problem in ProcessJ** | **104** |
| **Appendix F Full Adder in ProcessJ** | **113** |
| **Bibliography** | **120** |
| **Curriculum Vitae** | **123** |

# List of Tables

5.1 Runtime class-size in bytes. . . . . . . . . . 79

5.2 Benchmark test. . . . . . . . . . 80

5.3 CommsTime results. . . . . . . . . . 82

5.4 Mandelbrot picture results. . . . . . . . . . 82

5.5 *The Full Adder LOC* - ProcessJ vs Java generated code. . . . . . . . . . 84

# List of Figures

1.1 Outline of the ProcessJ source code conversion to an executable jar file. . . . . . . . . . . . 3

2.1 Bytecode for the sample program. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12

4.1 The ProcessJ grammar for a procedure. . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
4.2 The ProcessJ grammar for blocks and par-blocks. . . . . . . . . . . . . . . . . . . . . . . 50
4.3 The ProcessJ grammar for Channels. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
4.4 The ProcessJ grammar for claims. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
4.5 The ProcessJ grammar for timer. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
4.6 The ProcessJ grammar for alt statement. . . . . . . . . . . . . . . . . . . . . . . . . . . 61
4.7 The ProcessJ grammar for a barrier declaration. . . . . . . . . . . . . . . . . . . . . . . 65
4.8 The ProcessJ grammar for records. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
4.9 The ProcessJ grammar protocol. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
4.10 Bytecode for channel-read in ProcessJ. . . . . . . . . . . . . . . . . . . . . . . . . . . 77
4.11 Bytecode after instrumentation. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78

5.1 The CommsTime network. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
5.2 The mandelbrot picture generated by the parallel ProcessJ program. . . . . . . . . . . . . 83

# List of Listings

2.1 A sample Java program. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

2.2 ProcessJ code for barrier sync. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

2.3 The template in StringTemplate for *sync* code generation. . . . . . . . . . . . . . . . . . . 8

2.4 Code generator method for barrier sync code generation. . . . . . . . . . . . . . . . . . . 9

2.5 Generated Java code for barrier sync. . . . . . . . . . . . . . . . . . . . . . . . . . . 9

3.1 Sample occam-$\pi$ code for a procedure with a parallel block. . . . . . . . . . . . . . . . . 15

3.2 *JCSP* - sample code for process. . . . . . . . . . . . . . . . . . . . . . . . . . . . 16

3.3 Examples of some Open MPI APIs. . . . . . . . . . . . . . . . . . . . . . . . . . . . 18

4.1 The *PJProcess* class. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21

4.2 *PJProcess* - execution methods. . . . . . . . . . . . . . . . . . . . . . . . . . . . 21

4.3 *PJProcess* - ‘ready’ flag setter methods. . . . . . . . . . . . . . . . . . . . . . . . . 22

4.4 *PJProcess* - resumption placeholder methods. . . . . . . . . . . . . . . . . . . . . . 23

4.5 The *PJChannel* class. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24

4.6 *PJChannel* - status check methods. . . . . . . . . . . . . . . . . . . . . . . . . . . 25

4.7 *PJChannel* - claim methods. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26

4.8 The *PJOne2OneChannel* class. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

4.9 *PJOne2OneChannel* - write method. . . . . . . . . . . . . . . . . . . . . . . . . . . 27

4.10 *PJOne2OneChannel* - read method. . . . . . . . . . . . . . . . . . . . . . . . . . . 28

4.11 *PJOne2OneChannel* - extended rendezvous. . . . . . . . . . . . . . . . . . . . . . . . 28

4.12 The *PJMany2OneChannel* class. . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

4.13 *PJMany2OneChannel* - write method. . . . . . . . . . . . . . . . . . . . . . . . . . 29

4.14 *PJMany2OneChannel* - read method. . . . . . . . . . . . . . . . . . . . . . . . . . . 30

4.15 *PJMany2OneChannel* - extended rendezvous. . . . . . . . . . . . . . . . . . . . . . . 30

4.16 The *PJOne2ManyChannel* class. . . . . . . . . . . . . . . . . . . . . . . . . . . 31
4.17 *PJOne2ManyChannel* - write method. . . . . . . . . . . . . . . . . . . . . . . . . 31
4.18 *PJOne2ManyChannel* - read method. . . . . . . . . . . . . . . . . . . . . . . . . 32
4.19 *PJOne2ManyChannel* - extended rendezvous. . . . . . . . . . . . . . . . . . . . 32
4.20 The *PJMany2ManyChannel* class. . . . . . . . . . . . . . . . . . . . . . . . . . 33
4.21 *PJMany2ManyChannel* - read/write methods. . . . . . . . . . . . . . . . . . . . 33
4.22 *PJMany2ManyChannel* - extended rendezvous. . . . . . . . . . . . . . . . . . . 34
4.23 The *PJPar* class. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
4.24 The *PJTimer* class. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
4.25 *PJTimer* - the `killed` flag. . . . . . . . . . . . . . . . . . . . . . . . . . . 36
4.26 *PJTimer* - normal control methods. . . . . . . . . . . . . . . . . . . . . . . . 37
4.27 *PJTimer* - the Delayed interface implementation. . . . . . . . . . . . . . . . . 38
4.28 The *PJAlt* class. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
4.29 *PJAlt* - guards initialization. . . . . . . . . . . . . . . . . . . . . . . . . . 39
4.30 *PJAlt* - finding the ready guard. . . . . . . . . . . . . . . . . . . . . . . . . 40
4.31 The *PJBarrier* class. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
4.32 *PJBarrier* - enrolling and resigning. . . . . . . . . . . . . . . . . . . . . . . 41
4.33 *PJBarrier* - syncing. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
4.34 The *PJProtocolCase* class. . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
4.35 The *RunQueue* class. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
4.36 The *TimerQueue* class. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
4.37 *TimerQueue* - thread instance. . . . . . . . . . . . . . . . . . . . . . . . . . 44
4.38 The *InactivePool* class. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
4.39 The *Scheduler* class. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
4.40 *Scheduler* - run method. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
4.41 A generic procedure in ProcessJ. . . . . . . . . . . . . . . . . . . . . . . . . . 48
4.42 The generated Java process class for Listing 4.41. . . . . . . . . . . . . . . . . 49
4.43 ProcessJ code for a par block. . . . . . . . . . . . . . . . . . . . . . . . . . . 50
4.44 ProcessJ code for par block with enroll. . . . . . . . . . . . . . . . . . . . . . 51
4.45 Generated Java code for Listing 4.43. . . . . . . . . . . . . . . . . . . . . . . 52
4.46 Generated Java code for Listing 4.44. . . . . . . . . . . . . . . . . . . . . . . 53
4.47 Example of channel declarations. . . . . . . . . . . . . . . . . . . . . . . . . . 55

4.48 Example of channel read operation. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
4.49 Example of channel write operation. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
4.50 Generated Java code for Listing 4.47. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
4.51 Generated Java code for Listing 4.48. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
4.52 Generated Java code for Listing 4.49. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
4.53 ProcessJ code for claim statement. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
4.54 Another ProcessJ example for claim statement. . . . . . . . . . . . . . . . . . . . . . . 58
4.55 Generated Java code for Listing 4.53. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
4.56 Generated Java code for Listing 4.54. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
4.57 ProcessJ timer declaration and read. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
4.58 ProcessJ timer timeout statement. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
4.59 Generated Java code for timer read in Listing 4.57. . . . . . . . . . . . . . . . . . . . . . 60
4.60 Generated Java code for timeout in Listing 4.58. . . . . . . . . . . . . . . . . . . . . . . 61
4.61 A generic example of an alt statement in ProcessJ. . . . . . . . . . . . . . . . . . . . . . 63
4.62 Generated Java code for Listing 4.61. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
4.63 The PJBarrier declaration. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
4.64 Par block enrolled on a barrier. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
4.65 A barrier-enrolled procedure. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
4.66 Generated Java code for Listing 4.63. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
4.67 Generated Java code for Listing 4.64. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
4.68 Generated Java code for Listing 4.65 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
4.69 Record declaration and extension. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
4.70 Record literal. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
4.71 A more complex record example. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
4.72 Generated Java code for Listing 4.69. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
4.73 Generated Java code for Listing 4.70. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
4.74 Generated Java code for Listing 4.71. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
4.75 An example of protocol declaration. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
4.76 ProcessJ code for a protocol with an extends. . . . . . . . . . . . . . . . . . . . . . . . . 71
4.77 ProcessJ code for a protocol with multiple extends. . . . . . . . . . . . . . . . . . . . . . 72
4.78 ProcessJ code for protocol with empty case tag. . . . . . . . . . . . . . . . . . . . . . . 72
4.79 Generated Java code for Listing 4.75. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72

4.80 Generated Java code for Listing 4.76. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
4.81 Generated Java code for Listing 4.77. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
4.82 Generated Java code for Listing 4.78. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
4.83 ProcessJ code for a par block. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
4.84 Java code for the channel read in par block. . . . . . . . . . . . . . . . . . . . . . . . 76

5.1 Example code of *Full Adder* AND gate in ProcessJ. . . . . . . . . . . . . . . . . . . . . 84
5.2 Generated Java code for *Full Adder* AND gate - part1. . . . . . . . . . . . . . . . . . . 85
5.3 Generated Java code for *Full Adder* AND gate - part1. . . . . . . . . . . . . . . . . . . 86
5.4 Generated Java code for *Full Adder* AND gate - part2. . . . . . . . . . . . . . . . . . . 87
5.5 Generated Java code for *Full Adder* AND gate - part3. . . . . . . . . . . . . . . . . . . 88

A.1 ProcessJ code for benchmarking # of processes that can be run on a single-core JVM. . . . . 94

B.1 ProcessJ code - parallel *mandelbrot* part 1. . . . . . . . . . . . . . . . . . . . . . . . 95
B.2 ProcessJ code - parallel *mandelbrot* part 2. . . . . . . . . . . . . . . . . . . . . . . . 96

C.1 Java code - sequential *mandelbrot* part 1. . . . . . . . . . . . . . . . . . . . . . . . . 97
C.2 Java code - sequential *mandelbrot* part 2. . . . . . . . . . . . . . . . . . . . . . . . . 98

D.1 Parallel Mandelbrot in C using OpenMPI - part 1. . . . . . . . . . . . . . . . . . . . . . 99
D.2 Parallel Mandelbrot in C using OpenMPI - part 2. . . . . . . . . . . . . . . . . . . . . . 100
D.3 Parallel Mandelbrot in C using OpenMPI - part 3. . . . . . . . . . . . . . . . . . . . . . 101
D.4 Parallel Mandelbrot in C using OpenMPI - part 4. . . . . . . . . . . . . . . . . . . . . . 102
D.5 Parallel Mandelbrot in C using OpenMPI - part 5. . . . . . . . . . . . . . . . . . . . . . 103

E.1 ProcessJ code - *santa* part 1. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 104
E.2 ProcessJ code - *santa* part 2. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 105
E.3 ProcessJ code - *santa* part 3. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 106
E.4 ProcessJ code - *santa* part 4. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 107
E.5 ProcessJ code - *santa* part 5. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 108
E.6 ProcessJ code - *santa* part 6. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 109
E.7 ProcessJ code - *santa* part 7. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 110
E.8 ProcessJ code - *santa* part 8. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 111
E.9 ProcessJ code - *santa* part 9. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 112

F.1 ProcessJ code - *fullAdder* part 1. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 113
F.2 ProcessJ code - *fullAdder* part 2. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 114
F.3 ProcessJ code - *fullAdder* part 3. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 115
F.4 ProcessJ code - *fullAdder* part 4. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 116
F.5 ProcessJ code - *fullAdder* part 5. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 117
F.6 ProcessJ code - *fullAdder* part 6. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 118
F.7 ProcessJ code - *fullAdder* part 7. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 119

# Chapter 1

# Introduction

Concurrent programming is needed to efficiently exploit today's multi-core hardware capabilities. In the past, programmers experienced speedup by running their sequential program on newer and more capable but very expensive CPUs. Though the transistor count in the CPU has seen the exponential increase that Moore predicted, the clock speed tops out at 5GHz [Sut09]. Therefore, major chip makers like Intel and AMD have moved on to multi-core architectures. Every personal computer today comes equipped with at least 2 to 4 cores. Therefore, in order to efficiently make use of a multi-core system, a program must be concurrent, or parallel. Concurrency in a program allows for distributing tasks to run simultaneously on many cores or CPUs, thus taking advantage of the increased computational power. Concurrent programming can be hard if not done with the right tools and methods. One often utilized approach is a shared-memory-with-threads approach. Multiple threads operate concurrently and share state in memory. However, programming using threads and reasoning about its execution integrity can be very difficult considering usage of various combination of locks, mutexes and semaphores to prevent race conditions and support linearizability. Here we introduce a new process-oriented, concurrent language, ProcessJ [PS13], where no memory is shared and communication is explicit message passing on channels. We think that it might be a possible solution to the complexities of programming with thread and lock as adapting the CSP design will help detect livelocks and deadlocks using model checking tools such Failures-Divergence Refinement (FDR) [For00].

ProcessJ is a general programming language that supports concurrency with Communicating Sequential Processes (CSP) [Hoa78] semantics and Java-like syntax. One of the goals is making it available for multiple platforms. Current targets are Linux binaries using C and the CCSP runtime [Moo99] and the JVM. In this thesis, we present the design of a JVM runtime system, called JVMCSP, and the code generator in the ProcessJ compiler.

## 1.1 Objective

The objective of this undertaking is to run ProcessJ code on the JVM, through the use of generated Java code, the Java compiler, and a simple single-core scheduler. We intend to run millions of extremely lightweight processes on the JVM. As discussed in [PS14], the JVM threading model that uses the pre-emptive scheduling mechanism is not an option. For that reason, ProcessJ needs its own user-level scheduler. Later on, a more efficient multicore scheduler is planned (see Section 7.1). We also need to write a robust runtime system in Java to fully support all the language features for concurrency and develop a code generator that translates the ProcessJ source to Java source.

Fig. 1.1 shows a complete picture of the necessary elements and the control flow of turning ProcessJ source code into an executable jar file. All the elements in *yellow* were developed as a part of this thesis. ProcessJ source file is compiled using the ProcessJ compiler which generates a Java source file. It is then compiled into Java class files using the Java compiler. Resumption points are instrumented (Section 4.3.2) into the bytecodes using the ASM tool [BLC02]. The instrumented classes are then packaged along with the ProcessJ jar and the runtime jar using a script to generated the final executable jar file.

## 1.2 Outline

In Chapter 1 we introduced the problem addressed in this thesis. In Chapter 2, we briefly describe CSP, give some background on pre-emptive vs cooperative scheduling, the tools we have used to accomplish the task, and finally, process-oriented design in ProcessJ. In Chapter 3, we look at some related languages and libraries to understand why ProcessJ, another CSP based language, is necessary. Chapter 4 contains a detailed description of the implementation and the approach we took to design the ProcessJ runtime, JVMCSP, and how ProcessJ source code is translated to its equivalent Java code. We also look at how an important aspect of ProcessJ, yielding and resumption, is made to work on the JVM by bytecode rewriting techniques. In Chapter 5, we present the results of timing tests, runtime class object sizes, the total number of processes we were able to run on a single-CPU JVM, and consider some conformity tests that we successfully ran. Chapter 6 concludes the thesis with a summary of what was done and what the results were. With ProcessJ, we aim to build a fully capable multi-core and multi-backend concurrent language. As this thesis only covers the runtime and the code generation for a single-core scheduler with JVM there is more work to be done. Chapter 7 lists the future work and some implementation improvements that would be beneficial in terms of the running time and space.

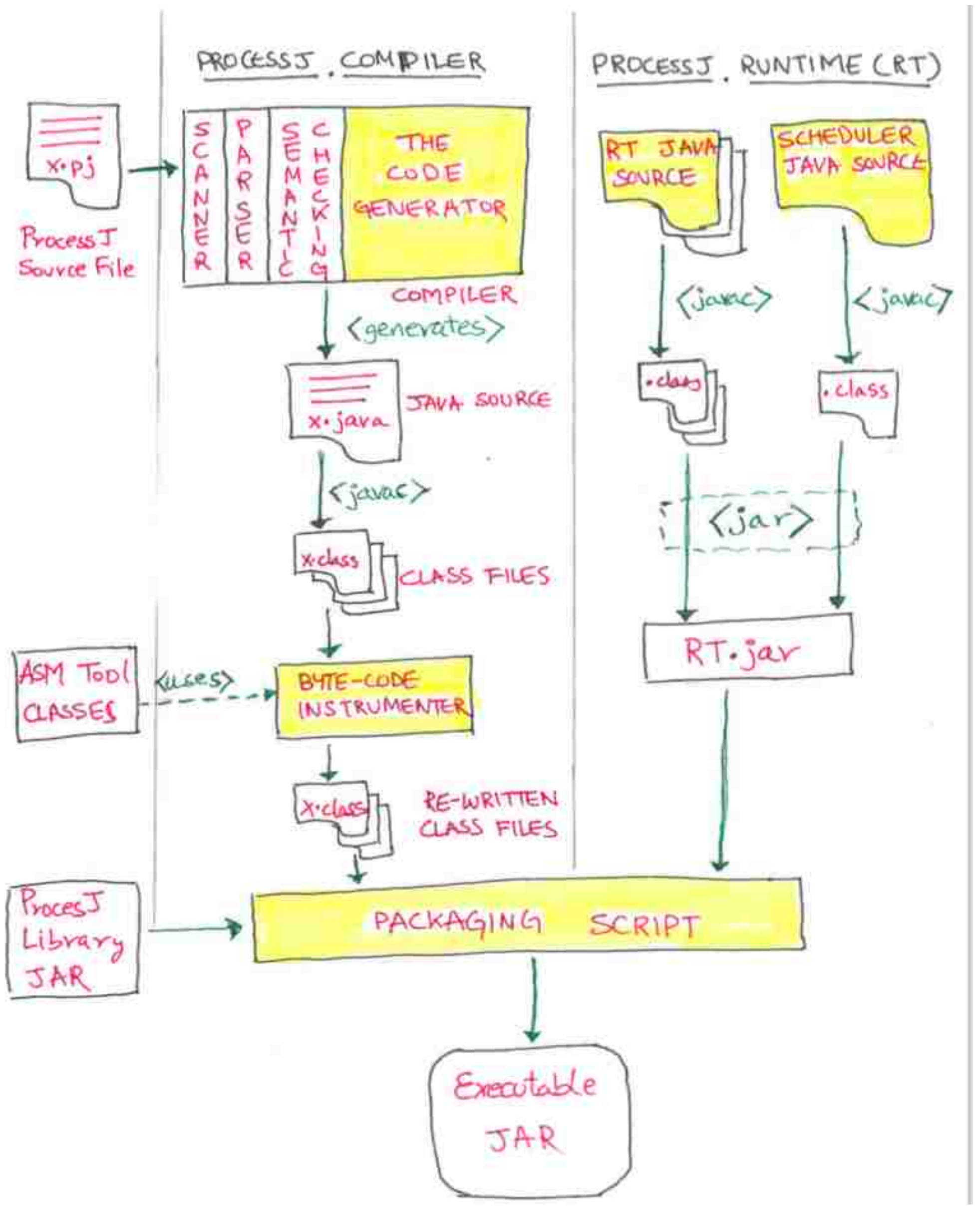

Figure 1.1: Outline of the ProcessJ source code conversion to an executable jar file.

# Chapter 2

# Background

This section gives the background on necessary concepts and design decisions used for ProcessJ.

## 2.1 Communicating Sequential Processes (CSP)

Process-oriented design can not be discussed without giving some insight on Communicating Sequential Processes (CSP). CSP [Hoa78] is, in the most basic sense, a style of notation borne of the need to accurately describe the interactions that may occur between concurrent agents. It is a process algebra that describes process composability through the use of parallel primitive. It is an explicit message passing design based on communication and is sufficiently expressive to enable reasoning about deadlock and livelock using tools like FDR.

It encapsulates fundamental principles of communication. It is semantically defined in terms of structured mathematical model. However, we do not need to be mathematically sophisticated to work with CSP. That sophistication is pre-engineered into the model. We benefit from this simply by using it. ProcessJ adopts the concurrency primitives of CSP in its grammar. CSP provides two classes of primitives in its process algebra: *events* that represent communications or interactions and *primitive processes* that represent fundamental behaviors, like STOP (the process that communicates nothing, also called deadlock), and SKIP (which represents successful termination). Let us look at the notational representation of some events in CSP.

**Event Prefixing:** CSP uses the $\rightarrow$ operator, called the prefix operator, to link events and processes. `e` $\rightarrow$ `P` performs event `e` and then behaves like process `P`.

**Parallel Composition:** The composition of two processes, say `P` and `Q`, usually written `(P || Q)`, is the key primitive distinguishing the process calculi from sequential models of computation. It is called a parallel composition. Parallel composition allows computation in `P` and `Q` to proceed simultaneously and independently. Moreover, it also allows interaction, that is synchronization and flow of information from `P` to `Q` (or vice versa) on a communication primitive, a channel, shared by both. Crucially, a process can be connected to more than one channel at a time, and, as we shall see, itself choose which, if any, channel to engage on.

**Communication:** Inter-process communication and synchronization via message passing is fundamental to concurrency with CSP. Message passing is done through primitives called the *channels*. Channels modeled by CSP are inherently synchronous: a process waiting to read, or receive data, from a channel will block until the data is sent, or written, by the writer. This is also called the *rendezvous behavior*. The same is true for a writer; not until the reader is ready will the writer continue execution.

In classical CSP, communication is simply synchronization on an event (the channel), but a message can be exchanged by writing `c.v`, where `c` is the channel and `v` is the value. For an event (or communication) `c.v` to complete, any other process that can engage on event c must do so. For a typical one-to-one channel communication that means that two processes must both perform the event `c` or `c.v`. To be useful in programming, it is possible to denote the direction in which the message flows: `c?v` means reading a value `v` from a channel `c`, and `c!v` means writing a value `v` to a channel `c`.

With this we can write CSP code like this:

```
P(x) = c!x → P(x+1)
```

Which is a process `P` that, when started, is given an initial value for `x`, communicates `x` on `c` and recurses with `x` incremented by one. If we also define

```
Q = c?y → Q
```

We can execute `P` and `Q` concurrently:

```
P(1) || Q
```

CSP supports regular if-statements and arithmetic, but we still have to introduce one more concept, namely *external choice*, or alts. This is one of the corner stones of CSP; an external choice written

```
P [ ] Q
```

offers the internal events of `P` and `Q`, both of which must be processes. This allows us two run two sending processes and one reading process that multiplexes between the two senders:

```
P1(x) = c1!x → P1(x+1)
P2(x) = c2!x → P2(x+2)
Q = c1?y → Q [] c2?y → Q
(P1 || P2) || Q
```

The external choice operator is of course the alternation operator in ProcessJ. Much more could be said about CSP, but this covers the concept we need for understanding the bulk of the thesis (but perhaps not for doing the actual implementation, but since this is not a thesis on CSP we shall stop here).

## 2.2 Scheduling

Scheduling is the means by which the operating system, or a runtime, distributes necessary resources and processor time for execution of the processes or threads in the system. The program that performs this task is called the scheduler. It is a requirement for multi-tasking systems or concurrent/parallel systems. A scheduler maintains a run queue and schedules the items in it to run. Generally, there are two flavors of scheduling: preemptive and non-preemptive or cooperative scheduling.

### 2.2.1 Preemptive

In preemptive scheduling, the scheduler decides when a process is to cease running and a new process is to resume running [Lov03]. The intervention of execution by the scheduler is called preemption. Preemption is done based on various algorithms and policies such as time-slicing, process priority, etc.

### 2.2.2 Non-preemptive or Cooperative

In non-preemptive or cooperative scheduling, a process does not stop running until it voluntarily decides to do so [Lov03]. The act of voluntarily suspending oneself is called yielding. Process-oriented design has concurrency constructs that yield, thus allowing it to work with a simple cooperative user-level scheduling mechanism. Any CSP based programming language such as occam, including ProcessJ, uses cooperative scheduling [SWB90].

## 2.3 The ASM Library

ASM is a Java bytecode manipulation and analysis library. It can be used both for directly generating Java bytecode which will produce the appropriate class files [BLC02] and for transforming compiled Java classes [Eri11]. It is a simple, yet robust, and well designed modular API. It is used in ProcessJ for transforming and instrumenting *goto* jumps for process-resumption (to support cooperative scheduling). The main advantages of the ASM tool are as follows [Eri11]:

- It has a simple, well designed and modular API that is easy to use.
- It is well documented and has an associated Eclipse plugin.
- It provides support for the latest Java version, Java 7.
- It is small, fast, and very robust.
- Its large user community can provide support for new users.
- Its open source license allows you to use it in almost any way you want

The ASM library provides two APIs for generating and transforming compiled classes: the core API provides an *event based* representation of classes, while the tree API provides an *object based* representation [Eri11]. In this thesis, we are using the tree API as instrumenting jump addresses by finding the *label()* method invocations is easier to do with the *object based* representation of the bytecode as we were able to inspect instructions of the tree without necessarily going through one instruction at a time like we would have to do with the core API.

Listing 2.1 shows a simple Java program and Figure 2.1 shows the compiled class bytecodes of the same class.

We intend to make the various executions of the *foo()* method jump to different locations in the method based on the value of the *runLabel* variable when it is invoked multiple times. For that, we will use the ASM tool to instrument the bytecode in 2.1 to add `goto` instructions with the addresses of the *label(i)* method invocations to follow the corresponding *resume(i)* methods for necessary jumps in the execution. Section 4.3.2 discusses this in detail.

## 2.4 The StringTemplate Templating Engine

This is a Java template engine for generating source code, web pages, or any other formatted text output [Ter13]. With the code generation for ProcessJ, our intention is to generate Java code from ProcessJ

```java
public class Sample {

  public void label(int i){};
  public void resume(int i){};

  public void foo(int runLabel){
    switch(runLabel) {
      case 0: resume(0);
      case 1: resume(1);
    }
    label(0);
    System.out.println("Started at label(0)");
    label(1);
    System.out.println("Started at label(1)");
  }
}
```

Listing 2.1: A sample Java program.

source code. Therefore, we chose the StringTemplate engine to generate source code by creating the necessary translation templates.

To illustrate how ProcessJ source code is translated to Java code by the code generator with the use of StringTemplate, let us look at the barrier sync statement in Listing 2.2.

```
sync(b); // where b is a barrier type in ProcessJ
```

Listing 2.2: ProcessJ code for barrier sync.

Listing 2.3 shows the template in StringTemplate that is used to generate the code for the barrier sync in Java. The parameter values passed to the template, *barrier* and *jmp*, are replaced with their corresponding elements bounded by <>. The final rendered string will be the Java code.

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

Listing 2.3: The template in StringTemplate for *sync* code generation.

Listing 2.4 shows the code generator visitor-method for a sync statement. In line 3, the necessary template is fetched, and the values for *barrier* and *jmp* are set in lines 4 and 5, and line 7 renders the Java code seen in Listing 2.5.

```
1: public String visitSyncStat(SyncStat st) {
2:   Log.log(st.line + ": Visiting a SyncStat");
3:   ST template = group.getInstanceOf("SyncStat");
4:   template.add("barrier", st.barrier().visit(this));
5:   template.add("jmp", _jumpCnt);
6:
7:   return template.render();
8: }
```

Listing 2.4: Code generator method for barrier sync code generation.

```
b.sync(this);
this.runLabel = 2;
yield();
label(2);
```

Listing 2.5: Generated Java code for barrier sync.

## 2.5 Process-oriented Design in ProcessJ

Process-oriented design deals with processes, network of processes and various forms of synchronization and communication between them [Wel00]. It is based on CSP which defines the key concepts of process-orientation (processes, pars, channels, barriers, networks, network hierarchies, choice or alternatives, protocols and synchronization patterns [PW]). It models the problem environment as a structured network of communicating processes, with each process responsible for managing the state and behavior of the individual entities in the system. The ProcessJ runtime, the JVMCSP, has Java classes for the CSP primitives. The following CSP primitives are included in the ProcessJ runtime:

- **Processes:** in ProcessJ form the basic building blocks for complex systems. Processes compose using par blocks and synchronize through channels and barriers. Considered individually, a process is just an independent sequential program that is in charge of its own state and behavior; in addition, a process decides itself when to engage with its environment  this is contrary to concurrency in an object oriented system in which threads execute code in objects. In process-oriented programming a process determines when it wishes to engage in synchronizing events. Therefore, a process is a component that encapsulates some data structures and algorithms for manipulating its data. Both its data and algorithms are private. The simplest form of interaction between processes is done by using explicit message passing with channels.

- **Channels:** The simplest mechanism for inter-process communication is reading and writing data across channels. Channels in ProcessJ are synchronous, unidirectional and unbuffered. Channels have a reading end and a writing end, and these ends can also be shared between one or more processes. Based on which ends are shared, there are four types of channels, namely one-2-one, one-2-many, many-2-one and many-2-many.

- **Barriers:** Barriers are multi-way *synchronization points* in a ProcessJ program. Processes enrolled on barriers all need to synchronize on a barrier for all processes to continue.

- **Alternatives or alts:** Non-determinism is a factor in many real-life applications where the visible outcome is a function of the order in which events happen. An alt statement is a method of introducing non-determinism in ProcessJ. An alt statement consists of a number of *guarded* statements. To execute an alt, each guard is evaluated, and of the guards that are ready, one is chosen at random and its corresponding statement is executed. The guards of an alt statement can be channel reads, timeouts, and skips.

- **Par:** A par block in ProcessJ is used to run the code in it concurrently. It is simply a block of code with the keyword par pre-fixed.

- **Par for:** A block of code can be parallelized a number of times using the `par for` loop. It is similar to a `for` loop with the par keyword pre-fixed. The content of the `par for` block gets parallelized by the number of iterations in the *for loop*.

- **Timers:** A timer in ProcessJ is an ever-ticking clock. A timer can be read much like a channel. Timers can also be used by a process to timeout or sleep for a specified period of time.

In addition, ProcessJ has two type constructors inspired by occam:

- **Protocols:** The protocol type constructor in ProcessJ has many similarities to a union data type in C. It consists of a number of tag-named variable lists, but like records, it can inherit from other protocol types. The idea of a protocol type is typically to serve as a datatype, a protocol, for channel communication. Protocol values can only be accessed in a switch statement to avoid invalid data values.

- **Records:** A record in ProcessJ is much like a struct in C, except the extends part, which lets a new record inherit an existing records fields.

### 2.5.1 Targets

ProcessJ is a multi-backend programming language. Current targets are Linux binaries using C and the CCSP [Moo99] runtime and the JVM.

```
public class Sample
  public Sample();
    Code:
      0: aload_0
      1: invokespecial "< init >":()V
      4: return

  public void label(int);
    Code:
      0: return

  public void resume(int);
    Code:
      0: return

  public void foo(int);
    Code:
      0: iload_1
      1: lookupswitch  // 2
        0: 28
        1: 33
        default: 38

      28: aload_0
      29: iconst_0
      30: invokevirtual resume:(I)V
      33: aload_0
      34: iconst_1
      35: invokevirtual resume:(I)V
      38: aload_0
      39: iconst_0
      40: invokevirtual label:(I)V
      43: getstatic java/lang/System.out:Ljava/io/PrintStream;
      46: ldc String Started at label(0)
      48: invokevirtual java/io/PrintStream.println:(Ljava/lang/String;)V
      51: aload_0
      52: iconst_1
      53: invokevirtual label:(I)V
      56: getstatic java/lang/System.out:Ljava/io/PrintStream;
      59: ldc String Started at label(1)
      61: invokevirtual java/io/PrintStream.println:(Ljava/lang/String;)V
      64: return
```

Figure 2.1: Bytecode for the sample program.

# Chapter 3

# Related Work

## 3.1 Process resumption and cooperative scheduling on the JVM

Concurrency on the JVM is typically achieved through threads which map directly to operating system threads. Communication between threads on the JVM is done through shared memory and coordination is achieved though the use of wait, notify, and join. If the threading abstraction of a runtime relies on the operating system, then the scheduling is dependent on the operating system as well. The threading abstraction on the JVM is not user controllable, and it maps to operating system threads. Operating system threads are typically heavyweight and the number of such threads that can be allocated is relatively small (typically in the order of tens of thousands only). ProcessJ is CSP based, and the process oriented programming model is built on the ability to compose smaller processes to build bigger processes. Typically, the base processes are small and it is not uncommon to write process oriented software that comprise millions of processes, something that would not be possible if a process mapped directly to an operating system thread.

A resumable mobile process is defined in [PK09] as a process that can be temporarily terminated by a programmer-inserted `suspend` statement and the control returned to the caller. At some later time, it can be restarted at the instruction immediately following the suspend point and with the exact same local state (i.e., all local variables contain the same values as they did when the method was suspended). That paper explores the resumability of a mobile process on a single or different JVMs. In this thesis we do not consider process mobility, but the byte code rewriting techniques used in [PK09] can be directly transferred to cooperatively scheduled processes on the JVM.

In [PK09] cooperative scheduling was not considered, but explicit suspend statements were inserted in the code by the programmer. The behavior of these suspend statements can be mimicked by the synchronizing statements in ProcessJ, and therefore can serve as natural yield points for a process. In [PK09] the control flow is directed to the suspend point upon resumption by the use of a `switch` table with cases representing the `suspend` statements in the source code. This switch statement appears at the start of each procedure. Since Java does not have a programmer-level `goto` statements, the switch cases are filled with dummy method-calls that later get rewritten to be bytecode level gotos to the code immediately following the yield points. We have adopted a very similar technique in this thesis: we generate dummy method-calls to a resume() method in the cases of the switch, the yield points are denoted by dummy methods yield() and label() methods; these dummy method invocations are then analyzed by the ASM tool and replaced with gotos, nops, and returns.

A simple cooperative scheduler for ProcessJ was designed in [PS14]. We have used a very similar scheduler with a few modifications such as adding a timer queue for timer timeouts and an inactive process list for runtime deadlock detection logic for this thesis. [PS14] also introduces the idea used in [PK09] as a possible solution for implementing a cooperative non-preemptive scheduled CSP runtime on the JVM. With detailed examples of CSP primitives in Java, it suggests how Java code should be generated for ProcessJ source code. A technique for preserving and restoring local state of a process using an activation record is also introduced in [PS14]. However, for the ease of writing the code generator and for maintaining simplicity in the generated code, we chose to turn all locals and parameters to fields. As the run queue holds instance of a process class, the local state is automatically retained this way. To make sure turning locals and parameters into fields did not cause any unwanted performance degradation, we ran tests that showed the difference of accessing fields over locals carries an overhead of only around 1.0% to 2.5%. We borrowed the idea of turning locals to fields from [SP11]. [SP11] mainly explores mobile process resumption without bytecode rewriting unlike in [PK09]. It introduces the idea of using a `switch` table with a case holding the block of code until an explicit `suspend` statement and another case holding the block of code after `resume` statement, essentially creating a control flow structure. The process stores the control flow map, an integer value on which the switch table jumps on, and the local state as fields. When a mobile process is passed on to another JVM by serializing it through a channel, it is resumed at the correct location using the integer jump value by the switch table thus executing the code after the `resume` statement.

However, we have decided to use the techniques describe in [PK09] and further elaborated in [PS14] for its simplicity in generating the correct code.

## 3.2 Other Process Oriented Languages or Libraries

### 3.2.1 occam / occam-$\pi$ and the CCSP runtime

The occam language was created by INMOS as a de-facto language for the Transputer. occam-$\pi$ was later developed by the University of Kent with the Kent Retargetable occam Compiler (KRoC) that included the process mobility features of the $\pi$-calculus. Both these languages are general purpose process-oriented languages based on CSP.

The design of the CSP primitives in ProcessJ closely matches with occam-$\pi$ constructs. However, occam-$\pi$'s syntax is outdated and clunky. For example, code indentation matters, and the the keywords in the language are in all upper-case. occam-$\pi$ also only runs on a few Linux distributions making it rather restrictive language. ProcessJ has Java-like syntax and since, we are targeting the JVM, platform independent. We think that these factors will help make the adoption of the language easy.

To give an idea of how an occam-$\pi$ program looks like, Listing 3.1 shows a procedure with a par block written in the language. The code snippet defines a procedure named *main()* using the `PROC` keyword that does not take in any parameters. It also declares three channels using the `CHAN` keyword, namely *knock* that carries boolean data, and *reader.a* and *writer.b* that carry integer data and have sharing write-end denoted by `SHARED !`. With the use of `PAR` keyword in line 6, it declares a parallel block that runs the invocations of *foo()* and *bar()* parallely.

```
1: PROC main()
2:   SEQ
3:     CHAN BOOL knock;
4:     SHARED ! CHAN INT reader.a:
5:     SHARED ! CHAN INT writer.b:
6:     PAR
7:       foo(reader.a?, writer.b!)
8:       bar(knock!)
:
```

Listing 3.1: Sample occam-$\pi$ code for a procedure with a parallel block.

### 3.2.2 JCSP

JCSP [Wel99, WA99] is a Java library that enables Java programmers to use process oriented design in Java. Using the primitives, extension, and wrappers [Wel99] in the library, concurrent systems can be implemented.

A sample of JCSP code can be seen in Listing 3.2. In the sample, we see a process class *foo* that extends the JCSP process class, *CSProcess*. The *run()* method has two channels and a parallel composition of two other processes called *bar* and *baz* constructed using the JCSP *Parallel* primitive. Similar constructs exist for all CSP primitives.

```
public class foo implements CSProcess {
  public void run() {
    One2OneChannelInt c1 = new One2OneChannelInt();
    One2OneChannelInt c2 = new One2OneCHannelInt();

    new Parallel {
      new CSProcess[] {
        new bar(c1, c2),
        new baz(c2, c1)
      }
    }.run();
  }
}
```

Listing 3.2: *JCSP* - sample code for process.

The sample code resembles closely with the ProcessJ code. However, the need to implement our own runtime with the CSP primitives instead of using JCSP constructs as proposed in [SP11] is because a JCSP process map directly to Java threads. This limits the number of processes that can be run on a single JVM which does not fit well with our goal of running millions of processes with ProcessJ on a single JVM. In addition, using libraries to retrofit a new paradigm onto an existing language can severely reduce the usability of the paradigm as certain rules of the language must be obeyed, and sometimes some constructs have a less than natural representation when implemented this way. This is perhaps not well illustrated by the example in Figure 3.1, but an alt statement in JCSP is significantly more complicated than in ProcessJ as it is required of the programmer to correctly fill arrays with guards etc.

## 3.3 Other approaches to concurrency

There are various other methods and models to do concurrency programming such as the Actor Model, multi-threading, coordination languages, Message Passing Interface (MPI), etc. But here, we only look at two of those methods: MPI and Threads/Locks.

### 3.3.1 MPI

The Message Passing Interface Standard (MPI) is a message passing library standard based on the consensus of the MPI Forum [Bla16]. The goal of the Message Passing Interface is to establish a portable, efficient, and flexible standard for message passing that will be widely used for writing message passing programs. MPI is a specification for the developers and users of message passing libraries. By itself, it is not a library, but rather the specification of what such a library should be. There are various implementations of MPI such as Open MPI [oIURC], an open source implementation developed and supported by a consortium of academic, research, and industry partners, and MPICH [mpi], the ANL/MSU Freely available portable MPI Implementation both of which work with languages such as C, C++ and Fortran. There are also bindings for Java such as MPJ Express [Bry06].

MPI primarily addresses the message-passing parallel programming model: data is moved from the address space of one process to that of another process through cooperative operations on each process. Originally, MPI was designed for distributed memory architectures, which were becoming increasingly popular at that time (1980s - early 1990s) [Bla16]. As architecture trends changed, shared memory SMPs (Symmetric Multiprocessing) were combined over networks creating hybrid distributed memory / shared memory systems. MPI implementors adapted their libraries to handle both types of underlying memory architectures seamlessly. They also adapted/developed ways of handling different interconnects and protocols. All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs.

Just as CSP based languages, MPI too provides a higher level abstractions for concurrent programming. It can be challenging for programmers to interact directly with threads due to the complexities of thread and lock mechanism which we will discuss in the next section. However, as MPI is still implemented as a library to supplement existing languages, programmers might encounter the difficulty and the complexity, with MPI, of writing concurrent programs that do not generally infest when using an inherently concurrent programming language such as ProcessJ.

Unlike CSP based languages such as ProcessJ, MPI implementations, like Open MPI, normally do not provide explicit concurrency primitives like channels, barriers, etc. (though, MPICH does have its flavor of channels [Lab]). Communication in Open MPI is done by using a specific set of routines directly callable from the languages that are able to interface with the library (i.e. an API). The messages are sent and received by various processes in the system by using APIs such as `MPI_Send` and `MPI_Recv` with the help of a process identifier, its *rank*, that the message is meant for. Some examples of the Open MPI APIs are shown in Listing 3.3.

```
/* Initialize the infrastructure necessary for communication */
MPI_Init(&argc, &argv);

/* Identify this process */
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

/* Find out how many total processes are active */
MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

/* Send message to process #1 */
MPI_Send(buf, sizeof(buf), MPI_CHAR, 1, 0, MPI_COMM_WORLD);

/* Receive message from process #0 */
MPI_Recv(buf, sizeof(buf), MPI_CHAR, other_rank,0,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);

/* Tear down the communication infrastructure */
MPI_Finalize();
```

Listing 3.3: Examples of some Open MPI APIs.

A sample of a complete Open MPI C program is shown in Appendix D with an implementation of the parallel Mandelbrot program.

### 3.3.2 Threads/Locks

Threads and locks is another approach to concurrency taken by many languages such as C with pthreads [BL], python with pythreads [Fou], and Java with Java threads. Here we focus on the Java threads and how it behaves with the Java objects in providing concurrency.

The support for multi-threading in Java centers on synchronization, that is, coordinating activities and data access among multiple threads. The mechanism that Java uses to support synchronization is the *monitor*. Java's monitor supports two kinds of thread synchronization: *mutual exclusion* and *cooperation*. Mutual exclusion, which is supported in the Java Virtual Machine via object locks, enables multiple threads to independently work on shared data without interfering with each other. Cooperation, which is supported in the Java Virtual Machine via the wait and notify methods of class Object, enables threads to work together towards a common goal.

However, threads operate on a shared memory model using locks, mutexes and semaphores which increases the possibility of race hazards, deadlocks, and livelocks. They are certainly almost never recommended in an initial solution [OW04, Bel05]. Also, Java monitors are highly interdependent. Their semantics compose in complex ways. There is no help provided to guard against race hazards. Also, threads have no structure of its own though Java does have a `Thread` class in its standard library. There can be no threads within threads unlike with process-oriented design where we can have processes within processes forming a network of processes.

Java also models the real world with Object Oriented Paradigm (OOP) design. However, most objects are dead - they have no life of their own as described in [Wel00]. The object is invoked by an external thread of control, usually the main program thread. When there are multiple threads in the program, it gets more complicated as the object is at the mercy of any thread that sees it. Nothing can be done to prevent method invocation even if the object is not in a fit state to service it [Wel00]. The object is not in control of its life.

But we believe that the real world does not behave this way. Objects in the real world, which we will call a *process* here on forth in this section, interacts with its environment when it chooses to do so using some kind of medium, like channels. The state of a process is private. So, process-oriented design is more fit to represent the nature. Thus, we chose to go with CSP constructs for ProcessJ.

# Chapter 4

# Implementation

In this chapter we look at the implementation of the JVMCSP runtime classes and show what the grammar for the concurrency constructs in ProcessJ look like. We also show, with examples, how ProcessJ source code is translated into Java code. One of the targets for ProcessJ is the JVM, which allows us to piggyback on the platform independence that the Java language provides. However, this made it necessary for us to write our own runtime as Java only provides concurrency through threads. However, threads, as discussed in Sections 3.1 and 3.3.2, are not suitable for CSP style concurrency. For that same reason, Java CSP libraries such as JCSP, that has one-to-one mapping between its process and a Java thread, cannot be used.

ProcessJ uses a cooperative non-preemptive scheduler. This means that all processes must participate in the scheduling by relinquishing the CPU at certain time. A reasonable place for a process to yield is at any synchronization point. In Java, this means that the the *run()* method (Listing 4.2) in a process needs to return from the yielding point and be able to restart execution from that point when it is rescheduled. It also needs to preserve its state. As Java does not have explicit goto operations, there is no simple and straightforward way to do this in source code. However, it can be done at the byte code level using a bytecode rewriting tool like ASM. We explain how we have achieved this. We also consider state retention between invocations. There is a technique developed by Matthew Sowders [SP11].

## 4.1 The JVMCSP Runtime

In this section, we will look at all the Java classes that make up the JVMCSP Runtime. We also discuss how these are implemented and why, where necessary.

### 4.1.1 PJProcess

A process is a runnable class (in terms of the ProcessJ scheduler) abstracted to the *PJProcess* Java class. Any yielding procedure in ProcessJ source code is changed to a Java class that extends the *PJProcess* class as discussed in Section 4.2.2.

```
 1:  public abstract class PJProcess {
 2:    protected boolean ready = true;
 3:    protected boolean terminated = false;
 4:
 5:    public boolean isReady() {
 6:      return ready;
 7:    }
 8:
 9:    public void terminate() {
10:      terminated = true;
11:    }
12:
13:    //other methods
14:    ...
```

Listing 4.1: The *PJProcess* class.

Listing 4.1 shows the class declaration of the *PJProcess* class. The execution state of a process is given by two flags, namely the `ready` flag (line 2) and the `terminated` flag (line 3). They represent if a process is ready to run and if it has terminated, respectively.

```
15:    public static Scheduler scheduler;
16:
17:    public void schedule() {
18:      scheduler.insert(this);
19:    }
20:
21:    public abstract void run();
22:
23:    public void terminate() {
24:      terminated = true;
25:    }
26:
27:    public void finalize() {}
```

Listing 4.2: *PJProcess* - execution methods.

Processes get scheduled to run using the *schedule()* method in Listing 4.2 (line 17). A call to schedule places the process at the end of the scheduler's run queue. Each process also holds a static reference to the scheduler instance. The scheduler starts or restarts a yielded, but ready to run, process by invoking the *run()* method (line 21). Each process needs to implement the *run()* with the body of the ProcessJ procedure that it represents. As the *run()* method does not take any arguments, all the parameters of a procedure needs to be passed into the constructor of a process. Section 4.3.1 discusses how the state of a process is preserved for resumption by changing all constructor parameters and locals to fields. After a process terminates, by calling the *terminate()* method (line 23) at the end of its *run()* method, the scheduler invokes the *finalize()* method (line 27). Processes may override it if any clean-up is needed. For example, a process in a par block (see Section 4.2.3) will need to decrement the number of processes in it so that the par block process can eventually be rescheduled to run when the count reaches 0.

```
28:    public synchronized void setReady() {
29:      if (!ready) {
30:        ready = true;
31:        scheduler.inactivePool.decrement();
32:      }
33:    }
34:
35:    public void setNotReady() {
36:      if (ready) {
37:        ready = false;
38:        scheduler.inactivePool.increment();
39:      }
40:    }
```

Listing 4.3: *PJProcess* - 'ready' flag setter methods.

A process in ProcessJ facilitates cooperative scheduling in two parts; by allowing itself to be set ready or not ready and by implementing a mechanism for yielding and doing so voluntarily. The first part can be done by using the *setReady()* and *setNotReady()* methods as shown in Listing 4.3 in lines 28 and 35, respectively. The scheduler also maintains a pool of inactive processes (see Section 4.1.12.3) which is used to detect deadlock in the system. The count in the pool needs to be changed accordingly in lines 31 and 38.

An important point to note here is that the *setReady()* method is *synchornized* and the implementation of the abstract method, *run()*, in Listing 4.2, also needs to be *synchronized*. The reason for this might not be obvious as there is a third element involved here, the *TimerQueue* (Section 4.1.12.2). A process with a timeout statement (see Section 4.2.6) starts the timer it its *run()* method and yields, setting itself not ready to run. Once the timeout value has expired, the *TimerQueue* wakes up the process by calling the *setReady()* method on it. Consider a case where the timeout value is extremely low (in 10s of nanoseconds) or zero. It is highly probable for the *TimerQueue* to call *setReady()* before the process gets to calling *setNotReady()* in its *run()* method. When execution, unfortunately, follows this sequence, the process overwrites the ready value to not ready, yields and waits to be awoken. But in the perspective of the *TimerQueue*, it has already done its part of waking the process up. So, the process never resumes, causing deadlock in the system. By synchronizing both the *run()* method and the *setReady()* method of a process, we ensure that the latter can only be invoked on it (by the *TimerQueue* in this case) after the former has completed, thus preventing any deadlocks from occurring.

```java
    protected int runLabel = 0;

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

Listing 4.4: *PJProcess* - resumption placeholder methods.

The second part, a mechanism for yielding, to support cooperative scheduling is implemented with the help of the combination of placeholder methods and the byte code rewriting technique. The methods *yield()*, *resume()* and *label()*, in Listing 4.4, are the placeholders for the yielding point, jump-from and jump-to locations respectively. The *runLabel* field holds the switched-on value used by the process to find the correct address of the starting/resumption point when it is rescheduled to run (Section 4.2.2). In addition, the methods *label(x)* and *resume(x)* always come in pairs with the same integer value, *x*, so that a jump source and its destination can be matched. See Section 4.3.2 for how the ASM tool is used to instrument these resumption points with their respective addresses.

### 4.1.2 PJChannel

Channels in ProcessJ are abstracted to the templated runtime class *PJChannel*$< T >$ (Listing 4.5). The template type defines the type of data that the channel will carry. Although the ProcessJ grammar does have channel end primitives for read and write ends, the runtime only has the *PJChannel* class, a whole channel, that is composed of both read and write functionalities. There are 4 different kinds of channels, namely PJOne2OneChannel, PJMany2OneChannel, PJOne2ManyChannel and PJMany2ManyChannel, based on which ends (write/read), in ProcessJ source code, are being shared (see Section 4.2.4). They are discussed in sections after this. Listing 4.5 shows the class and field declaration for the *PJChannel* class.

```
1:  public abstract class PJChannel<T> {
2:    protected final static int TYPE_ONE2ONE = 0;
3:    protected final static int TYPE_ONE2MANY = 1;
4:    protected final static int TYPE_MANY2ONE = 2;
5:    protected final static int TYPE_MANY2MANY = 3;
6:
7:    protected int type;
8:    protected T data;
9:    protected boolean claimed = false;
10:   protected boolean read-ready = false;
11:   protected boolean reservedForAlt = false;
12:   protected PJProcess reservedForReader = null;
13:
14:   public abstract void write(PJProcess p, T item);
15:   public abstract T read(PJProcess p);
16:   public abstract T readPreRendezvous(PJProcess p);
17:   public abstract void readPostRendezvous(PJProcess p);
18:   public abstract void addReader(PJProcess p);
19:   public abstract void addWriter(PJProcess p);
20:
21:   //other methods
22:   ...
```

Listing 4.5: The *PJChannel* class.

The *PJChannel* class holds the following fields:

- Channel type constants.
- A `data` field to hold the value to be sent over the channel.
- A `read-ready` flag to control both read/write readiness of the channel.
- A `claimed` flag to denote if a shared channel end is claimed by a process.
- A `reservedForAlt` flag to denote if the channel is reserved for read in an alt.
- A `reservedForReader` field to hold the instance of the process that the read data is reserved for.

All the basic functionalities of a channel are abstracted (lines 14-19) as their implementation will depend on the type of the channel.

```
23:   synchronized public boolean isReadyToWrite() {
24:     return !read-ready;
25:   }
26:
27:   synchronized public boolean isReadyToRead(PJProcess p) {
28:     if (read-ready && reservedForReader != null) {
29:       return (reservedForReader == p);
30:     } else if (reservedForAlt) {
31:       return false;
32:     } else {
33:       return read-ready;
34:     }
35:   }
36:
37:   synchronized public boolean isReadyToReadAltAndReserve() {
38:     if (read-ready && !reservedForAlt
39:               && reservedForReader == null) {
40:       reservedForAlt = true;
41:       return true;
42:     } else {
43:       return false;
44:     }
45:   }
```

Listing 4.6: *PJChannel* - status check methods.

Processes cannot write to a channel if data has already been written on it since the channels in ProcessJ are unbuffered. Similarly, data can only be read if it exists or in other words, if it has already been written. This requires proper status checkings on the channel before trying to read or write on it.

The *PJChannel* class has implemented the status-check methods with the help of one or more of its flags (Listing 4.6). The *isReadyToWrite()* method (line 31) returns true if the channel holds no data.

However, we need to be careful when checking for the readiness for a read operation. Channel reads can be invoked in a procedure body (in par block for example) or as alt block guard expressions. If a channel read expression is used as an alt guard, some mechanism of reserving the data is needed. Listing 4.6 shows the method, *isReadyToRead()*, necessary for checking read readiness of an unshared channel and the method *isReadyToReadAltAndReserve()* for checking readiness of shared channels and making necessary reserves on them when data is available. The need for reserving data on a ready channel used in an alt guard expression is that the readiness check and the data read happens separately in a sequential manner. The alt class checks the readiness and the process class reads the data. But there is a possibility that some other process that holds the shared read end of the same channel might get to the data before the original process does. We want to make sure that the process that invokes the readiness check in an alt is the same process that gets the data.

```java
  synchronized public boolean claim() {
    boolean success = false;
    if (!this.claimed) {
      this.claimed = true;
      success = true;
    }
    return success;
  }

  synchronized public void unclaim() {
    this.claimed = false;
  }
}
```

Listing 4.7: *PJChannel* - claim methods.

For safe usage of shared ends of channels, they need to be claimed. The *PJChannel* class has the necessary methods, *claim()* and *unclaim()*, to do this (see Listing 4.7).

### 4.1.3 PJOne2OneChannel

Listing 4.8 shows the declaration of the `PJOne2OneChannel` class which extends the base abstract class `PJChannel`. It has reader and writer process reference variables that holds the reader end and the writer end of it respectively.

The read and write methods that were abstracted in the `PJChannel` class are implemented here as seen in Listing 4.9 and Listing 4.10. The *write()* method takes in a reference to the writer process and the data to write both of which are set to the variables of the class. Since all channels in ProcessJ are unbuffered and channel operations are synchronizing events, the writer cannot move on after having written the data on the channel until a reader has read it. So, the writer is set not ready to run and if a reader already exists, it is set ready to run as it will have been set not ready to run previously; again since channel operations are synchronizing, a reader cannot move on without having read the data for which a writer first needs to write something. After the data is read in the *read()* method, both the reader and the writer processes are released and a single channel operation is complete.

```java
public class PJOne2OneChannel<T> extends PJChannel<T> {
  private PJProcess writer = null;
  private PJProcess reader = null;
  public PJOne2OneChannel() {
    this.type = TYPE_ONE2ONE;
  }
  @Override
  synchronized public void addReader(PJProcess p) {
    reader = p;
  }
```

Listing 4.8: The *PJOne2OneChannel* class.

```java
  @Override
  synchronized public void write(PJProcess p, T item) {
    data = item;
    writer = p;
    writer.setNotReady();
    ready = true;
    if (reader != null) {
      reader.setReady();
    }
  }
```

Listing 4.9: *PJOne2OneChannel* - write method.

ProcessJ also allows `extended rendezvous` for channel reads. An extended rendezvous is a statement or a block of statements that gets executed after the read on a channel has taken place but before the writer processes are released. Listing 4.11 shows the necessary methods to accomplish this. The *readPreRendezvous()* method is called by a process to read the data and it executes the extended rendezvous code in it before calling the *readPostRendezvous()* method and completing the channel operation. It is essentially the *read()* method in Listing 4.9 broken down into two methods.

```
  @Override
  synchronized public T read(PJProcess p) {
    ready = false;
    writer.setReady();
    writer = null;
    reader = null;
    T myData = data;
    data = null;
    return myData;
  }
```

Listing 4.10: *PJOne2OneChannel* - read method.

```
  @Override
  synchronized public T readPreRendezvous(PJProcess p) {
    T myData = data;
    data = null;
    return myData;
  }
  @Override
  synchronized public void readPostRendezvous(PJProcess p) {
    ready = false;
    writer.setReady();
    writer = null;
    reader = null;
  }
}
```

Listing 4.11: *PJOne2OneChannel* - extended rendezvous.

### 4.1.4 PJMany2OneChannel

Listing 4.12 shows the declaration of `PJMany2OneChannel` class which extends the base abstract class `PJChannel`. It has a single reader process reference variable and a list for references of the writer processes.

```java
public class PJMany2OneChannel<T> extends PJChannel<T> {
  private PJProcess reader = null;
  private LinkedList<PJProcess> writers =
                                    new LinkedList<PJProcess>();
  public PJMany2OneChannel() {
    this.type = TYPE_MANY2ONE;
  }
  @Override
  synchronized public void addWriter(PJProcess p) {
    writers.add(p);
  }
  @Override
  synchronized public void addReader(PJProcess p) {
    reader = p;
  }
```

Listing 4.12: The *PJMany2OneChannel* class.

It has shared writing end which means there can be multiple writers. The *read()* and the *write()* methods, shown in Listing 4.13 and Listing 4.14, work similar to the `PJOne2OneChannel` except there is some extra logic to add and remove the writers from its list.

```java
  @Override
  synchronized public void write(PJProcess p, T item) {
    ready = true;
    data = item;
    writers.addFirst(p);
    p.setNotReady();
    if (reader != null) {
      reader.setReady();
    }
  }
```

Listing 4.13: *PJMany2OneChannel* - write method.

It also has the necessary methods for the `extended rendezvous` operation as shown in Listing 4.15.

```
  @Override
  synchronized public T read(PJProcess p) {
    T myData = data;
    data = null;
    ready = false;
    if (writers.size() > 0) {
      PJProcess writer = writers.removeFirst();
      writer.setReady();
    }
    return myData;
  }
```

Listing 4.14: *PJMany2OneChannel* - read method.

```
  @Override
  synchronized public T readPreRendezvous(PJProcess p) {
    T myData = data;
    data = null;
    return myData;
  }
  @Override
  synchronized public void readPostRendezvous(PJProcess p) {
    ready = false;
    if (writers.size() > 0) {
      PJProcess writer = writers.removeFirst();
      writer.setReady();
    }
  }
}
```

Listing 4.15: *PJMany2OneChannel* - extended rendezvous.

### 4.1.5 PJOne2ManyChannel

Listing 4.16 shows the declaration of the `PJOne2ManyChannel` class which extends the base abstract class `PJChannel`. This is a channel with shared reading ends. Thus, it has a list of reader references.

```java
public class PJOne2ManyChannel<T> extends PJChannel<T> {
  private PJProcess writer = null;
  private LinkedList<PJProcess> readers =
                                new LinkedList<PJProcess>();

  public PJOne2ManyChannel() {
    this.type = TYPE_ONE2MANY;
  }

  @Override
  synchronized public void addReader(PJProcess p) {
    readers.add(p);
  }
```

Listing 4.16: The *PJOne2ManyChannel* class.

Just like `PJMany2OneChannel`, the read and write methods (Listing 4.17 and Listing 4.18) has some extra code to maintain the readers list.

```java
  @Override
  synchronized public void write(PJProcess p, T item) {
    data = item;
    writer = p;
    writer.setNotReady();
    ready = true;
    if (readers.size() > 0) {
      PJProcess reader = readers.removeFirst();
      reservedForReader = reader;
      reader.setReady();
    }
  }
```

Listing 4.17: *PJOne2ManyChannel* - write method.

The `extended rendezvous` related methods are shown in Listing 4.19 which work as described in `PJOne2OneChannel`.

```
  @Override
  synchronized public T read(PJProcess p) {
    ready = false;
    reservedForReader = null;
    reservedForAlt = false;
    writer.setReady();
    T myData = data;
    data = null;
    return myData;
  }
```

Listing 4.18: *PJOne2ManyChannel* - read method.

```
  @Override
  synchronized public T readPreRendezvous(PJProcess p) {
    T myData = data;
    data = null;
    return myData;
  }

  @Override
  synchronized public void readPostRendezvous(PJProcess p) {
    ready = false;
    reservedForReader = null;
    reservedForAlt = false;
    writer.setReady();
  }
}
```

Listing 4.19: *PJOne2ManyChannel* - extended rendezvous.

### 4.1.6 PJMany2ManyChannel

Listing 4.20 shows the declaration of the `PJMany2ManyChannel` class which extends the base abstract class `PJChannel`. This is a channel that has both reading and writing ends shared. It maintains two lists for the multiple readers and writers. The read and write methods in Listing 4.21 does some extra work to manipulate the reader and writer list. The `extended rendezvous` work similar to all other types of channels and are shown in Listing 4.22.

```java
public class PJMany2ManyChannel<T> extends PJChannel<T> {
  private LinkedList<PJProcess> readers =
                                   new LinkedList<PJProcess>();
  private LinkedList<PJProcess> writers =
                                   new LinkedList<PJProcess>();
  public PJMany2ManyChannel() {
    this.type = TYPE_MANY2MANY;
  }
  @Override
  synchronized public void addWriter(PJProcess p) {
    writers.add(p);
  }
  @Override
  synchronized public void addReader(PJProcess p) {
    readers.add(p);
  }
```

Listing 4.20: The *PJMany2ManyChannel* class.

```java
  @Override
  synchronized public void write(PJProcess p, T item) {
    ready = true;
    data = item;
    writers.addFirst(p);
    p.setNotReady();
    if (readers.size() > 0) {
      PJProcess reader = readers.removeFirst();
      reader.setReady();
    }
  }
  @Override
  synchronized public T read(PJProcess p) {
    T myData = data;
    data = null;
    ready = false;
    if (writers.size() > 0) {
      PJProcess writer = writers.removeFirst();
      writer.setReady();
    }
    return myData;
  }
```

Listing 4.21: *PJMany2ManyChannel* - read/write methods.

```
  @Override
  synchronized public T readPreRendezvous(PJProcess p) {
    T myData = data;
    data = null;
    return myData;
  }

  @Override
  synchronized public void readPostRendezvous(PJProcess p) {
    ready = false;
    if (writers.size() > 0) {
      PJProcess writer = writers.removeFirst();
      writer.setReady();
    }
  }
}
```

Listing 4.22: *PJMany2ManyChannel* - extended rendezvous.

### 4.1.7 PJPar

ProcessJ allows concurrent/parallel execution of code by wrapping them in a par block (see Section 4.2.3). The *PJPar* Java class (Listing 4.23) is the runtime element that makes this possible.

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

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

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

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

Listing 4.23: The *PJPar* class.

Each statement in a par block will be treated as a process and scheduled to run. The process in which a par block is executed, say process *p1*, will remain not ready to run until all the processes in its par block have terminated. This behavior is implemented with the combination of the number of processes in the par block, the `processCount` field, and an instance of the process *p1*, the `process` field. These values are passed as the *PJPar* constructor parameters by *p1*. Each process in the par block will invoke the *decrement()* method upon termination (in their *finalize()* method), eventually bringing the `processCount` to zero and *p1* will be ready to run again.

### 4.1.8 PJTimer

Timers in ProcessJ are converted to the *PJTimer* class in the generated Java code (Listing 4.24). This class implements the Java concurrency library interface, *java.util.concurrent.Delayed*, to correctly work with the *TimerQueue* (see Section 4.1.12.2).

```java
public class PJTimer implements Delayed {
  private PJProcess process = null;
  private long delay = 0L;
  public final long timeout = 0L;
  public boolean started = false;
  public boolean expired = false;
  private boolean killed = false;

  public PJTimer(PJProcess process, long timeout) {
    this.process = process;
    this.timeout = timeout;
  }

  // more methods
  ...
```

Listing 4.24: The *PJTimer* class.

The state of a timer object is given by the fields `started`, `expired` and `killed` which are self-explanatory by name. However, the `killed` flag does have a special purpose as can be seen in Listing 4.25.

```java
  public void kill() {
    killed = true;
  }

  public PJProcess getProcess() {
    if (killed) {
      return null;
    } else {
      return process;
    }
  }
```

Listing 4.25: *PJTimer* - the `killed` flag.

A process with a timer timeout call will set itself not ready to run after the timer has been started in the timer queue. When the timeout period ends, the timer queue gets a reference to the process by calling the *getProcess()* method on the timer and sets the process ready to run. If, for some reason, that process chooses not to wait till the end of the timeout period, for example in alt blocks (see Section 4.2.7), the timer can be killed and the process can be set ready to run by itself or someone other than the timer queue. However, the timer queue still has to let the timer timeout as its queue, the Java concurrency interface, DelayQueue, to be specific, does not have a way to remove and discard its elements. In a situation like this, we do not want the timer queue to set a process ready for a killed timer object as the process might have moved on and yielded at some other point. A check on the `killed` flag in line 21 makes sure this does not happen.

```java
public void start() throws InterruptedException {
  this.delay = System.currentTimeMillis() + timeout;
  PJProcess.scheduler.insertTimer(this);
  started = true;
}

public void expire() {
  expired = true;
}

public static long read() {
  return System.currentTimeMillis();
}
```

Listing 4.26: *PJTimer* - normal control methods.

Listing 4.26 shows the *start()* method that inserts the timer to the timer queue through the scheduler. It sets the delay value to be the timeout value from the current time. The *expire()* method is invoked by the timer queue when the timeout ends normally and the method *read()* simply returns the current time as timers can also be read like channels to get the current time.

As mentioned earlier, this runtime class help with the the delay countdown mechanism by implementing the *Delayed* Java interface, specifically, the methods *getDelay()* and *compareTo()* in it (Listing 4.27). The timer queue uses the values returned by these methods to decide what has timed-out and to move elements around in the queue as needed, respectively.

### 4.1.9 PJAlt

An alternative, or alt block, in a process-oriented language is a way for a process to wait on a number of guarded events and choose any one of them arbitrarily from among the ready ones.

```java
  @Override
  public long getDelay(TimeUnit unit) {
    long diff = delay - System.currentTimeMillis();
    return unit.convert(diff, TimeUnit.MILLISECONDS);
  }

  @Override
  public int compareTo(Delayed o) {
    if (this.delay < ((PJTimer) o).delay) {
      return -1;
    }
    if (this.delay > ((PJTimer) o).delay) {
      return 1;
    }
    return 0;
  }
}
```

Listing 4.27: *PJTimer* - the Delayed interface implementation.

However, for the ease of implementation, we, in ProcessJ, have written a runtime class, *PJAlt* (Listing 4.28), that always chooses the first ready guard removing the arbitrary nature of it. It, essentially, functions like a pri-alt. More on this in Section 4.2.7.

The *PJAlt* class holds a reference to the process with the alt block, an array of boolean pre-guards, and an array of guards (Listing 4.29). A guarded event/process is guarded by either a just a guard or a combination of a boolean pre-guard and a guard. All false pre-guards is illegal but all not ready guards is OK. This is necessary as an alt block needs at least one event that is ready, so having all false pre-guards will not allow that. In such cases, the invoking process needs to throw a runtime exception.

An alt guard can be a channel read, a timer timeout, or a skip statement. In Listing 4.30, we show how a ready guard is selected. The first check done is on the boolean guard (line 27) which has to be 'true' before proceeding. The following lists the conditions for the second part of each guard:

- A **skip** guard is always ready (lines 28-29).
- A **timer timeout** guard can be ready in two ways; if it has not been started and the timeout value is less than or equal to zero or if it has expired (lines 34-35). The reason for the first condition is that a timeout of zero or less time can be considered timed-out even before it starts.
- A **channel read** guard is ready when there is data to be read on it (lines 42 and 47).

```java
public class PJAlt {
  private Object guards[];
  private boolean[] bGuards;
  private PJProcess process;

  public PJAlt(int caseCount, PJProcess p) {
    this.process = p;
    this.bGuards = new boolean[caseCount];
    this.guards = new Object[caseCount];
  }
  //other methods
  ...
```

Listing 4.28: The *PJAlt* class.

```java
  public boolean setGuards(boolean[] bg, Object[] guards) {
    this.bGuards = bg;
    this.guards = guards;

    for(boolean b: bg) {
      if (b) {
        return b;
      }
    }
    return false;
  }
```

Listing 4.29: *PJAlt* - guards initialization.

The index of the first ready guard is returned without further evaluating any other guards.

```java
  public int getReadyGuardIndex() {
    int chosen = -1;
    for (int i = 0; i < bGuards.length; i++) {
      if (bGuards[i]) {
        if (guards[i] instanceof String
          && "skip".equals(guards[i])) {
          chosen = i;
          break;
        } else if (guards[i] instanceof PJTimer) {
          PJTimer t = (PJTimer)guards[i];
          if ((!t.started && t.timeout <= 0L)
              || t.expired) {
            chosen=i;
            break;
          }
        } else if (guards[i] instanceof PJChannel) {
          PJChannel c = (PJChannel) guards[i];
          if (c.isSharedRead()){
            if (c.isReadyToReadAltAndReserve()) {
              chosen = i;
              break;
            }
          } else {
            if (c.isReadyToRead(process)) {
              chosen = i;
              break;
            }
          }
        }
      }
    }
    return chosen;
  }
}
```

Listing 4.30: *PJAlt* - finding the ready guard.

### 4.1.10 PJBarrier

Barriers in ProcessJ are a synchronizing point for multiple processes, also called multi-way synchronization points. The *PJBarrier* runtime class (Listing 4.31) provides the necessary methods to *enroll*, *synchronize* and *resign* from a barrier (Section 4.2.8).

```
1:  public class PJBarrier {
2:
3:    List<PJProcess> sycned = new ArrayList<PJProcess>();
4:    public int enrolled = 0;
5:
6:    public PJBarrier() {
7:      this.enrolled = 1;
8:    }
9:
10:   //other methods
```

Listing 4.31: The *PJBarrier* class.

A barrier maintains the count of the enrolled processes and a list of process references that have synced (syncing is described at the end of this section) as seen in lines 3 and 4. The constructor (lines 6-8) sets the initial count of enrollees to ‘1’ as any process that declares a barrier is automatically enrolled on it by definition [BWS05a].

```
  public synchronized void enroll(int m) {
    this.enrolled = this.enrolled + m - 1;
  }

  public synchronized void resign() {
    if (this.enrolled > 1) {
      this.enrolled = this.enrolled - 1;
    }
  }
```

Listing 4.32: *PJBarrier* - enrolling and resigning.

Any implementation of a barrier cannot do without a way to enroll on and resign from it; hence the *enroll()* and *resign()* methods as seen in Listing 4.32. Multiple processes can be enrolled on a barrier at once by passing the total count of the enrollees as an argument, *m*, to the *enroll(m)* method. The `enrolled` count of the barrier is increased by *m* and the declaring process is temporarily resigned from it by subtracting 1 from the count. The reason for this temporary resignation is that once the declaring process enrolls other processes on to the barrier, usually in a par block (see Section 4.2.3), it cannot take part in the multi-way synchronization. Any process enrolled on a barrier automatically resigns from it upon termination. The *resign()* method is invoked in its *finalize()* method. Each resigning process decrements the `enrolled` count by 1 except the last process so that the count before and after the multi-way synchronization remains the same. That way, the barrier can be reused until the declaring process terminates and resigns from it.

```
20:   public synchronized void sync(PJProcess process) {
21:     process.setNotReady();
22:     sycned.add(process);
23:     if (sycned.size() == enrolled) {
24:       for(PJProcess p : sycned) {
25:         p.setReady();
26:       }
27:       sycned.clear();
28:     }
29:   }
30: }
```

Listing 4.33: *PJBarrier* - syncing.

A process can only synchronize on the barriers on which it is enrolled. To sync, the process invokes the *sync()* method (Listing 4.33) by passing its reference as an argument. The barrier in turn sets the process not-ready to run and adds it to the `synced` list. Only after all the enrolled processes sync, can all of them move past the synchronization point. The *sync()* method also checks if all have synced and if so, sets all of them ready to run. In addition, it clears its enrollee pool so that it can be reused (lines 23-27).

### 4.1.11 PJProtocolCase

The *PJProtocolCase* class is simply a protocol tag value holder. All cases in a ProcessJ protocol (Section 4.2.10) will be changed to a class extending the *PJProtocolCase* class.

```
public class PJProtocolCase {
  public String tag = null;
}
```

Listing 4.34: The *PJProtocolCase* class.

### 4.1.12 The Scheduler and its Components

ProcessJ, in the scope of this thesis, uses a single-threaded/single-core cooperative non-preemptive scheduler (see Section 7.1 for the future works on a multi-core scheduler). This section describes the implementation of our simple scheduler and the necessary components for its operation.

#### 4.1.12.1 The Run Queue

The *RunQueue* class (Listing 4.35) maintains a queue with instances of the *PJProcess* class that are scheduled to run. The scheduler simply works on emptying this queue by running the processes that are ready to run to completion.

```
public class RunQueue {
  private LinkedList<PJProcess> queue =
            new LinkedList<PJProcess>();

  synchronized public void insert(PJProcess p) {
    queue.addLast(p);
  }

  synchronized public PJProcess getNext() {
    return queue.removeFirst();
  }

  synchronized public int size() {
    return queue.size();
  }
}
```

Listing 4.35: The *RunQueue* class.

#### 4.1.12.2 The Timer Queue

The *TimerQueue* class (Listing 4.36) maintains a queue with instances of the *PJTimer* class that are performing a timeout operation. To delegate the task of counting down the timeout delay and retrieving the timed-out timers, this runtime class uses *java.util.concurrent.DelayQueue*$< T >$, a Java concurrency library class, as the type for its queue.

The timer queue, essentially, is another thread in the system that runs along with the scheduler thread. However, it does not contribute to the single-threadedness of the process-scheduling. It just serves to inde-

```
1:  public class TimerQueue {
2:    public static BlockingQueue<PJTimer> delayQueue =
3:                  new DelayQueue<PJTimer>();
4:
5:    public synchronized void insert(PJTimer timer)
6:                throws InterruptedException {
7:      delayQueue.offer(timer);
8:    }
9:
10:   public synchronized boolean isEmpty() {
11:     return delayQueue.isEmpty();
12:   }
13:
14:   // other methods
15:   ...
```

Listing 4.36: The *TimerQueue* class.

pendently keep track of the timer delays. The *TimerQueue* class has an instance of a Java thread (Listing 4.37. The scheduler takes care of starting or killing (interrupting) this thread. Upon start-up, it takes a timer out of the delay queue, expires the timer, and sets any process, using this timer, ready to run (lines 26-33). The *DelayQueue* instance only allows timers that have timed-out to be taken out of the queue, which at that point has 'bubbled-up' to the head of the queue. So, the timer thread waits in line 21 until one can be taken out.

```
16:   private Thread timerThread = new Thread(new Runnable() {
17:     @Override
18:     public void run() {
19:       try {
20:         while (true) {
21:           PJTimer timer = (PJTimer) delayQueue.take();
22:           timer.expire();
23:           PJProcess p = timer.getProcess();
24:           if (p != null) p.setReady();
25:         }
26:       } catch (InterruptedException e) { return; }
27:     }
28:   });
29: }
```

Listing 4.37: *TimerQueue* - thread instance.

### 4.1.12.3 Inactive Pool

The scheduler uses the *InactivePool* class (Listing 4.38) to keep track of the number of processes in the *run queue* that are not ready to run to detect deadlock in the system. Processes actively modify the `count` in the pool (see Section 4.3).

```java
public class InactivePool {
  private int count = 0;
  public synchronized void decrement() {
    this.count--;
  }
  public synchronized void increment() {
    this.count++;
  }
  public int getCount() {
    return this.count;
  }
}
```

Listing 4.38: The *InactivePool* class.

### 4.1.12.4 The Scheduler

We have a simple non-preemptive single-threaded scheduler (Listing 4.39) in ProcessJ. It maintains a *run queue* that is used for process scheduling and an inactive-pool count for runtime deadlock detection. It also takes care of starting the timer queue thread (see Section 4.36).

```java
public class Scheduler extends Thread {
  private final TimerQueue tq = new TimerQueue();
  private final RunQueue rq = new RunQueue();
  public final InactivePool ip = new InactivePool();

  // methods to insert items to the run queue and timer queue.

  // other methods
  ...
```

Listing 4.39: The *Scheduler* class.

The scheduler takes a process out of the *run queue*, as long as there is one, runs it if it is ready and if not, puts it back to the end of the queue (Listing 4.40). After a process yields, if it has terminated, its *finalize()*

method is invoked, otherwise the process is again put back at the end of the queue. Unlike preemptive schedulers, this scheduler cannot decide when a process can run or when a process should be paused (or preempted) to allow other processes to run. The cooperative part of the scheduling system needs to be provided by the processes instead by voluntarily yielding and giving the control back to the scheduler (see Section 4.1.1).

```
10:   @Override
11:   public void run() {
12:     tq.start();
13:     while (rq.size() > 0) {
14:       PJProcess p = rq.getNext();
15:       if (p.isReady()) {
16:         p.run();
17:         if (!p.terminated()) {
18:           rq.insert(p);
19:         } else {
20:           p.finalize();
21:         }
22:       } else {
23:         rq.insert(p);
24:       }
25:       if (inactivePool.getCount() == rq.size()
26:             && rq.size() > 0 && tq.isEmpty()) {
27:         System.err.println("System is deadlocked!");
28:         tq.kill();
29:         System.exit(1);
30:       }
31:     }
32:     tq.kill();
33:   }
34: }
```

Listing 4.40: *Scheduler* - run method.

The scheduler determines that a runtime deadlock has occurred if there are processes in the *run queue* but none of them are ready to run and the *timer queue* does not have any active timers that could potentially have woken of one of the processes in the *run queue* (lines 25-30).

## 4.2 Language Constructs and Code Generation

In this section, we describe the ProcessJ language constructs specific to concurrency and discuss how Java code is generated for them. Specifically, we look at what the grammar for the various language features look like, the semantic description, any static semantics related to them, and finally how they are translated to Java code coupled with the ProcessJ runtime elements required at runtime.

### 4.2.1 Basic Statements, Expressions and Loop/Condition Controls

Syntactically all the basic constructs such as statements, expressions, loops and condition controls have a one-to-one translation from ProcessJ source code to Java code. This was part of the design decisions made for the language.

### 4.2.2 Process

The process is a key construct in the process-oriented design. It is the element that is executed concurrently. A yielding procedure in ProcessJ, is changed to process; that is, a class extending the `PJProcess` class is generated for such procedure (proc). A non-yielding procedure is changed to a normal Java method. Constructs such as channels, barriers, timer timeouts, par block and alts can cause a procedure to yield.

#### Syntax

The context-free grammar of the language pertaining to a procedure/proc declaration can be seen in Figure 4.1.

$$\begin{array}{lll}
\textit{proc\_type} & \rightarrow & \textit{modifier}^{*}\ \textbf{proc type}\ \text{ID}\ \textbf{(}[\textbf{type}\ \text{ID}\textbf{, type}\ \text{ID})^{*}]\textbf{)}\ [\text{annotations}] \\
 & & [\{\ \text{statement}^{*}\ \}] \\
\textit{modifier} & \rightarrow & \textit{public} \mid \textit{private} \mid \textit{native} \mid \textit{const} \mid \textit{mobile} \mid \textit{protected} \\
\textit{annotations} & \rightarrow & \textbf{[}(\text{ID}\textbf{=}\ (\text{ID} \mid \textit{boolean\_literal} \mid \textit{numeric\_literal}))^{*}\textbf{]}
\end{array}$$

Figure 4.1: The ProcessJ grammar for a procedure.

#### Semantic Description

Listing 4.41 shows an example of a generic ProcessJ procedure. It is called *foo*, takes in two parameters *x* and *y*, and has a local declaration *b*. The parameters and the local declaration in the example will be used to show how the variable names will be converted to be unique in the generated Java code. The

`...Statements...` is a placeholder that represents one or more statements in the body of the procedure. We will be using the same placeholder in the sections that follow after this. If `...Statements...` has the use of any synchronizing constructs such as channels, barriers, par blocks, etc., this procedure will be changed to a Java process class, and extension of the PJProcess class, else it will be changed to a normal Java method by removing the `proc` keyword.

```
proc void foo(int x, double y) {
  boolean b = true;
  ...Statements...
}
```

Listing 4.41: A generic procedure in ProcessJ.

### Code Layout

Listing 4.42 shows the Java generated code for the ProcessJ procedure in Listing 4.41 assuming that it is contained in a file named *A.pj*.

The *run()* method of the `PJProcess` must be implemented as that is the method the scheduler will call in order to execute the process. It contains the body of the ProcessJ procedure. As the `run()` method does not take in any parameters, any parameters passed to the original procedure must be passed to the constructor of the class. All the parameters and locals of a procedure are changed to fields in the `PJProcess` class to retain state as discussed in Section 4.3.1. Also, it can be seen in Listing 4.42 that the parameters and the locals are renamed with *_pd$<name>* and *_ldX$<name>* respectively, where *X* in the latter is a number count of the generated names to prevent any name conflicts in the generated code. In all the sections after this, the original names will be used in the Java generated code so that it is easier to read and relate with ProcessJ code. Just remember that they will look different in the actual generated code. The *run()* method also contains a switch table at its start which holds all the resume addresses that jumps to corresponding labels.

```
public class A {
  class foo extends PJProcess {
    int _pd$x;
    double _pd$y;
    boolean _ld1$b;

    public foo(int _pd$x, double _pd$y) {
      this._pd$x = _pd$x;
      this._pd$y = _pd$y;
    }
    @Override
    public void run() {
      switch(this.runLabel) {
        case 0: break;
        case 1: resume(1); break;
        case 2: resume(2); break;
        ...
      }

      _ld1$b = true;
      ...Statements...
      terminate();
    }
  }
}
```

Listing 4.42: The generated Java process class for Listing 4.41.

### 4.2.3 Par Block

A par block in ProcessJ is used to run the code in it concurrently. It is simply a block of code with the keyword par pre-fixed.

#### Syntax

The grammar rules for regular blocks and par-blocks are shown in Figure 4.2.

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

Figure 4.2: The ProcessJ grammar for blocks and par-blocks.

#### Semantic Description

A block is implicitly *sequential*, so each statement in a block is executed in order from top to bottom. However, in ProcessJ, it is possible to declare a block *parallel*. This is done by prefixing the word `par` to the block. For example in Listing 4.43:

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

Listing 4.43: ProcessJ code for a par block.

The above example shows a par-block with two statements, namely the invocations of the functions `foo` and `bar`. A par-block makes each statement a process that runs concurrently with the other statements in the par-block, and the entire par block does not terminate until every process of the block has terminated.

It is not possible to declare variables in a par-block for a number of reasons: it is not possible to read and write the same variable in a par-block as that causes a race condition on the specific variable. If a variable can only be read or written, then a newly defined variable can only be written through its initializer or an assignment and never read, and since the variable goes out of scope at the end of the block, such an endeavor would be meaningless. The ProcessJ compiler has a parallel usage checker that checks for possible race conditions.

A par-block can enroll its processes on zero or more barriers as seen in Listing 4.44.

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

Listing 4.44: ProcessJ code for par block with enroll.

The code in Listing 4.44 executes `foo`, `bar`, and `baz` concurrently while enrolling all three processes on the barriers `b` and `c`. Note, all three processes are passed the barriers on which they are enrolled. If we did not do that they could not synchronize on the barriers, and if just one of them does not synchronize on a barrier it is enrolled on, then none of the other processes enrolled on that barrier can move past their synchronization point, and we have a potential for a deadlock (unless that particular process terminates).

### Code Layout

Listing 4.45 shows the generated Java code for Listing 4.43. It shows a simple par block with two yielding processes, *foo()* and *bar()*. An instance of `PJPar` is created and the owner process and the count for the number of processes in the par block is passed as parameter. After scheduling the two processes, the par block owner process sets itself not ready to run and yields. The *finalize()* method of the two processes in the par block is overriden to decrement the par block count after their execution is complete. When the count reaches zero, the process with the par block is set ready to run.

A par block can be enrolled on one or more barriers (Section 4.2.8). Listing 4.46 shows an example of a par block enrolled on two barriers. It is the generated Java code for Listing 4.44. The par block contains three processes and all of them are enrolled on the two barriers by passing the barriers as parameters to them. In the *finalize()* method, along with decrementing the par block count, they also need to resign from the barriers after each of their execution is complete.

```java
final PJPar par1 = new PJPar(2, this);
(new foo(){
  public void finalize() {
    par1.decrement();
  }
}).schedule();
(new bar(){
  public void finalize() {
    par1.decrement();
  }
}).schedule();
setNotReady();
this.runLabel = 1;
yield();
label(1);
```

Listing 4.45: Generated Java code for Listing 4.43.

```java
PJBarrier b = new PJBarrier();
PJBarrier c = new PJBarrier();
final PJPar par1 = new PJPar(3, this);
b.enroll(3);
c.enroll(3);
(new foo( b, c ){
  public void finalize() {
    par1.decrement();
      b.resign();
      c.resign();
    }
}).schedule();
(new bar( b, c ){
  public void finalize() {
    par1.decrement();
      b.resign();
      c.resign();
    }
}).schedule();
(new baz( b, c ){
  public void finalize() {
    par1.decrement();
    b.resign();
    c.resign();
  }
}).schedule();
setNotReady();
this.runLabel = 1;
yield();
label(1);
```

Listing 4.46: Generated Java code for Listing 4.44.

### 4.2.4 Channels

A channel in a process-oriented language is a medium for communication between two or more processes. A process can write or read data from a channel, technically, a channel-end. ProcessJ has unbuffered, blocking, synchronous channels. A sender blocks until the message has been exchanged.

Channels, by design, are unidirectional. Each channel has two ends; a reading end and a writing end. Here, we simply call them channel-ends. Typically, one process holds on to one channel end and communicates with another process holding the other end of the same channel. A channel end can be shared between two or more processes. A read/write operation on a shared channel end can only be done one at a time by making a claim (see Section 4.2.5) on it. That means that data on a shared reading end is only received by one of the processes, namely the one holding the end at that time.

#### Syntax

The grammar for channels and channel-end declaration is given in Figure 4.3.

$$\begin{array}{lcl} \textit{channel} & \rightarrow & [\ \textbf{shared}\ (\ \textbf{read} \mid \textbf{write}\ )]\ \textbf{chan} < \textit{type} > \\ \textit{channel\_end} & \rightarrow & [\ \textbf{shared}\ ]\ \textbf{chan} < \textit{type} > \textbf{.}\ (\ \textbf{read} \mid \textbf{write}\ ) \end{array}$$

Figure 4.3: The ProcessJ grammar for Channels.

A channel can have no shared ends, a shared reading end, a shared writing end or both.

#### Semantic Description

Listing 4.47 shows an example of a channel declaration in line 1, named ‘c’, that carries integer values. Note the use of the <>. Channels must be declared to carry a type, and almost any type can be used. Channels cannot be passed as parameters to a procedure, only channel-ends can be used as parameters. Though we do have mobile channel-ends, it is not covered by the scope of this thesis and so, channels cannot be communicated on other channels at the moment. Channels are auto allocated and immutable, that is, they do not need to be created by using the kyeword *new*. Other types of declarations can also be seen in Listing 4.47.

Listing 4.48 shows the read operation on a channel. We can simply invoke the *read()* method which returns the data from the channel if it exits.

The write operation of a channel can be seen in Listing 4.49. Just like ready, we simply invoke the *write()* method on the channel.

```
chan<int> c;
chan<int>.read cr;
chan<boolean>.write cw;
shared chan<int> cs;
shared write chan<long> csw;
chan<int>[] carr = new chan<int>[100];
```

Listing 4.47: Example of channel declarations.

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

Listing 4.48: Example of channel read operation.

### Code Layout

Listing 4.50 shows the generated Java code for the channel declarations in Listing 4.47. Based on which end is shared, correct Java runtime class is used to instantiate the channels.

Listing 4.51 shows the generated Java code for channel read operation in Listing 4.48. Since a channel in ProcessJ is synchronous and unbuffered, before writing, its status is checked to see if there is data to read. If none exist, it sets the process not ready to run and yield. If it successfully reads data, it yields again for fairness and returns back to the end of the read operation code when it is run next time.

Listing 4.52 shows the generated Java code for channel write operation in Listing 4.49. As with the read, the status of the channel needs to be checked to see if it is empty and data can be written on it. It it is successful in writing data, it yields for fairness and resumes after the write operation code when run next time. If it fails to write, it yields with a status not ready to run and re-checks if the channel is ready to be written on when run next time.

```
x = out.write(10);
```

Listing 4.49: Example of channel write operation.

```
PJChannel<Integer> c = new PJOne2OneChannel<Integer>();
PJChannel<Integer> cr = new PJOne2OneChannel<Integer>();
PJChannel<Boolean> cw = new PJOne2OneChannel<Integer>();
PJChannel<Integer> cs = new PJMany2ManyChannel<Integer>();
PJChannel<Long> csw = new PJMany2OneChannel<Long>();
PJChannel<Integer>[] carr= new PJOne2OneChannel<Integer>[100]();
```

Listing 4.50: Generated Java code for Listing 4.47.

```
label(1);                                  // return here is read fails
if(in.isReadyToRead(this)) {                  // check if there is data
  x = in.read(this);                                      // read data
  this.runLabel = 2;                   // set runLabel to go to label(2)
  yield();                                                    // yield
} else {
  setNotReady();                         // set process not ready to run
  in.addReader(this);                   // add the reader to the channel
  this.runLabel = 1;                   // set runLabel to go to label(1)
  yield();                                                    // yield
}
label(2);                                // return here if read succeeds
```

Listing 4.51: Generated Java code for Listing 4.48.

```
label(1);                                 // return here if write fails
if (out.isReadyToWrite()) {                // check if channel is empty
  out.write(this, 10);                                   // write data
  this.runLabel = 2;                   // set runLabel to go to label(2)
  yield();                                                    // yield
} else {
  setNotReady();                         // set process not ready to run
  this.runLabel = 1;                   // set runLabel to go to label(1)
  yield();                                                    // yield
}
label(2);                               // return here if write succeeds
```

Listing 4.52: Generated Java code for Listing 4.49.

### 4.2.5 Claim

A claim statement is used to control access to shared channel-ends. Multiple ends can be claimed at the same time by a process. This restricts any other processes from performing read/write operations on those ends until the claiming process releases them.

#### Syntax

The grammar for a claim statement is shown in Figure 4.4. A `channel_end` can be an abbreviation or a more complex channel expression as shown by the grammar part *expression.(read | write)* in Figure 4.4 such as *x[i].read* where *x* is a channel.

$$\begin{array}{lcl} \textit{claim\_statement} & \rightarrow & \textbf{claim (}\ \textit{channel\_end}\ (\ \textbf{,}\ \textit{channel\_end})^*\ \textbf{)}\ \textit{statement} \\ \textit{channel\_end} & \rightarrow & \text{ID} \\ & | & \textit{channel\_end}\ \text{ID}\ \textbf{=}\ \textit{expression} \\ & | & \textit{expression}\ \textbf{.}\ (\ \textbf{read}\ |\ \textbf{write}\ ) \end{array}$$

Figure 4.4: The ProcessJ grammar for claims.

A channel-end can be one of three things:

- an identifier denoting a variable of channel-end type.
- an expression of channel type with .write/.read.
- an abbreviation of a channel-end expression.

#### Semantic Description

Shared channel ends must be claimed before being used. The channel ends can be more complex expressions of channel-end type or abbreviations. Listing 4.53 shows an example of a claim statement with two channel ends, a read end `a`, and the write end of a channel `b`. After the execution of the block following the claim statement, the claimed channel ends are automatically released and can be used by other processes.

```
claim (a, b.write) {
   ...Statements...
}
```

Listing 4.53: ProcessJ code for claim statement.

A claim statement can be followed by a block as shown in Listing 4.53 or a single statement, represented by `...Statement...`, as shown in Listing 4.54. It can also be observed in Listing 4.54 that abbreviation needs to be done for channel ends retrieved from an array.

```
claim(shared chan<int>.read ccr = chanEnds[i])
   ...Statement...
```

Listing 4.54: Another ProcessJ example for claim statement.

### Code Layout

Listing 4.55 shows the Java code generated for Listing 4.53. The *claim()* method on each channel is called in line 2 which sets the `claimed` flag on the channel end if other processes have not already done so. The `...Statements...` is only executed if both claims are successful. If not, the ends are released and the process yields at line 5 such that it jumps to label(1) next time it is scheduled and retries to make claims on those ends. It should be noted that before the process calls the *yield()* method, it does not set itself not-ready to run, that is, it is still ready to run. Upon successful execution of `...Statements...`, the locks on the channel ends are released by calling the *unclaim()* method on them.

```
1: label(1);
2: if(!a.claim() ||!b.claim() ) {
3:   a.unclaim();
4:   b.unclaim();
4:   this.runLabel = 1;
5:   yield();
6: }
7: ...Statements...
8: a.unclaim();
9: b.unclaim();
```

Listing 4.55: Generated Java code for Listing 4.53.

In Listing 4.56, we can see the code generated for claim statements that attempt to make claims on channel ends in an array. The aliasing part is lifted off of the claim statement and a variable is set as seen on line 1.

```
PJOne2ManyChannel<Integer> ccr; //field
...
ccr = (chanEnds[i]);
label(3);
if(!ccr.claim() ) {
  ccr.unclaim();
  this.runLabel = 3;
  yield();
}
...Statement...
ccr.unclaim();
```

Listing 4.56: Generated Java code for Listing 4.54.

### Static Semantics

Nested claim statements are not allowed. With nested claims, there is a possibility for deadlock. We do not allow par/invocation in a claim for the same reason as they can have other claim statements.

### 4.2.6 Timers

A timer is an ever ticking clock. A timer can be *read* much like a channel (Listing 4.57). However, unlike channels, timer reads are not synchronized - when we read the time, we just want the latest value. We don't want to have to wait for the clock to tick. Data, all the time-values we didn't read, gets lost but that is OK as we do not want the clock to stop just because we do not look at it. Another purpose of the timer is to facilitate the timeout function or in other words allow a process to sleep for a user specified amount of time.

### Syntax

The grammar for the timer can be seen in Listing 4.5.

*timer* → **timer**

Figure 4.5: The ProcessJ grammar for timer.

### Semantic Description

Listing 4.57 shows the timer declaration in ProcessJ and how current time value can be read from it.

```
timer t;
long time = t.read();
```

Listing 4.57: ProcessJ timer declaration and read.

A read of a timer read returns a long value. A timer can also be used as an ‘alarm clock’, that is, it can be set to block until it times out after a preset time. For example, if we want a timer to stop execution for 1 second we can write code like in Listing 4.58.

```
t.timeout(1000000);//time for timeout is measured in milliseconds
```

Listing 4.58: ProcessJ timer timeout statement.

Code like Listing 4.58 is typically not how a timeout is used. It is often used as a guard in an alt statement such that, if no other guard is ready, the alt statement can time out.

## Code Layout

Listing 4.59 shows how timer declaration and timer read looks like in Java code. Notice that for the read operation, we do not need any timer instance as the *read()* is a static method in the `PJTimer` class. The reason for making is unrelated to an instance or object is that reading the current value does not require any individual state of the timer class.

```
PJTimer t;
long time = PJTimer.read();
```

Listing 4.59: Generated Java code for timer read in Listing 4.57.

Listing 4.60 shows how timeout in ProcessJ is translated to Java code. Remember that though the read operation is not a synchronizing event, timer timeout is and the process needs to go to sleep which means it needs to yield. A timer can also be interrupted by the user in which case, it stops counting down and moves on.

Timers work with the timer queue in the runtime that essentially takes care of maintaining and counting down the time in the timer objects (Listing 4.1.12.2).

```
t = new PJTimer(this, 1000000);
try {
  t.start();
  setNotReady();
  this.runLabel = 1;
  yield();
} catch (InterruptedException e) {
  System.out.println("PJTimer Interrupted Exception!");
}
label(1);
```

Listing 4.60: Generated Java code for timeout in Listing 4.58.

### 4.2.7 Alternative (ALT)

Another new construct that C or Java does not have is *alternation* (or *alt* for short). An alt statement consists of a number of *guarded* statements. To execute an alt, each guard is evaluated, and of the guards that are ready, one is chosen at random and its corresponding statement is executed. A `pri alt` (prioritized alt) does not choose at random, but chooses the top most ready guard.

#### Syntax

The grammar for an alt statement can be found in Figure 4.6.

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

Figure 4.6: The ProcessJ grammar for alt statement.

#### Semantic Description

Each guard can be preceded by an optional Boolean expression – if the Boolean expression is false, the guard is not considered. The selection process for which statement of an alt to execute follows the following algorithm:

1. For each case of an alt statement do the following: If the there is a Boolean expression and it evaluates to *true*, then check if the guard is *ready*:

    - A skip guard is *always* ready.
    - A timeout is ready if the amount of time given in the timeout has elapsed since the alt was evaluated the first time[1].
    - A channel-read guard is ready if there is a committed sender at the other end of the channel, that is, if a communication of a piece of data on the channel is ready to proceed once a read operation is started.

2. From the set of guarded statements for which the guards are ready, if the alt is a prioritized alt, pick the fist one, and if not, pick one at random.

3. If no guards are ready, the alt block yields; however, it remains ready to run and will be re-evaluated next time it is run.

Since a skip guard is always ready, it can serve as default option in an alt statement. If the optional *pri* key word is used, the alt then becomes a prioritized alt in which guarded statement that appear lexicographically earlier than others will be chosen if ready over other ready statements appearing later.

Listing 4.61 shows a generic example of an alt statement in ProcessJ. It has a channel-read guard with a boolean pre-guard, a timer timeout guard, and a skip guard.

### Code Layout

The generated Java code for Listing 4.61 can be seen in Listing 4.62. It can be seen in the Java code that ProcessJ alt statement is converted to an instance of *PJAlt* class with the number of cases, 3, set as parameter. The pre-guards and the guards are also set in alt object. Empty pre-guard is treated as a true value. If all of the guards are false, a runtime exception is thrown. Rest of the code follows the algorithm described in Section 4.2.7.

### Static Semantics

Barriers are not allowed as alt guards for now as it gets very complicated to implement.

[1]This is highly unlikely to be the case the first time around unless the timeout period is 0L or a negative value, but if no other guards are ready, the process is descheduled, and when it gets rescheduled, enough time might have elapsed for the timeout to have happened.

```
timer t;
int v;
alt {
  (a >0) & v = in.read() : {
    ...Statements1...
  }
  t.timeout(100) : {
    ...Statements2...
  }
  skip : {
    ...Statements3...
  }
}
```

Listing 4.61: A generic example of an alt statement in ProcessJ.

```java
PJAlt alt = new PJAlt(3, this);            // create alt object
                              // initialize timers for timeouts
t = new PJTimer(this, 100);
tmp1 = in;
                                                 // guard array
Object[] guards = {tmp1, t, PJAlt.SKIP_GUARD};
tmp0 = (a > 0);                                    // pre-guard
boolean[] boolGuards = {tmp0, true, true};   // pre-guard array
                                   // set guards and pre-guards
boolean bRet = alt.setGuards(boolGuards, guards);
if (!bRet) {              // check if all pre-guards are false
  System.out.println("RuntimeException: One of the boolean
                     pre-guards needs to be  true!!");
  System.exit(1);
}
label(2);                // return here if no guards are ready
chosen = alt.getReadyGuardIndex();         // pick ready guard
switch(chosen) {
  case 0:                                      // ready guard
    v = tmp1.read(this);                       // do the read
    ...Statements1...
    break;
  case 1:                                    // timeout guard
    ...Statements2...
    break;
  case 2:                                       // skip guard
      ...Statements3...
    break;
  case -1:                                  // no ready guards
                     // start timers if this is the first time
    if (!t.started) {
      t.start();
    }
    break;
  }

  if (chosen == -1) {
    yield(2);                   // yield if no guard was ready
  } else {
    if (t.started && !t.expired) {             // kill timers
      t.kill();
    }
    yield(3);                           // yield for fairness
  }
  label(3);                 // return here if the alt succeeds
}
```

Listing 4.62: Generated Java code for Listing 4.61.

### 4.2.8 Barrier

A barrier is a multi-way *synchronization point* in a ProcessJ program. A process can `enroll` on a barrier, which means that it may synchronize on it (using the `sync` keyword). When a process synchronizes on a barrier, it will be held at that barrier until every other process enrolled on that barrier also synchronizes on it. Only when all enrolled processes have called `sync` on the barrier in question will every process be allowed to proceed past the `sync` call. If just one of the enrolled processes does not call `sync`, all other processes will be prevented from progressing in their execution. Barriers can be passed like any other primitive value to procedures, but cannot be sent over channels; that is, barriers cannot be declared `mobile`. A process enrolled on a barrier automatically resigns the barrier enrollment upon termination [BWS05b].

#### Syntax

The grammar for a barrier can be seen in Listing 4.7.

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

Figure 4.7: The ProcessJ grammar for a barrier declaration.

#### Semantic Description

A barrier is declared simply by using the `barrier` keyword (Listing 4.63).

```
barrier b;
```

Listing 4.63: The PJBarrier declaration.

Listing 4.64 shows an example of the usage of barriers. A par block enrolls on a barrier so that, in turn, the procedure invocations in it can be enrolled on the same barrier.

```
par enroll (b) {
  foo(b);
  bar(b);
}
```

Listing 4.64: Par block enrolled on a barrier.

A procedure that is enrolled on a barrier can call the *sync()* method at any point in its execution to yield and wait for other processes enrolled on that barrier before moving on.

```
proc void foo(barrier b) [yield=true] {
  ...Statements1...
  sync(b);
  ...Statements2...
}
```

Listing 4.65: A barrier-enrolled procedure.

## Code Layout

Barrier declaration in the generated Java code are initialized with an instance of the `PJBarrier` class (Listing 4.66).

```
PJBarrier b = new PJBarrier();
```

Listing 4.66: Generated Java code for Listing 4.63.

Listing 4.67 shows the generated code for Listing 4.64 where a par block with two procedures is enrolled on the barrier ‘b’ by setting the counter in the `PJBarrier` class to 2. The two procedures in the par block are changed to processes and in their *finalize()* method, they resign from the barrier after the execution of their *run()* method is complete.

Listing 4.68 shows the generate code for one of the procedures, namely *foo*, so as to show an example of the *sync()* call. When the sync method is invoked, the counter kept in the `PJBarrier` object is decremented and the process is added to a queue such that it can be set ready to run when every process enrolled on the barrier has called *sync()*. The *sync()* method also sets the process not ready to run.

```
final PJPar par2 = new PJPar(2, this);
b.enroll(2);
(new simpleBarrier.foo( b ){
  public void finalize() {
    par2.decrement();
    b.resign();
  }
}).schedule();
(new simpleBarrier.bar( b ){
  public void finalize() {
    par2.decrement();
    b.resign();
  }
}).schedule();
setNotReady();
this.runLabel = 2;
yield();
label(2);
```

Listing 4.67: Generated Java code for Listing 4.64.

```
public static class foo extends PJProcess {
  PJBarrier b;
  public foo(PJBarrier b) {
    this.b = b;
  }
  @Override
  public synchronized void run() {
    switch(this.runLabel) {
      case 0 break;
      case 1 resume(1); break;
    }
    ...Statements1...
    b.sync(this);
    this.runLabel = 1;
    yield();
    label(1);
    ...Statements2...
    terminate();
  }
}
```

Listing 4.68: Generated Java code for Listing 4.65

### 4.2.9 Records

A record in ProcessJ is much like a struct in C, except the extends part, which lets a new record inherit an existing records fields.

#### Syntax

The grammar for a record declaration is shown in Figure 4.8.

```
record_type_declaration  →  modifier* record ID [extends ID (, ID)*] {
                            (type variable_id (, type variable_id )* ;)+
                            }
variable_id              →  ID
                            variable_id [ ]
```

Figure 4.8: The ProcessJ grammar for records.

Records are allocated dynamically using the keyword new followed by a record literal and the null value represents a non-allocated value.

#### Semantic Description

Listing 4.69 shows declaration of two records, namely ‘K’ and ‘P’ that extends ‘K’. That means, record ‘P’ also includes the field ‘z’ by extension.

```
record K {
  int z;
}
record P extends K{
  int x;
  int y;
}
```

Listing 4.69: Record declaration and extension.

Listing 4.70 shows an example of a record literal or how a record is instantiated with values. Though record ‘P’ only has two integer values, instantiating it with three values is needed as it extends record ‘K’.

A more complex example of a record with different data types can be seen in Listing 4.71.

```
proc void foo() {
  P myRecord = new P{1, 2, 3};
}
```

Listing 4.70: Record literal.

```
public record Client {
  string full_name;
  string address1, city;
  int zip;
  Transaction transactions[];
}
```

Listing 4.71: A more complex record example.

### Code Layout

Listing 4.72 shows the generated Java code for Listing 4.69. The interesting code here is for *Record_P* which includes the field ‘z’ as well since it extends *Record_K*.

```
public static class Record_K {
  public int z;
  public Record_K(int z) {
    this.z = z;
  }
}
public static class Record_P {
  public int x;
  public int y;
  public int z;
  public Record_P(int x, int y, int z) {
    this.x = x;
    this.y = y;
    this.z = z;
  }
}
```

Listing 4.72: Generated Java code for Listing 4.69.

The Java code for the record literal is very similar to ProcessJ code with the only difference being the generated name for the records (Listing 4.73).

Listing 4.74 shows the generated Java code for the more complex record example in Listing 4.71. Here

```java
public static void foo() {
  Record_P myRecord=new Record_P(1, 2, 3);
}
```

Listing 4.73: Generated Java code for Listing 4.70.

the *Record_Transaction* is another record type declared in the ProcessJ code which means that records can have other records, or arrays of them, as its field.

```java
 public static class Record_Client {
   public String full_name;
   public String address1;
   public String city;
   public int zip;
   public Record_Transaction[] transactions;
   public Record_Client(String full_name, String address1,
                String city, int zip,
                Record_Transaction[] transactions) {
     this.full_name = full_name;
     this.address1 = address1;
     this.city = city;
     this.zip = zip;
     this.transactions = transactions;
   }
 }
```

Listing 4.74: Generated Java code for Listing 4.71.

### 4.2.10 Protocol

The protocol type constructor in ProcessJ has many similarities to a union data type in C. It consists of a number of tag-named variable lists, but like records, it can inherit from other protocol types. The idea of a protocol type is typically to serve as a datatype, a protocol, for channel communication. Using a protocol type for communicating allows for different types of value to be communication (see Appendix E for a realistic use of protocols).

#### Syntax

The grammar for declaring protocol types can be seen in Figure 4.9.

$$\textit{protocol\_type\_declaration} \rightarrow \textit{modifier}^* \ \textbf{protocol} \text{ ID } [\textbf{extends} \text{ ID } (\textbf{,} \text{ ID})^*]$$
$$( \ \{ \ (\text{ID} \textbf{ : } \{ \ (\textit{type} \text{ ID } \textbf{;})^* \ \})^* \ \} \ | \ \textbf{;} \ )$$

Figure 4.9: The ProcessJ grammar protocol.

## Semantic Description

An example of protocol declaration can be seen in Listing 4.75.

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

Listing 4.75: An example of protocol declaration.

A variable of type `P` contains **either** an integer `number` and a double `amount` while being tagged as `request` **or** a boolean `status` while being tagged `reply`.

One major difference between unions in C and protocols in ProcessJ is that protocols can *extend* existing protocols. For example, consider the example in Listing 4.76.

```
public protocol P1 extends P {
  deny: { int code;}
}
```

Listing 4.76: ProcessJ code for a protocol with an extends.

It declares a protocol type `P1` which *inherits* the `request` and `reply` cases from `P` and further extends the type by adding a new case called `deny`. However, the inheritance of a protocol is reversed compared to Java. The protocol that extends another protocol behaves like the parent of the extended protocol.

A protocol can extend as many other protocols as needed. Listing 4.77 illustrates this.

Here, `P7` will contain all the cases from `P6`, `M2`, and `Y6`; if any of those three protocol types have similar tags an error will be produced by the compiler.

A protocol body can be empty but only if it extends at least one other protocol. However, it is perfectly OK for a protocol case not to have any declarations in its list. That is often useful if a case simply serves

```
public protocol P7 extends P6, M2, Y6 ;
```

Listing 4.77: ProcessJ code for a protocol with multiple extends.

to mark a certain choice that does not need to carry any data, like for example an acknowledgment in Listing 4.78.

```
public protocol P2 {
  request: { int package_no; }
  ack: {}
}
```

Listing 4.78: ProcessJ code for protocol with empty case tag.

### Code Layout

```
public static class Protocol_P{
  public static class Protocol_P_request extends PJProtocolCase {
    public int number;
    public double amount;
    public Protocol_P_request(int number, double amount) {
      this.number = number;
      this.amount = amount;
      this.tag = "request";
    }
  }
  public static class Protocol_P_reply extends PJProtocolCase {
    public boolean status;
    public Protocol_P_reply(boolean status) {
      this.status = status;
      this.tag = "reply";
    }
  }
}
```

Listing 4.79: Generated Java code for Listing 4.75.

Listing 4.79 shows the generated Java code for a protocol declaration. A ProcessJ protocol gets translated to a Java class and each case is also translated to an inner class in the protocol's class. The inner classes extend the runtime element `PJProtocolCase` (Section 4.1.11) which has *tag* field. The constructor of the case classes take in the field values of the protocol tag during instantiation.

Listing 4.80 shows another Java generated code example with protocol extension for Listing 4.76. As seen in the example, it does not explicitly mention the extended class in the Java code unlike ProcessJ code. The reason for that is, the proper use of extended protocols is checked in the ProcessJ code by the compiler before the code generation phase. As far as Java code is concerned, as long as the inner classes of the protocol case extends the `PJProtocolCase` class and the channels passing protocols using the same class as the data type, the proper usage of the extended protocols can be replicated.

```java
public static class Protocol_P1{
  public static class Protocol_P1_deny extends PJProtocolCase {
    public int code;
    public Protocol_P1_deny(int code) {
      this.code = code;
      this.tag = "deny";
    }
  }
}
```

Listing 4.80: Generated Java code for Listing 4.76.

Listing 4.81 shows another example of a protocol that is extending multiple protocols. Though multiple inheritance is not possible with Java, since we do not need to explicitly extend the protocols as mentioned in the paragraph above, it is not an issue in the generated code.

```java
public static class Protocol_P7{ }
```

Listing 4.81: Generated Java code for Listing 4.77.

Listing 4.82 shows an example with an empty protocol case called *ack* from Listing 4.78.

```
public static class Protocol_P2{
  public static class Protocol_P2_request extends PJProtocolCase{
    public int package_no;
    public Protocol_P2_request(int package_no) {
      this.package_no = package_no;
      this.tag = "request";
    }
  }
  public static class Protocol_P2_ack extends PJProtocolCase {
    public Protocol_P2_ack() {
      this.tag = "ack";
    }
  }
}
```

Listing 4.82: Generated Java code for Listing 4.78.

## 4.3 State Management and Rewriting for Resumption

In this section, we describe why state management or retention is necessary in ProcessJ, the technique implemented for doing so, and how the ASM tool was used to instrument the process class objects for the yield point resumptions.

### 4.3.1 State Management

As ProcessJ procedures can yield and be rescheduled at a later time, its state must be preserved between invocations. An approach to store all locals in activation record-like structures on a stack (essentially an array of Objects) was described in [PS14]. The locals were stored before the procedure yielded and restored upon rescheduling. However, we have taken a different approach here. Instead of having an activation stack, we convert all the locals and formals to fields. As the object of the process is what goes into and out of the run queue, a list of processes scheduled to run (Section 4.1.12.1), it carries with it its state. The *run()* method of a process does not take any parameters, so, we pass them to the constructor. Since lifting the locals and parameters from their scope to become fields of the class can potentially cause name conflicts, the original names are prefixed with auto-generated strings. The locals are prefixed with *_ldX$* where 'X' is an auto incremented integer value and the parameters are prefixed with *_pd$*. We were initially concerned that accessing fields over locals might take longer. But our tests showed the difference is an overhead of around 1% to 2.5%. It is not much, specially, considering that there is also an overhead in the activation stack approach in accessing it as a field and putting data into and taking data out of them multiple times during the lifetime of the process.

### 4.3.2 Bytecode Instrumentation with ASM for Resumption

The ProcessJ code yields at synchronizing points such as a par block, a barrier or a channel read/write. As we are targeting the JVM platform in this thesis, we generate Java source for a ProcessJ source. Now, the difficulty with doing a yield in the middle of a Java method, which is equivalent to returning from it, is that we can not have dead code by inserting explicit return statements in the middle of a method body. Though there is a *goto* reserved keyword in Java, it is only used by the Java compiler and not available in the source language.

To accomplish process yielding and resumption, we have implemented the approach of bytecode rewriting proposed in [PK09]. We used the ASM tool for this rewriting, which we term *instrumentation*.

Let us look at a piece of ProcessJ pseudo-code for a par block with a channel read and a channel write expressions (Listing 4.83). For this example, we will focus only on the code generated for the channel read operation in line 2.

```
1: par {
2:   x = c1.read();
3:   c2.write(10);
4: }
```

Listing 4.83: ProcessJ code for a par block.

For the yielding purpose, the *PJProcess* class has 3 placeholder methods, namely *resume()*, *label()*, and *yield()*. Listing 4.84, shows the generated Java code for the channel read in the par block in line 2. As all statements in a par block are treated as a process, we wrap the reading operation with a new `PJProcess` instance and generate the reading code in the run method of it.

```
new Process() {
  public synchronized void run() {
    switch(this.runLabel) {
      case 0: break;
      case 1: resume(1); break;
      case 2: resume(2); break;
    }
    label(1)};
    if(c1.isReadyToRead(this)) {
      // channel code...
      this.runLabel = 2;
      yield()};
    else {
      // channel code...
      this.runLabel = 1;
      yield()};
    }
    label(2);
    terminate();
  }
  public void finalize() {
    par.decrement();
  }
}
```

Listing 4.84: Java code for the channel read in par block.

Figure 4.10 shows the starting part of the Java bytecode generated in the class file after compiling the channel read Java code in Listing 4.84. The switch statement has been translated into a `tableswitch` instruction and the calls to `resume` follow at address 35 and 43. At address 48, we see the `label(1)` invocation from the generated code and `label(2)` follows it at some later address.

```
public synchronized void run();
  Code:
    0: aload_0
    1: getfield runLabel I
    4: tableswitch  // 0 to 2
            0: 32
            1: 35
            2: 43
        default: 48

    32: goto 48
    35: aload_0
    36: iconst_1
    37: invokevirtual resume/(I)V
    40: goto 48
    43: aload_0
    44: iconst_2
    45: invokevirtual resume/(I)V
    48: aload_0
    49: iconst_1
    50: invokevirtual label/(I)V
    xx: ...
    xx: ...
```

Figure 4.10: Bytecode for channel-read in ProcessJ.

We then use the ASM bytecode manipulation tool to find all the addresses of all the `label()` invocation and insert a goto instruction each that jumps to correct labels after the corresponding resumes. Figure 4.10 shows the instrumented Java bytecode. Address 37 shows the first resume call, `resume(1)`, and at address 40, we have inserted a goto instruction that jumps to address 54 which is the address of `label(1)`. Similarly, for `resume(2)` at address 48, a goto instruction that jumps to address 139 is inserted after it which is the address for `label(2)`. Since the `yield` points need to behave like a return statement, after each yield instruction in addresses 103 and 133, a goto instruction that jumps to the address of the return statement, 148, is inserted.

```
public synchronized void run();
    Code:
      0: aload_0
      1: getfield runLabel I
      4: tableswitch  // 0 to 2
             0: 32
             1: 35
             2: 46
           default: 54

      32: goto 54
      35: aload_0
      36: iconst_1
      37: invokevirtual resume/(I)V
      40: goto 54
      43: nop
      44: nop
      45: athrow
      46: aload_0
      47: iconst_2
      48: invokevirtual resume/(I)V
      51: goto 139
      54: aload_0
      55: iconst_1
      56: invokevirtual label/(I)V
      xx: ...
      xx: ...
      103: invokevirtual yield/()V
      106: goto 148
      xx: ...
      xx: ...
      133: invokevirtual yield/()V
      136: goto 148
      139: aload_0
      140: iconst_2
      141: invokevirtual label/(I)V
      144: aload_0
      145: invokevirtual terminate/()V
      148: return
```

Figure 4.11: Bytecode after instrumentation.

# Chapter 5

# Results

## 5.1 The Runtime Object Sizes

In this section we consider the size (in bytes) of each runtime class which were found by serializing their instances and calculating the sizes of the corresponding byte arrays. This information can be helpful in determining runtime space requirements for ProcessJ programs.

| Runtime Class | Size |
|---|---|
| PJProcess* | 68 bytes |
| PJAlt* | 223 bytes |
| PJBarrier* | 127 bytes |
| PJChannel | 143 bytes |
| PJOne2OneChannel* | 204 bytes |
| PJOne2ManyChannel* | 269 bytes |
| PJMany2OneChannel* | 269 bytes |
| PJMany2ManyChannel* | 283 bytes |
| PJPar | 133 bytes |
| PJProtocolCase | 63 bytes |
| PJTimer (alone) | 119 bytes |
| PJTimer (w/ Process) | 182 bytes |

Table 5.1: Runtime class-size in bytes.

The runtime classes marked with a * in Table 5.1 are given in *base sizes*, that is, they grow in size depending on the number of other runtime instances associated with them. For example, the *PJProcess* class size of 68 bytes is for a process alone. Its size will increase with respect to the size of the other runtime objects it contains. The unmarked classes do not increase in size; for example, *PJPar*'s size remains the same as shown in Table 5.1, as it only takes in an integer value and an instance of a process as parameters during its instantiation and no other objects are associated with it after that.

## 5.2 ProcessJ Benchmarking for Max Number of Processes

As with any concurrent language, it is an interesting exercise to determine how many processes the runtime supports. Here, we show the results of our benchmarking test for ProcessJ.

We used a simple ping-pong type program with two processes communicating one piece of data back and forth once across two different channels. Another process, the main, runs a par block with the two processes and wraps the par block with a par for loop that iterates a number of times based on a user provided count. The actual code is shown in Appendix A. The execution architecture we used is:

- Intel(R) Xeon(R) CPU (32-core) E5-2630 v3 @ 2.40GHz with 128GB RAM running GNU/Linux (3.10.0-327.4.5.el7.x86_64). On this machine, we managed to run 480,900,001 processes.

Table 5.2 shows the results of this test. We can see that, in each execution, the context switch is at least twice the number of processes run. This is due to the nature of the ping-pong program. As mentioned earlier, we were able to run over 480 million processes on a single-CPU JVM, though at an expense of 126 GB of RAM. But we think that it is a fair trade in being able to run that many processes which can be used for very interesting modeling problems such as boids simulation and blood clotting models. However, we can see that even in 1.79 GB of memory, we were able to run 7 million processes something **no** thread based system can do; not even close.

| # of Processes | # of Context Switches | Execution Time | Memory Used |
|---:|---:|---:|---:|
| 7,000,001 | 15,000,002 | 7.53 secs | 1.79 GB |
| 10,500,001 | 22,500,002 | 16.03 secs | 3.02 GB |
| 14,000,001 | 30,000,002 | 25.86 secs | 4.10 GB |
| 210,000,001 | 450,000,002 | 642.8 secs | 63.91 GB |
| 350,000,001 | 750,000,002 | 1235.12 secs | 94.50 GB |
| 420,000,001 | 900,000,002 | 1443.40 secs | 125.82 GB |
| 476,000,001 | 1,020,000,002 | 1800.79 secs | 126.11 GB |
| 480,900,001 | 1,030,500,002 | 1801.40 secs | 126.20 GB |

Table 5.2: Benchmark test.

## 5.3 Timing (CommsTime)

Here we estimate the time it takes to perform channel communication using the CommsTime benchmark (Figure 5.1) from [PS14]. The CommsTime benchmark is a process network with 4 sub-processes (*prefix*, *consumer*, *delta*, and *succ*). As described in [PS14], for each number consumed by the consumer process, 4 channel communications must happen. The benchmark was performed on the same two execution architectures and in the same way as done in [PS14]. That paper used a ProcessJ prototype runtime system for this test and the process prototype was called `LiteProc`. Comparisons were done against JCSP channel communication and here, we will be vetting our ProcessJ process against the both of them. The two execution architectures are:

- Mac Pro 4.1, OS X Snow Leopard, Intel i7 Quad-core Xenon 2.93 MHz with 8GB RAM.
- AMD dual 16 core Opteron 6274 (2.2 GHz) with 64GB 1,333 MHz DDR3 ECC Registered RAM running CentOS 6.3 (Linux 2.6.32).

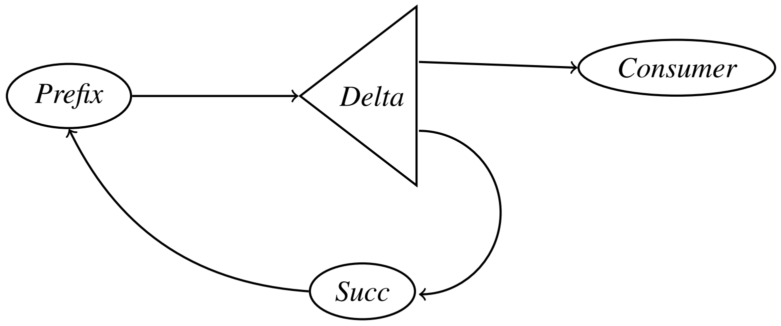

Figure 5.1: The CommsTime network.

Table 5.3 reports both the results from [PS14] and the result from this thesis (found in the *PJProcess* column) so that comparisons can be made. It can be seen that *PJProcess* out-performs both the *LiteProc* and the *JCSP* in every aspect. The timings obtained on the AMD architecture in [PS14] may have been slightly off (slower than expected), but the timings of the OS/X machine seem more reasonable, so for a realistic comparison it may make more sense to compare the LiteProc and JVMCSP (the system developed in this thesis) to get a idea of the size of the improvement.

For the OS/X machine we saw an improvement in iteration and communication time of around 10% and an improvement of context switching of almost 50%. The improvement in context switching time is most likely because of the removal of the code that associated with creating activation record objects and storing into and retrieving from these objects the values of the parameters and local variables.

## 5.4 Conformity Tests

To confirm that each language constructs we designed and implemented worked as expected, we performed small individual tests. However, the only way to ensure that that they worked correctly with each other was

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

Table 5.3: CommsTime results.

to implement real-life concurrency problems. To that avail, we have used three programs written in ProcessJ. These programs and the result of compiling and running them are presented in the following subsections.

### 5.4.1 Mandelbrot Generation

The Mandelbrot picture generation is a problem that requires a lot of computation. It is usually an easy program to parallelize as it is embarrassingly parallel.

We implemented four versions of the Mandelbrot program to perform some time computations: a sequential version in Java (Appendix B), a sequential version in ProcessJ which looks similar to the sequential version in Java except instead of methods it has procedures, a parallel version in ProcessJ parallelizing the row computations (Appendix C), and a parallel version in ProcessJ parallelizing every pixel computation. To parallelize the sequential version of the code, we need to add the *par* keyword to the outer loop (the display height loop) of the sequential version, thus parallelizing computation of each row and for the pixel parallization, *par* keyword is required on both the outer and the inner loop.

We generated a 4,000x3,000 Mandelbrot picture using all four programs (Figure 5.2). We also timed their executions (Table 5.4) and found the sequential Java version to take around 6.24 seconds and the row-parallel ProcessJ version took around 6.05 seconds. This shows that there is no benefit in parallelizing this problem on a single-core, which is not surprising, but we believe there will be significant speed-up with a multi-core scheduler. However, it does illustrate that everything ran correctly and also that the overhead of creating and scheduling the JVMCSP processes is negligible.

Table 5.4: Mandelbrot picture results.

| Version | Time (Sec.) | #Processes | Context Switches |
|---|---|---|---|
| Java sequential | 6.24 | 1 | 0 |
| ProcessJ sequential | 6.21 | 1 | 0 |
| ProcessJ row parallelized | 6.05 | 3,001 | 3001 |
| ProcessJ pixel parallelized | 31.98 | 12,000,001 | 12,003,001 |

Figure 5.2: The mandelbrot picture generated by the parallel ProcessJ program.

The execution time for the pixel-parallel ProcessJ version took 31.98 seconds which is very high. Parallelizing each pixel needed 12 million processes. The object creation time in Java for them was found to be 25.57 seconds, thus the computation after that only took 6.41 seconds. The actual execution time is comparable to other versions of the program. Though, it should be noted that even though it seems that the object creating time is significant, in reality it takes the JVM only around 2.15 -seconds to create an object.

### 5.4.2 The Santa Claus Problem

The Santa Claus Problem is an excellent concurrency exercise as defined in []. We implemented a solution in ProcessJ (Appendix E), generated code for it and executed it successfully. As the solution for the Santa Claus problem use various constructs of the language such as *par blocks*, *parfor blocks*, *channel communication*, *alt blocks*, *Protocol types* and *barriers*, this is a good test to verify the correct operation of almost all the CSP primitives of ProcessJ, including protocols.

### 5.4.3 Full Adder

We also implemented an 8-bit full adder in ProcessJ. As its components, it has a multiplexer, several gates such as AND, OR, NOT, NAND and XOR, a single-bit adder, a four-bit adder and a full eight-bit adder. There are a total of 632 processes that compose to generate the result of the full adder. This is a good exercise for a thorough channel communication test.

Table 5.5 shows the Line of Code (LOC) count for each component in ProcessJ versus the Java generated code.

| Component | ProcessJ LOC | Generated Java LOC |
|---|---|---|
| MUX | 8 | 91 |
| AND | 8 | 94 |
| OR | 8 | 94 |
| NOT | 5 | 44 |
| NAND | 7 | 35 |
| XOR | 12 | 76 |
| ADDER.ONE | 14 | 94 |
| ADDER.FOUR | 9 | 71 |
| ADDER.FULL | 7 | 81 |

Table 5.5: *The Full Adder LOC* - ProcessJ vs Java generated code.

Listing 5.1 shows an example of one of the components, the AND gate, in ProcessJ and in Listings 5.2-5.5 we illustrate the equivalent generated code in Java. This side-by-side comparison, so to speak, is interesting for as it gives a good picture of what the generated Java source code looks like for a piece of ProcessJ code and how the CPS primitives in ProcessJ are translated to the JVMCSP runtime classes.

```
proc void andGate(chan<boolean>.read in1, chan<boolean>.read in2,
 ↪  chan<boolean>.write out)[yield=true]{
  boolean x=false, y=false;
  par{
    x = in1.read();
    y = in2.read();
  }
  out.write(x && y);
}
```

Listing 5.1: Example code of *Full Adder* AND gate in ProcessJ.

We can see in Listing 5.2 that the procedure `andGate` in Listing 5.1 gets translated to a Java class extending `PJProcess` runtime class. The content of the procedure is written in to the *run()* method of the class. All yield points are placed in a switch block that switches on the `runLabel` at the beginning of the *run()* method for resumption purpose. The `par` block is translated to `PJPar` class instance with the two channel read operations associated to it. Each channel read operation is wrapped in an anonymous process as they are yielding events that need to be scheduled and blocks the `andGate` process until complete. The channel write operation is translated with Java channel write code template (Listing 5.5) and placed at the end of the *run()* method of the `andGate` process. All other components of the `fullAdder` are translated in a similar manner depending on the operations they contain.

```
public static class andGate extends PJProcess {
  PJChannel<Boolean> in1, in2, out;
  boolean x, y;
  public andGate(PJChannel<Boolean> in1,
          PJChannel<Boolean> in2, PJChannel<Boolean> out) {
    this.in1 = in1;
    this.in2 = in2;
    this.out = out;
  }
  @Override
  public synchronized void run() {
    switch(this.runLabel) {
      case 0: break;
      case 1: resume(1); break;
      case 6: resume(6); break;
      case 7: resume(7); break;
    }
    x = false;
    y = false;
```

Listing 5.2: Generated Java code for *Full Adder* AND gate - part1.

```
    final PJPar par1 = new PJPar(2, this);
    new PJProcess(){
      @Override
      public synchronized void run() {
        switch(this.runLabel) {
          case 0: break;
          case 2: resume(2); break;
          case 3: resume(3); break;
        }
        label(2);
        if(in1.isReadyToRead(this)) {
          x = in1.read(this);
          this.runLabel = 3;
          yield();
        } else {
          setNotReady();
          in1.addReader(this);
          this.runLabel = 2;
          yield();
        }
        label(3);
        terminate();
      }
      @Override
      public void finalize() {
        par1.decrement();
      }
    }.schedule();
```

Listing 5.3: Generated Java code for *Full Adder* AND gate - part1.

```java
    new PJProcess(){
      @Override
      public synchronized void run() {
        switch(this.runLabel) {
          case 0: break;
          case 4: resume(4); break;
          case 5: resume(5); break;
        }
        label(4);
        if(in2.isReadyToRead(this)) {
          y = in2.read(this);
          this.runLabel = 5;
          yield();
        } else {
          setNotReady();
          in2.addReader(this);
          this.runLabel = 4;
          yield();
        }
        label(5);
        terminate();
      }

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

Listing 5.4: Generated Java code for *Full Adder* AND gate - part2.

```
    setNotReady();
    this.runLabel = 1;
    yield();
    label(1);

    label(6);
    if (_pd$out.isReadyToWrite()) {
      out.write(this, (x && y));
      this.runLabel = 7;
      yield();
    } else {
      setNotReady();
      this.runLabel = 6;
      yield();
    }
    label(7);
    terminate();
  }
}
```

Listing 5.5: Generated Java code for *Full Adder* AND gate - part3.

# Chapter 6

# Conclusion

In this thesis, we have implemented a code generator for translating ProcessJ source code into Java code as well as a robust runtime system that includes representations of the CSP primitives (par, alt, barrier, etc.), and a simple non-preemptive cooperative scheduler. To verify the design-correctness of the runtime classes for the CSP primitives, we wrote and executed several complex tests such as the Santa Claus problem, a full-adder, and the Mandelbrot set computation.

In order to test the capabilities of the scheduler with respect to the maximum number of processes that the system can handle, we used a simple ping-pong type program, and the size of the run queue maxed out at almost 481,000,000 processes. This was done on a 128GB machine and used a total of 126GB. We have no doubt that had we had 256GB or more we could have achieved an almost linear growth in terms of the number of processes we could handle.

Another important measure is context switching time. We measured context switching time using the CommsTime benchmark and ran it on the same two machines used in [PS14] in order to obtain comparable timings. We saw an improvement from 1.322 to 0.69 s per context switch on the Mac/OS-X architecture, which is a significant improvement. This improvement, we hypothesize, is due to the changes made to how locals and parameters are handled; in [PS14] activation records were kept, and the locals needed to be saved into an activation record at every yield-point and restored from an activation at every resumption point. The overhead of this storing and restoring is a significant amount of the time it takes to re-invoke a process by the scheduler.

We also demonstrated, with the help of execution-time comparisons between the sequential and the parallel version of the Mandelbrot program, that even on a single-core, the incurred overhead is not prohibitive. The difference in runtime between the sequential Java program, the sequential ProcessJ program and the concurrent ProcessJ program was negligible. However, the parallel version that parallelized every pixel computation (with a nested `par for` loop) instead of just the rows, performed very poorly; 31.98 secs versus 6.05 seconds for the parallel-row version. To generate a 4,000x3,000 pixel Mandelbrot, the parallel-pixel version created 12,000,000 processes (1 for each pixel). Further inspection showed that there was an overhead of 25.57 seconds to create the process objects on the JVM; thus the execution only took 6.41 seconds which is comparable to 6.05 seconds of the parallel-row version. As for the overhead, it comes from our choice of targeting the JVM. Object creation cannot be avoided  not even if we switch back to using activation records. Ultimately, it is partially the job of the programmer to chose the correct granularity of concurrency when writing parallel programs.

In conclusion, we have developed a robust runtime capable of handling a large number of processes without a significant context-switching overhead and a code generator for the ProcessJ compiler that utilizes this runtime to executing concurrent processes on the JVM. All the tests we have run have outperformed the prototype and the JCSP libraries, so the goal that we set out to achieve has been met.

# Chapter 7

# Future Work

## 7.1 Multi-Core Scheduler

A multi-core scheduler for the ProcessJ runtime is in the works; we have already designed the runtime elements with proper synchronized accesses. Even though it this was not required for the single-core scheduler, not too many modifications will be required for it to work with the multi-core scheduler. Having the multi-core scheduler will not increase the number of processes that can be run on a single JVM as this is memory dependent [PS14]. However, in theory, it will speed up the execution time. Writing a multi-core scheduler is not a simple task, but we believe that the ground work established in this thesis by the fully working and tested single-core scheduler and some techniques such as described in [RSB12] will be helpful in accomplishing this feat.

## 7.2 Libraries

Libraries are an important part of any language. We already have some basic libraries for standard input and output, random number generation, and a string library. However, many more libraries are needed before the language is ready for a general release.

## 7.3 Blocking I/O Calls

Blocking I/O calls can be a problem for a cooperative scheduler, there is no easy way to handle external calls in the same way as we handle internal synchronization primitives like channel communication; once an external I/O call blocks, the entire scheduler (or at least the thread running the scheduler) will block. We cannot simply de-schedule and wait for the call to finish in order to be woken up again (or set ready to run), but blocking I/O calls can be handled by spawning a new individual Java thread [PS14].

## 7.4 Mobile Procedure and Polymorphic Resumption

Mobile processes were not covered by the scope of this thesis, but the ProcessJ language already supports them. Mobile processes are not part of CSP, but an extension defined by the $\pi$-calculus [Mil99]. The necessary changes to the runtime and the code generator in order to support mobile processes is minimal; since the cooperative scheduling of non-mobile processes was based on work developed in order to support mobile processes on the JVM [PK09], all explicit yield points (called suspend points) are handled similarly to implicit ones. The major change in the compiler is the re-implementation of the name resolution phase as the scoping rules for such processes are slightly different [PK09].

## 7.5 Alternative (alt)

Alts are currently implemented as a busy-wait system where they keep yielding with a ready to run status and keep getting rescheduled until one of the guards is ready. We would like to improve this naive implementation by yielding with a not-ready status and have the guard that gets ready set the alt ready to run.

## 7.6 Claim

Shared channel-ends are claimed before any operation is done on them. The claiming mechanism is also implemented as a busy-wait system. A process continues checking if a channel-end can be claimed by yielding with a ready to run status and getting re-scheduled by the scheduler. As this wastes cycles, it needs to be improved by a process yielding with not-ready status if it is unsuccessful in claiming a channel-end and being woken up by the same channel-end when a claim can be successfully processed.

## 7.7 Run Queue

The current single-core scheduler has a single run queue which holds both ready to run and not ready to run processes. An improvement on this would be to have two different run queues [PS14]; one for the ready processes and the other for not-ready processes. This will decrease the overhead of iterating through non-ready processes that the current scheduler does. To make this work, it again requires some book keeping and process-waking-up mechanisms in many runtime elements. This improvement will be done as a part of the multi-core scheduler.

# Appendix A

# Billions of Processes Benchmark Test For ProcessJ

```
proc void foo(chan<int>.read c1r, chan<int>.write c2w)[yield=true]{
  int x;
  par {
    x = c1r.read();
    c2w.write(10);
  }
}
proc void bar(chan<int>.write c1w, chan<int>.read c2r)[yield=true]{
  int y;
  par {
    y = c2r.read();
    c1w.write(20);
  }
}
proc void main(string[] args)[yield=true]{
  par for (int i=0; i<100000; i++) {
    chan<int> c1, c2;
    par {
      foo(c1.read, c2.write);
      bar(c1.write, c2.read);
    }
  }
}
```

Listing A.1: ProcessJ code for benchmarking # of processes that can be run on a single-core JVM.

# Appendix B

# Parallel MandelBrot in ProcessJ

```
import images.pgm;
public proc int cal_pixel(double real, double imag) {
  int count, max;
  double z_real, z_imag;
  double temp, lengthsq;

  max = 256;
  z_real = 0;
  z_imag = 0;
  count =0;
  do {
    temp = z_real * z_real - z_imag * z_imag + real;
    z_imag = 2 * z_real * z_imag + imag;
    z_real = temp;
    lengthsq = z_real * z_real + z_imag * z_imag;
    count++;
  } while ((lengthsq < 4.0) && (count < max));
  return count;
}
```

Listing B.1: ProcessJ code - parallel *mandelbrot* part 1.

```
proc void main(string[] args)[yield=true] {
  int disp_width  = 1024;
  int disp_height = 768;

  int mandelbrot[][] = new int[disp_height][disp_width];

  double real_min = -0.7801785714285;
  double real_max = -0.7676785714285;
  double imag_min = -0.1279296875000;
  double imag_max = -0.1181640625000;

  double scale_real = (real_max-real_min)/disp_width;
  double scale_imag = (imag_max-imag_min)/disp_height;

  par for (int y=0; y<disp_height; y++) {
    for (int x=0; x<disp_width; x++) {
      double c_real = real_min + ((double) x * scale_real);
      double c_imag = imag_min + ((double) y * scale_imag);
      mandelbrot[y][x] = 256 - cal_pixel(c_real, c_imag);
    }
  }

  write_P2_PGM(mandelbrot, "mm.pgm", 256);
}
```

Listing B.2: ProcessJ code - parallel *mandelbrot* part 2.

# Appendix C

# Sequential MandelBrot in Java

```
import java.io.PrintWriter;
public class mandelbrotseq {
  public static int cal_pixel(double real, double imag) {
    int count, max;
    double z_real, z_imag;
    double temp, lengthsq;

    max = 256;
    z_real = 0;
    z_imag = 0;
    count = 0;
    do {
      temp = z_real * z_real - z_imag * z_imag + real;
      z_imag = 2 * z_real * z_imag + imag;
      z_real = temp;
      lengthsq = z_real * z_real + z_imag * z_imag;
      count++;
    } while ((lengthsq < 4.0) && (count < max));
    return count;
  }
```

Listing C.1: Java code - sequential *mandelbrot* part 1.

```
  public static boolean write_P2_PGM(int pic[][], String filename, int
   ↪  max) {
    // assume that pic is rectangular.
    boolean ok = true;
    try {
      int width = pic[0].length;
      int height = pic.length;
      PrintWriter writer = new PrintWriter(filename, "UTF-8");
      writer.println("P2");
      writer.println(width + " " + height);
      writer.println(max);
      for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++)
          writer.print(pic[i][j] + " ");
        writer.println("");
      }
      writer.close();
    } catch (Exception e) {
      ok = false;
    }
    return ok;
  }

  public static void main(String args[]) {
    int disp_width = 4000;
    int disp_height = 3000;
    int mandelbrot[][] = new int[disp_height][disp_width];
    double real_min = -0.7801785714285;
    double real_max = -0.7676785714285;
    double imag_min = -0.1279296875000;
    double imag_max = -0.1181640625000;
    double scale_real = (real_max - real_min) / disp_width;
    double scale_imag = (imag_max - imag_min) / disp_height;
    for (int y = 0; y < disp_height; y++) {
      for (int x = 0; x < disp_width; x++) {
        double c_real = real_min + ((double) x * scale_real);
        double c_imag = imag_min + ((double) y * scale_imag);
        mandelbrot[y][x] = 256 - cal_pixel(c_real, c_imag);
      }
    }
    write_P2_PGM(mandelbrot, "mm.pgm", 256);
  }
}
```

Listing C.2: Java code - sequential *mandelbrot* part 2.

# Appendix D

# Parallel Mandelbrot in C using Open MPI library.

```
/************************************************
A simple MPI program to create Mandelbrot picture using blocking
 ↪  send/recv method. Master
receives calculations done by each worker in the correct sequence by
 ↪  waiting for each
worker to complete in an ascending order.

The program contains one master (process 0) and multiple workers. Each
 ↪  worker calculates
the amount of work the needs to be done by dividing the total height by
 ↪  the number of
workers in the program. To make it optimal, the remainder of the
 ↪  division is distributed
between the workers by 1 or 0 row each.

After each worker is done with the calculation, it sends its block of
 ↪  work to the master.
Master writes the Mandelbrot file after all workers are done.

@Author Cabel Dhoj Shrestha
Oct 2015
************************************************/
```

Listing D.1: Parallel Mandelbrot in C using OpenMPI - part 1.

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <sys/time.h>
/*
 Function to calculate real time as UNIX time command does not work for
 ↪  parallel programs.
*/
void printTime(int rank, char *task, int size, struct timeval t1,
 ↪  struct timeval t2) {
  long l;
  long secs, usecs;
  l = t2.tv_sec*1000000+t2.tv_usec-(t1.tv_sec*1000000+t1.tv_usec);
  secs = l/1000000;
  usecs = l%1000000;
  printf("%d:%s, time:,%d,%ld.%ld\n", rank, task, size, secs,usecs);
}

/*
 The main method.
*/
int main(int argc, char *argv[]) {
  if (argc != 9) {
    printf("Usage:\n Mandelbrot width height real-min real-max imag-min
     ↪  imag-max mapfile outfile\n");
    exit(1);
  }
  struct timeval t1, t2, comm1, comm2, compu1, compu2, write1, write2;
  gettimeofday(&t1, NULL);
  int disp_width, disp_height;
  float real_min, real_max, imag_min, imag_max, scale_real, scale_imag;
  float c_real, c_imag;
  int count, max;
  float z_real, z_imag;
  float temp, lengthsq;
  FILE *f;
  int x,y,i,worker;
  char str[256];
  int rank, size, block_height;
  int *block;
  /*----- End of variable declarations. -----*/
```

Listing D.2: Parallel Mandelbrot in C using OpenMPI - part 2.

```
/* Decode arguments */
disp_width  = atoi(argv[1]);
disp_height = atoi(argv[2]);
real_min = atof(argv[3]);
real_max = atof(argv[4]);
imag_min = atof(argv[5]);
imag_max = atof(argv[6]);
/*====== MPI code begins from here. =====*/
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
/*=========== MASTER ===========*/
if (rank == 0) {
  int map[3][257];
  int *pic = (int *)malloc(disp_width*disp_height*sizeof(int*));
  for(worker = 1; worker < size; worker++) {
    block_height = disp_height/(size - 1) + (worker <=
     ↪ disp_height%(size-1) ? 1 : 0);
    block = (int*)malloc(disp_width * block_height * sizeof(int*));
    /*------- Start-time Block ---------*/
    gettimeofday(&comm1, NULL);
    /*---------------X-----------------*/
    MPI_Recv(&block[0], block_height * disp_width, MPI_INT, worker,
     ↪ 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    /*------- End-time Block ---------*/
    gettimeofday(&comm2, NULL);
    printTime(rank, "Communication", size, comm1, comm2);
    /*---------------X-----------------*/
    /* Write the block calculation to pic array to write the
     ↪ Mandelbrot later.*/
    for (y=0; y<block_height; y++) {
      for (x=0; x<disp_width; x++) {
        pic[((y + ((worker-1)*block_height)) * disp_width) + x] =
         ↪ block[(y * disp_width) + x];
      }
    }
  }
```

Listing D.3: Parallel Mandelbrot in C using OpenMPI - part 3.

```
    /*-------- Start-time Block ----------*/
    gettimeofday(&write1, NULL);
    /*---------------X------------------*/
    /* Load the required colour map file */
    f = fopen(argv[7],"r");
    for(i=0;i<=256;i++) {
      fgets(str,1000,f);
      sscanf(str,"%d %d %d",&(map[0][i]),&(map[1][i]),&(map[2][i]));
    }
    fclose(f);
    /* Creating P6 format instead of P3 ppm file to speed up. */
    f = fopen(argv[8],"wb");
    fprintf(f,"P6\n%d %d\n255\n",disp_width,disp_height);

    for (y=0; y<disp_height; y++) {
      for (x=0; x<disp_width; x++) {
        static unsigned char color[3];
        color[0] = map[0][pic[(y * disp_width) + x]];
        color[1] = map[1][pic[(y * disp_width) + x]];
        color[2] = map[2][pic[(y * disp_width) + x]];
        (void) fwrite(color, 1, 3, f);
      }
    }
    fclose(f);
    free(pic);
    /*-------- End-time Block ----------*/
    gettimeofday(&write2, NULL);
    printTime(rank, "I/O", size, write1, write2);
    /*---------------X------------------*/
  }
  /*====== End Master =======*/

  /*========= SLAVES =======*/
  else {
    /*-------- Start-time Block ----------*/
    gettimeofday(&compu1, NULL);
    /*-----------------X------------------*/
    /* Compute scaling factors */
    scale_real = (real_max-real_min)/disp_width;
    scale_imag = (imag_max-imag_min)/disp_height;
    /* Calculate work to be done. */
    block_height = disp_height/(size - 1) + (rank <=
     ↪ disp_height%(size-1) ? 1 : 0);
    /* Allocate memory. */
    block = (int*)malloc(disp_width * block_height * sizeof(int*));
```

Listing D.4: Parallel Mandelbrot in C using OpenMPI - part 4.

```
    /* Perform Mandelbrot calculation */
    for (y= 0; y< block_height; y++) {
      for (x=0; x<disp_width; x++) {
        c_real = real_min + (x * scale_real);
        c_imag = imag_min + ((y + ((rank-1) * block_height)) *
          ↪ scale_imag);
        max = 256;
        z_real = 0;
        z_imag = 0;
        count =0;
        do {
          temp = z_real * z_real - z_imag * z_imag + c_real;
          z_imag = 2 * z_real * z_imag + c_imag;
          z_real = temp;
          lengthsq = z_real * z_real + z_imag * z_imag;
          count++;
        } while ((lengthsq < 4.0) && (count < max));
        block[(y * disp_width) + x] = count;
      }
    }
    /*------- End-time Block ---------*/
    gettimeofday(&compu2, NULL);
    printTime(rank, "Computation", size, compu1, compu2);
    /*--------------X-----------------*/
    MPI_Send(&block[0], block_height * disp_width, MPI_INT, 0, 0,
      ↪ MPI_COMM_WORLD );
  }
  /*====== End Slaves ======*/
  /*------- End-time Block ---------*/
  gettimeofday(&t2, NULL);
  printTime(rank, "Total Execution", size, t1, t2);
  /*--------------X-----------------*/
  MPI_Finalize();
  return 0;
}
```

Listing D.5: Parallel Mandelbrot in C using OpenMPI - part 5.

# Appendix E

# The Santa Claus Problem in ProcessJ

```
import std.io;
import std.random;

const int N_REINDEER = 9;
const int G_REINDEER = N_REINDEER;
const int N_ELVES = 10;
const int G_ELVES = 3;
const int HOLIDAY_TIME = 100000;
const int WORKING_TIME = 200000;
const int DELIVERY_TIME = 100000;
const int CONSULTATION_TIME = 200000;

// continued on the next page
```

Listing E.1: ProcessJ code - *santa* part 1.

```
protocol Reindeer_msg {
 holiday:    { int id; } //start of vacation postcard; reindeer id
 deer_ready: { int id; } //back from vacation; reindeer id
 deliver:    { int id; } //start of toy delivery; reindeer id
 deer_done:  { int id; } //return from toy delivery; reindeer id
}

protocol Elf_msg {
 working:   { int id; } //start of work shift; elf id
 elf_ready: { int id; } //want to consult Santa; elf id
 waiting:   { int id; } //in the waiting room; elf id
 consult:   { int id; } //consulting; elf id
 elf_done:  { int id; } //end of consultation; elf id
}

protocol Santa_msg {
 reindeer_ready: { }                  //woken up by reindeer
 harness:        { int id; }          //harnessing this reindeer; id
 mush_mush:      { }                  //start of toy delivery
 woah:           { }                  //end of toy delivery
 unharness:      { int id; }          //unharnessing this reindeer; id
 elves_ready:    { }                  //woken up by party of elves
 greet:          { int id; }          //greet this elf; id
 consulting:     { }                  //consulting with elves
 santa_done:     { }                  //end of consultation
 goodbye:        { int id; }          //show elf the door; id
}

protocol Message extends Reindeer_msg, Elf_msg, Santa_msg;

proc void random_wait(long max_wait, long seed)[yield=true] {
  timer t;
  long wait;
  initRandom(seed);
  wait = longRandom();
  t.timeout(wait);
}
```

Listing E.2: ProcessJ code - *santa* part 2.

```
proc void display (chan<Message>.read in) [yield=true]{
  Message msg;
  while (true) {
    msg = in.read();
    switch(msg) {
      case holiday:
        println("
          ↪  Reindeer-" + msg.id + ": on holiday ... wish you were
          ↪  here");
          break;
      case deer_ready:
        println("
          ↪  Reindeer-" + msg.id + ": back from holiday ... ready for
          ↪  work");
        break;
      case deliver:
        println("
          ↪  Reindeer-" + msg.id + ": delivering
          ↪  toys...la-di-da-di-da-di-da");
        break;
      case deer_done:
        println("
          ↪  Reindeer-" + msg.id + ": all toys delivered...want a
          ↪  holiday");
        break;
      case working:
        println("                                 Elf-" + msg.id + ":
          ↪  working");
        break;
      case elf_ready:
        println("                                 Elf-" + msg.id + ": need
          ↪  to consult Santa, ;(");
        break;
      case waiting:
        println("                                 Elf-" + msg.id + ": in the
          ↪  waiting room...");
        break;
      case consult:
        println("                                 Elf-" + msg.id + ": about
          ↪  these toys...??");
        break;

    // continued on the next page
```

Listing E.3: ProcessJ code - *santa* part 3.

```
        case elf_done:
          println("                              Elf-" + msg.id + ":
            ↪ OK...we'll build it, bye...");
          break;
        case reindeer_ready: println("Santa: Ho-ho-ho...the reindeer are
          ↪ back!"); break;
        case harness: println("Santa: harnessing reindeer:" + msg.id);
          ↪ break;
        case mush_mush: println("Santa: mush mush ..."); break;
        case woah: println("Santa: woah...we're back home!"); break;
        case unharness: println("Santa: un-harnessing reindeer:" +
          ↪ msg.id); break;
        case elves_ready: println("Santa: Ho-ho-ho...some elves are
          ↪ here!"); break;
        case greet: println("Santa: hello elf:" + msg.id); break;
        case consulting: println("Santa: consulting with elves...");
          ↪ break;
        case santa_done: println("Santa: OK, all done - thanks!"); break;
        case goodbye: println("Santa: goodbye elf:" + msg.id); break;
      }
    }
  }

proc void p_barrier_knock (const int n, chan<boolean>.read a,
 ↪ chan<boolean>.read b,  chan<boolean>.write knock) [yield=true]{
 while (true) {
   for (int i=0; i<n; i++) {
     boolean any;
     any = a.read();
   }
   knock.write (true);
   for (int i=0; i<n; i++) {
     boolean any;
     any = b.read();
   }
 }
}
```

Listing E.4: ProcessJ code - *santa* part 4.

```
proc void p_barrier (const int n, chan<boolean>.read a,
 ↪  chan<boolean>.read b) [yield=true]{
 while (true) {
   for (int i=0; i<n; i++) {
     boolean any;
     any = a.read();
   }
   for (int i=0; i<n; i++) {
     boolean any;
     any = b.read();
   }
 }
}

proc void syncronize (shared chan<boolean>.write a, shared
 ↪  chan<boolean>.write b) [yield=true]{
 claim (a) a.write(true);
 claim (b) b.write(true);
}

proc void reindeer (const int id, const long seed, barrier
 ↪  just_reindeer, barrier santa_reindeer, shared chan<int>.write
 ↪  to_santa, shared chan<Reindeer_msg>.write report) [yield=true]{
  long my_seed = seed; long wait = HOLIDAY_TIME; long t;
  timer tim;
  while (true) {
    claim (report)
      report.write(new Reindeer_msg{ holiday: id });
    random_wait(wait, my_seed);
    claim (report)
      report.write(new Reindeer_msg{ deer_ready: id });
    sync (just_reindeer); //wait for all deer to return
    claim (to_santa)
      to_santa.write(id); //send id and get harnessed
    sync (santa_reindeer);
    claim (report)
      report.write(new Reindeer_msg{ deliver: id});
    sync (santa_reindeer);
    claim (report)
      report.write(new Reindeer_msg{ deer_done: id });
    claim (to_santa)
      to_santa.write(id);
  }
}
```

Listing E.5: ProcessJ code - *santa* part 5.

```
proc void elf (const int id, const long seed, shared
 ↪ chan<boolean>.write elves_a, shared chan<boolean>.write elves_b,
 ↪ shared chan<boolean>.write santa_elves_a, shared
 ↪ chan<boolean>.write santa_elves_b, shared chan<int>.write to_santa,
 ↪ shared chan<Elf_msg>.write report)[yield=true] {

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

  while (true) {
    claim (report)
      report.write(new Elf_msg{ working: id});
    random_wait(wait, my_seed);
    claim (report)
      report.write(new Elf_msg{ elf_ready: id });
    syncronize (elves_a, elves_b);
    claim (to_santa)
      to_santa.write(id);
    syncronize (santa_elves_a, santa_elves_b);
    claim (report)
      report.write(new Elf_msg{ consult: id });
    syncronize (santa_elves_a, santa_elves_b);
    claim (report)
      report.write(new Elf_msg{ elf_done: id });
    claim (to_santa)
      to_santa.write (id);
  }
}
```

Listing E.6: ProcessJ code - *santa* part 6.

```
proc 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) [yield = true]{
  long my_seed = seed; long t, wait;
  timer tim;

  while (true) {
    int id;
    boolean any;
    pri alt {
      id = from_reindeer.read(): {  // Reindeer ready
        claim (report) {
          report.write(new Santa_msg{ reindeer_ready: });
          report.write(new Santa_msg{ harness: id });
        }
        for (int i=0; i< G_REINDEER-1; i++) {
          id = from_reindeer.read();
          claim (report)
            report.write(new Santa_msg{ harness: id });
        }
        claim (report)
          report.write(new Santa_msg{mush_mush:});
        sync (santa_reindeer);
        t = tim.read();
        tim.timeout (1000);
        claim (report)
          report.write (new Santa_msg{woah:});
        sync (santa_reindeer);
        for (int i=0; i< G_REINDEER; i++) {
          id = from_reindeer.read({
            claim (report)
              report.write (new Santa_msg{ unharness:id });
          });
        }
      }
      // continued on the next page
```

Listing E.7: ProcessJ code - *santa* part 7.

```
        any = knock.read (): { // 3 Elves ready
          claim (report)
            report.write (new Santa_msg{ elves_ready: });

          for (int i=0; i<G_ELVES; i++) {
            id = from_elf.read();
            claim (report)
              report.write (new Santa_msg{ greet: id });
          }
          claim (report)
            report.write (new Santa_msg{ consulting: });
          syncronize (santa_elves_a, santa_elves_b);
          t = tim.read();
          tim.timeout (1000);
          claim (report)
            report.write (new Santa_msg{ santa_done: });
          syncronize (santa_elves_a, santa_elves_b);
          for (int i=0; i<G_ELVES; i++) {
            id = from_elf.read ({
            claim (report)
              report.write(new Santa_msg{ goodbye: id });
            });
          }
        }
      }
    }
  }
```

Listing E.8: ProcessJ code - *santa* part 8.

```
proc void main(string[] args)[yield=true] {
  timer tim;
  long seed;
  seed = tim.read();
  seed = (seed >> 2) + 42;

  barrier just_reindeer, santa_reindeer;

  shared write chan<boolean> elves_a, elves_b;
  chan<boolean> knock;
  shared write chan<boolean> santa_elves_a, santa_elves_b;
  shared write chan<int> reindeer_santa, elf_santa;
  shared write chan<Message> report;
  par {
    par enroll (santa_reindeer) {
      santa (seed + (N_REINDEER + N_ELVES), knock.read,
      reindeer_santa.read, elf_santa.read, santa_reindeer,
      santa_elves_a.write, santa_elves_b.write, report.write);

      par for (int i=0; i<N_REINDEER; i++) enroll (just_reindeer,
       ↪ santa_reindeer)
        reindeer (i, seed + i, just_reindeer, santa_reindeer,
         ↪ reindeer_santa.write, report.write);
    }

    par for (int i=0; i<N_ELVES; i++)
      elf (i, N_REINDEER + (seed + i), elves_a.write, elves_b.write,
       ↪ santa_elves_a.write, santa_elves_b.write, elf_santa.write,
       ↪ report.write);

    display (report.read);
    p_barrier_knock (G_ELVES, elves_a.read, elves_b.read, knock.write);
    p_barrier (G_ELVES + 1, santa_elves_a.read, santa_elves_b.read);
  }
}
```

Listing E.9: ProcessJ code - *santa* part 9.

# Appendix F

# Full Adder in ProcessJ

```
/**
 * This program simulates the Full Adder as described in the Visual
   ↪  Occam: High Level
 * Visualization and Design of Process by Mikola Michal Slomka.
*/
import std.io;

/**
 * The main method (execution point).
 */
proc void main(string[] args)[yield=true] {
  myMain();
}

/**
 * Not gate.
 * @param in: in signal.
 * @param out: out channel.
 */
proc void notGate(chan<boolean>.read in, chan<boolean>.write
 ↪  out)[yield=true]{
  boolean x=false;
  x = in.read();
  out.write(!x);
}
```

Listing F.1: ProcessJ code - *fullAdder* part 1.

```
/**
 * Or gate.
 * @param in1: first in signal.
 * @param in2: second in signal.
 * @param out: out channel.
 */
proc void orGate(chan<boolean>.read in1, chan<boolean>.read in2,
 ↪  chan<boolean>.write out)[yield=true]{
  boolean x=false, y=false;
  par{
    x = in1.read();
    y = in2.read();
  }
  out.write(x || y);
}

/**
 * And Gate.
 * @param in1: first in signal.
 * @param in2: second in signal.
 * @param out: out channel.
 */
proc void andGate(chan<boolean>.read in1, chan<boolean>.read in2,
 ↪  chan<boolean>.write out)[yield=true]{
  boolean x=false, y=false;
  par{
    x = in1.read();
    y = in2.read();
  }
  out.write(x && y);
}
```

Listing F.2: ProcessJ code - *fullAdder* part 2.

```
/**
 * Nand Gate.
 * @param in1: first in signal.
 * @param in2: second in signal.
 * @param out: out channel.
 * @return void.
 */
proc void nandGate(chan<boolean>.read in1, chan<boolean>.read in2,
 ↪  chan<boolean>.write out)[yield=true]{
  chan<boolean> a;
  par{
    andGate(in1, in2, a.write);
    notGate(a.read, out);
  }
  return;
}

/**
 * Multiplexer.
 * @param in1: in signal.
 * @param out1: first out channel.
 * @param out2: second out channel.
 * @return void.
 */
proc void muxGate(chan<boolean>.read in, chan<boolean>.read out1,
 ↪  chan<boolean>.write out2)[yield=true]{
  boolean x=false;
  x = in.read();
  par{
    out1.write(x);
    out2.write(x);
  }
   return;
}
```

Listing F.3: ProcessJ code - *fullAdder* part 3.

```
/**
 * xor Gate.
 * @param in1: in signal.
 * @param out1: first out channel.
 * @param out2: second out channel.
 */
proc void xorGate(chan<boolean>.read in1, chan<boolean>.read in2,
 ↪  chan<boolean>.write out)[yield=true]{
  chan<boolean> a, b, c, d , e, f, g, h, i;
  par{
    muxGate(in1, a.read, b.write);
    muxGate(in2, c.read, d.write);
    nandGate(b.read, d.read, e.write);
    muxGate(e.read, f.read, g.write);
    nandGate(a.read, f.read, h.write);
    nandGate(c.read, g.read, i.write);
    nandGate(h.read, i.read, out);
  }
}
/**
 * Single Adder.
 * @param in1: in signal.
 * @param in2: in signal.
 * @param in3: in signal.
 * @param result: first out signal.
 * @param carry: second out signal. (Remainder).
 */
proc void oneBitAdder(chan<boolean>.read in1, chan<boolean>.read in2,
 ↪  chan<boolean>.read in3, chan<boolean>.write result,
 ↪  chan<boolean>.write carry)[yield=true]{
  chan<boolean> a, b, c, d, e, f, g, h, i, j, k;
  par{
    muxGate(in1, a.read, b.write);
    muxGate(in2, c.read, d.write);
    xorGate(a.read, c.read, e.write);

    muxGate(e.read, f.read, g.write);
    muxGate(in3, h.read, i.write);
    xorGate(f.read, h.read, result);

    andGate(g.read, i.read, j.write);
    andGate(b.read, d.read, k.write);
    orGate(j.read, k.read, carry);
  }
}
```

Listing F.4: ProcessJ code - *fullAdder* part 4.

```
/**
 * fourBitadder
 */
proc void fourBitAdder(chan<boolean>.read inA0, chan<boolean>.read
 ↪  inA1, chan<boolean>.read inA2, chan<boolean>.read inA3,
 ↪  chan<boolean>.read inB0, chan<boolean>.read inB1,
 ↪  chan<boolean>.read inB2, chan<boolean>.read inB3,
 ↪  chan<boolean>.read inCarry, chan<boolean>.write result0,
 ↪  chan<boolean>.write result1, chan<boolean>.write result2,
 ↪  chan<boolean>.write result3, chan<boolean>.write
 ↪  carry)[yield=true]{
  chan<boolean> a, b, c;
  par{
    oneBitAdder(inA0, inB0, inCarry, result0, a.write);
    oneBitAdder(inA1, inB1, a.read, result1, b.write);
    oneBitAdder(inA2, inB2, b.read, result2, c.write);
    oneBitAdder(inA3, inB3, c.read, result3, carry);
  }
}
/**
 * 8-bit Adder.
 */
proc void eightBitAdder(chan<boolean>.read inA0, chan<boolean>.read
 ↪  inA1, chan<boolean>.read inA2, chan<boolean>.read inA3,
 ↪  chan<boolean>.read inA4, chan<boolean>.read inA5,
 ↪  chan<boolean>.read inA6, chan<boolean>.read inA7,
 ↪  chan<boolean>.read inB0, chan<boolean>.read inB1,
 ↪  chan<boolean>.read inB2, chan<boolean>.read inB3,
 ↪  chan<boolean>.read inB4, chan<boolean>.read inB5,
 ↪  chan<boolean>.read inB6, chan<boolean>.read inB7,
 ↪  chan<boolean>.read inCarry, chan<boolean>.write result0,
 ↪  chan<boolean>.write result1, chan<boolean>.write result2,
 ↪  chan<boolean>.write result3, chan<boolean>.write result4,
 ↪  chan<boolean>.write result5, chan<boolean>.write result6,
 ↪  chan<boolean>.write result7, chan<boolean>.write
 ↪  outCarry)[yield=true]{
  chan<boolean> a;
  par{
    fourBitAdder(inA0, inA1, inA2, inA3,
                 inB0, inB1, inB2, inB3,
                 inCarry,
                 result0, result1, result2, result3,
                 a.write);
    fourBitAdder(inA4, inA5, inA6, inA7,
                 inB4, inB5, inB6, inB7,
                 a.read,
                 result4, result5, result6, result7,
                 outCarry);
  }
}
```

Listing F.5: ProcessJ code - *fullAdder* part 5.

```
proc void myMain()[yield=true]{
  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;
  //Result of addition.
  boolean f0, f1, f2, f3, f4, f5, f6, f7;
  boolean c, inC;

  //Choose numbers here!
  p7 = false;
  p6 = true;
  p5 = false;
  p4 = false;
  p3 = false;
  p2 = true;
  p1 = false;
  p0 = false;

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

  par{
    //Our first number.
    a7.write(p7);
    a6.write(p6);
    a5.write(p5);
    a4.write(p4);
    a3.write(p3);
    a2.write(p2);
    a1.write(p1);
    a0.write(p0);
    // continued on the next page.
```

Listing F.6: ProcessJ code - *fullAdder* part 6.

```
      //Our second number.
      b7.write(q7);
      b6.write(q6);
      b5.write(q5);
      b4.write(q4);
      b3.write(q3);
      b2.write(q2);
      b1.write(q1);
      b0.write(q0);
      //Initial Carry
      inCarry.write(inC);

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

      f0 = r0.read();
      f1 = r1.read();
      f2 = r2.read();
      f3 = r3.read();

      f4 = r4.read();
      f5 = r5.read();
      f6 = r6.read();
      f7 = r7.read();

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

    println("Carry was: " + c);

    return;
  }
```

Listing F.7: ProcessJ code - *fullAdder* part 7.

# Bibliography

[Bel05] Abhijit Belapurkar. CSP for Java programmers, Part 2: Concurrent programming with JCSP, June 2005. Available at: `http://www.ibm.com/developerworks/java/library/j-csp2/`.

[BL] Blaise Barney and Lawrence Livermore National Laboratory. POSIX Threads Programming. Available at: `https://computing.llnl.gov/tutorials/pthreads/`.

[Bla16] Blaise Barney and Lawrence Livermore . Message Passing Interface (MPI), 2016. Available at: `https://computing.llnl.gov/tutorials/mpi/#What`.

[BLC02] E. Bruneton, R Lenglet, and T. Coupaye. Asm: a code manipulation tool to implement adaptable systems. In *Adaptable and extensible component systems*, November 2002.

[Bry06] Bryan Carpenter and Aamir Shafi and Mohsan Jameel and Guillermo Lopez Taboada and HPC Lab and SEECS and NUST. MPJ Express, 2006. Available at: `http://mpj-express.org/`.

[BWS05a] F.R.M. Barnes, P.H. Welch, and A.T. Sampson. Barrier synchronisations for occam-pi. In Hamid R. Arabnia, editor, *Proceedings of PDPTA 2005*, pages 173–179, Las Vegas, Nevada, USA, June 2005. CSREA press.

[BWS05b] F.R.M. Barnes, P.H. Welch, and A.T. Sampson. Barrier synchronisations for occam-pi. In Hamid R. Arabnia, editor, *Proceedings of the 2005 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2005)*, pages 173–179, Las Vegas, Nevada, USA, June 2005. CSREA press. ISBN: 1-932415-58-0.

[Eri11] Eric Bruneton. *ASM 4.0 - A Java bytecode engineering library*, 2011.

[For00] Formal Systems (Europe) Ltd., 3, Alfred Street, Oxford. OX1 4EH, UK. *FDR2 User Manual*, May 2000.

[Fou] Python Software Foundation. Thread-based parallelism. Available at: `https://docs.python.org/3/library/threading.html`.

[Hoa78] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[Lab] Argonne National Laboratory. CH3 And Channels. Available at: `https://wiki.mpich.org/mpich/index.php/CH3_And_Channels`.

[Lov03] Robert Love. The linux process scheduler. November 2003.

[Mil99] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999. ISBN: 0-52165-869-1.

[Moo99] J. Moores. CCSP – a Portable CSP-based Run-time System Supporting C and occam. In B.M. Cook, editor, *Architectures, Languages and Techniques for Concurrent Systems*, volume 57 of *Concurrent Systems Engineering series*, pages 147–168, Amsterdam, The Netherlands, April 1999. WoTUG, IOS Press. ISBN: 90-5199-480-X.

[mpi] MPICH: High-Performance Portable MPI. Available at: `https://www.mpich.org/`.

[oIURC] The Trustees of Indiana University, Indiana University Research, and Technology Corporation. Open MPI: Open Source High Performance Computing. Available at: `https://www.open-mpi.org/`.

[OW04] Scott Oaks and Henry Wong. *Java Threads*. OReilly Media, Inc., 2004.

[PK09] Jan B. Pedersen and Brian Kauke. Resumable Java Bytecode - Process Mobility for the JVM. In *Communicating Process Architectures 2009*. IOS Press, 2009.

[PS13] Jan B. Pedersen and Marc L. Smith. ProcessJ: A Possible Future of Process-Oriented Design. In Jan F. Broenink Kevin Chalmers Jan B. Pedersen Peter H. Welch, Frederick R. M. Barnes and Adam T. Sampson, editors, *Communicating Process Architectures 2013*, pages 133–156, November 2013.

[PS14] Jan B. Pedersen and Andreas Stefik. Towards Millions of Processes on the JVM. 2014.

[PW] Jan B. Pedersen and Peter H. Welch. The symbiosis of concurrency and verification: Teaching and case studies.

[RSB12] Carl G. Ritson, Adam T. Sampson, and Frederick R.M. Barnes. Multicore scheduling for lightweight communicating processes. *Science of Computer Programming*, 77(6):727–740, June 2012. Article in press. doi:10.1016/j.scico.2011.04.006.

[SP11] Matthew Sowders and Jan B. Pedersen. Mobile process resumption in java without bytecode rewriting. In *Proceedings of Parallel and Distributed Processing Techniques and Applications (PDPTA'11)*, July 2011.

[Sut09] Herb Sutter. The free lunch is over - a fundamental turn toward concurrency in software. *Communications of the ACM*, 2009.

[SWB90] J.P.E. Sunter, K.C.J. Wijbrans, and A.W.P. Bakkers. Cooperative Priority Scheduling in Occam. In H.S.M. Zedan, editor, *Proceedings of the 13th occam User Group Technical Meeting: Real-Time Systems with Transputers*, Transputer and Occam Engineering, pages 175–185, Amsterdam, The Netherlands, September 1990. IOS Press. ISBN: 90-5199-041-3.

[Ter13] Terence Parr 2013. StringTemplate, 2013. Available at: `http://www.stringtemplate.org/`.

[WA99] P.H. Welch and P.D. Austin. The JCSP (CSP for Java) Home Page, 1999. Available at: `http://www.cs.kent.ac.uk/projects/ofa/jcsp/`.

[Wel99] P.H. Welch. CSP for Java (JCSP), 1999. Available at: `http://www.cs.kent.ac.uk/projects/ofa/jcsp/`.

[Wel00] P.H. Welch. Process Oriented Design for Java – Concurrency for All. In *PDPTA 2000*, volume 1, pages 51–57. CSREA Press, June 2000. ISBN: 1-892512-52-1.

# Curriculum Vitae

Graduate College

University of Nevada, Las Vegas

Cabel Dhoj Shrestha

Degrees:

Bachelor of Science in Computer Engineering 2007

University of Cebu, Cebu, Philippines

Thesis Title: The JVMCSP Runtime and Code Generator for ProcessJ in Java

Thesis Examination Committee:

Chairperson, Dr. Jan 'Matt' Pedersen, Ph.D.

Committee Member, Dr. Kazem Taghva, Ph.D.

Committee Member, Dr. Andreas Stefik, Ph.D.

Graduate Faculty Representative, Dr. Hualiang (Harry) Teng, Ph.D.