University of Southampton

Faculty of Physical and Applied Sciences

Electronics and Computer Science

# JCC: Java framework for building concurrent data structures using combinators

by

Ales Cirnfus

September 6th, 2013

A dissertation submitted in partial fulfilment of the degree of

MSc in Software Engineering

by examination and dissertation

# List of Figures

# Acronyms

| | |
|---|---|
| **CAS** | compareAndSet. 5, 9, 23, 28 |
| **CSP** | Communicating Sequential Processes. 4 |
| **JCC** | Java Concurrent Combinators. 2, 3, 8, 9, 11, 13–17, 23–26, 28, 30–33, 35 |
| **JUC** | java.util.concurrent. 1, 4, 5, 20, 21, 23 |
| **JVM** | Java Virtual Machine. 4, 5 |
| **KPN** | Kahn Process Networks. 4 |
| **LTS** | labelled transition system. 2, 7, 8 |
| **SOS** | structural operational semantics. 8 |
| **STM** | software transactional memory. 4, 5 |

## Abstract

For a long time, concurrent programming was considered the prerogative of experts. However, the recent shift towards multi-core hardware architectures has made concurrency an essential part of the software developer's toolkit. Consequently, considerable efforts have been put into creating tools for safe and reliable concurrent programming. Yet, a universally accepted concurrency model is still missing and the search for this holy grail of concurrency goes on.

This paper describes the design and development of Java Concurrent Combinators (JCC), a framework rooted in the formalisms presented by the Wire Calculus which aims to provide a high-level, expressive and composable abstraction over the thread-based concurrency model on the Java Virtual Machine (JVM). Manifold wire and data combinators are defined and implemented as part of the framework. The power of composition is demonstrated by creating complex concurrent data structures from the core combinators and encapsulating them into Java components.

While only in its infancy, the JCC framework represents a promising step towards defining a new concurrency model that has the potential to alleviate many problems associated with threads and provide a high level of determinism, expressiveness and composability.

## Acknowledgements

I would like to thank the following people for their help and support throughout this project:

My supervisor, Dr Julian Rathke, for helping me identify a suitable concurrency-related project and giving me honest feedback and suggestions that guided me in the right direction.

Dr Pawel Sobociński for providing insight into the Wire Calculus and describing the intended semantics of individual wires.

# Contents

# Chapter 1

# Introduction

## 1.1 Background Information

Concurrency is a challenging area of computer science that has been subject to intensive research since the mid 1960s. While many concurrency models have been proposed, there is one approach that is dominant in modern general-purpose languages - shared memory with threads [17]. Yet, it is widely acknowledged that there are many issues with the thread-based concurrency model. As concurrent computations can interleave in a non-deterministic way, multi-threaded programs become easily susceptible to undesirable side-effects such as deadlocks, race conditions and starvation.

To make things worse, for a long time, concurrency was treated as a second-class citizen in many general-purpose programming languages. The reason was simple. The majority of software was written in a purely sequential way and performance gains were sustained due to advances in chip speeds and transistor density; phenomenon known as Moore's Law[1]. The remaining minority of applications that required concurrency were mostly so-called "embarrassingly parallel systems" such as web servers [17]. Concurrent processes in these systems were essentially disjoint and data sharing was managed through database abstractions which provided their own concurrency mechanisms such as transactions. As a result, concurrency support in general-purpose languages such as Java remained limited to low-level primitives (e.g. synchronized, wait(), notify()) which were difficult to use correctly.

However, in recent years, concurrency issues have received a great deal of attention due to the predicted end of Moore's Law. As explained by Kaku [13], the power of silicon microprocessors cannot double every two years indefinitely. We have reached the point where these exponential increases are no longer sustainable due to the limits set out by the laws of thermodynamics and quantum physics. Consequently, for the foreseeable future, performance boosts are to be acquired through multi-core microprocessors rather than faster single-core ones [14][28]. While adding power through multiple processors seems to be straightforward for chip manufactures, it has far-reaching implications for software developers. Suddenly, concurrency becomes a key ingredient in the production of efficient software as only well-written concurrent applications can harvest the power offered by parallel cores.

Ubiquitous concurrency means that programmers require simpler, easy-to-use abstractions over threads to write correct and efficient concurrent applications [14]. These demands have led to the introduction of higher-level concurrency libraries such as java.util.concurrent (JUC) in Java which provide a wide range of concurrent primitives and data structures for fine-grained concurrency. However, such libraries typically only tackle standard

[1]`http://en.wikipedia.org/wiki/Moore%27s_law`

concurrency problems and are hard to extend by composition [30]. The next step is to develop a collection of expressive and composable components which encapsulate fine-grained concurrent operations. Such components can be freely combined to create complex concurrent data structures in a safe and efficient way.

## 1.2 Aims and Objectives

The primary objective of this project is to develop a framework of JCC which provides a set of standard components that can be combined both vertically and horizontally to create complex concurrent data structures with minimum effort. The standard combinators defined in the framework shall encapsulate common concurrency patterns (e.g. optimistic retry loop, backoff scheme) and provide implementations of general concurrent algorithms (e.g. lock-free stacks and queues). Communication between combinators shall be provided through a message passing system with particular emphasis on atomicity and failure handling. The secondary objective of the project is to build complex concurrent data structures from the core combinators provided in the framework and test them for correctness and performance.

The JCC framework is inspired by ideas introduced by Dr Pawel Sobociński in his work on the Wire Calculus [25]. The calculus consists of a number constants defined in terms of a two-labelled transition system (LTS). Such constants can be thought of as components with ports along the left and right boundaries. These components can be composed over two binary operators. The sequential synchronisation operator wires two constants together along a common boundary while the tensor operator provide a parallel non-communicating composition. There are also dynamic operators such as a CSP-like choice operator which define the behaviour of components.

In order to achieve these goals an initial set of objectives was established at the beginning of the process. These objectives can be found in the Project Brief (Appendix A).

## 1.3 Project Scope

As stated in the previous section, the scope of this project is limited to the implementation of core concepts introduced in the Wire Calculus and testing the resulting framework for correctness and performance.

It is not within the scope of this project to implement a graphical drag-and-drop interface for the JCC framework or realize any other features mentioned in the Future Work chapter.

## 1.4 Overview of Chapters

### 1.4.1 Background Research and Literature Review

The purpose of the background research and literature review is to provide an overview of different concurrency models and evaluate their relevance to the proposed framework. This chapter also reviews concurrent algorithms that could be implemented with JCC.

### 1.4.2 System Analysis and Design Decisions

This chapter translates the key concepts specified in the Wire Calculus into system requirements and explores how these requirements could be implemented.

### 1.4.3 Development of the JCC Framework

Beginning with an overview of the development environment and technologies used in the implementation of the JCC Framework, this chapter describes how the core classes as well as specific combinators were implemented.

### 1.4.4 Building Complex Concurrent Data Structures with JCC

This chapter focuses practical uses of the JCC framework and shows how the core combinators can be composed into useful concurrent data structures and message-passing circuits.

### 1.4.5 Correctness and Performance Testing

This chapter explores performance overheads that combinators exhibit in comparison to hand-coded concurrent data structures and also assesses if the JCC combinators behave in the expected way.

### 1.4.6 Future Work

The purpose of this chapter is to suggest how the project could evolve in the future.

### 1.4.7 Evaluation

This chapter reflects on the way the project was managed and whether the key objectives were met. It also includes a personal evaluation.

### 1.4.8 Conclusion

This chapter summarises the key findings of the project.

Chapter 2

# Background Research and Literature Review

## 2.1 Concurrency

There seems to be very little consensus about how to do concurrency 'the right way'. The majority of popular programming languages such as Java, C/C++, C#, Python and Ruby provide concurrency support by giving multiple threads (lightweight processes) concurrent access to shared, mutable state. Other languages, Erlang being the most prominent, opt for message-passing concurrency where communication between processes is accomplished by sending and receiving immutable messages; typically asynchronously without blocking. As opposed to shared-state concurrency, processes in the message-passing paradigm run in isolation and do not share any state. Other alternative concurrency paradigms include software transactional memory (STM) and dataflow concurrency.

Considering the nature of this project the primary focus of the research is on concurrency paradigms implemented either directly in the Java language or other languages that execute on the JVM.

### 2.1.1 Shared-State Concurrency

The main issue with shared-state concurrency is mutability [3]. Sharing mutable state without coordination may yield unpredictable results as one thread can modify data which is currently being used by another thread [7][27]. Therefore, coordination constructs such as locks must be used to synchronise access to shared state. While synchronisation through locking establishes safety it may introduce issues regarding liveness (e.g. deadlock, starvation). These problems make shared-state concurrency notoriously hard and lead to criticism of the paradigm. For example, Lee et al. [17] suggests that the level of non-determinism exhibited by threads renders this concurrency model unusable. Lee argues for the development of coordination languages built on formalisms such as Communicating Sequential Processes (CSP) [11] and Kahn Process Networks (KPN) [12] which should provide strong determinism with non-deterministic behaviour introduced explicitly only when needed. While other researches [23][26][8] also acknowledge the issues associated with shared-state concurrency they believe that high-level abstractions can be created on top of threads.

Focusing on Java's support for shared-state concurrency, primitives such as *synchronized* and *volatile* are an integral part of the language while higher-level utilities are implemented in additional libraries. The *java.util.concurrent (JUC)* library [16] provides a framework of advanced synchronisers (e.g. Reentrant Locks, Barriers, Latches) as well as numerous concurrent collections such as *CopyOnWriteArrayList*, *Concurren-*

*tHashMap* and *ConcurrentLinkedQueue*. There are also implementations of Thread Pools, Atomic variables, Futures and other building blocks essential for fine-grained concurrency. The benefits of JUC are highlighted by [7] and [27]:

- JUC makes programming with threads easier and more efficient
- JUC offers more flexible synchronisers with finer granularity
- JUC provides concurrent data structures with increased scalability

Another notable benefit of JUC is the excellent documentation[1] and also the availability of other resources [7][16][27]. However, industry-strength libraries such as JUC do not represent a silver bullet when it comes to solving problems with shared-state concurrency. Lee et al. [17] claims that pruning non-determinism away from threads will never yield satisfactory results as deterministic aims cannot be achieved using non-deterministic means. Turon [30] acknowledges the importance of libraries for fined-grained concurrency but argues that they are typically very conservative and only cater for the most common needs. He also highlights issues with their expressiveness and composability. Yet more criticism comes from Bonér [3] who suggests that shared-state concurrency hardly ever provides the simplest solution to a specific problem and promotes the development of multi-paradigm concurrency toolkits such as Akka[2].

### 2.1.2 Software Transactional Memory

The *java.util.concurrent.atomic* includes classes capable of performing atomic state transitions - compareAndSet (CAS) operations - on numbers and object references [7]. These capabilities are utilised in many of JUC's collections which provide atomic operations such as *enqueue* and *dequeue*. Yet, as pointed out by Turon [30], it is impossible to compose multiple atomic operations into a single atomic block.

STM offers a concurrency paradigm which treats the memory as a transactional dataset [3]. Similarly to databases, transactions in STM *begin*, *commit* or *abort/rollback* and are typically performed optimistically with automatic retries upon collisions [24]. Unlike memory locking, STM transactions are composable and therefore provide a solution for the k-word CAS problem mentioned by Turon [30].

There have been numerous attempts to implement STM for the JVM. DSTM2 [10] implements STM in a pure Java library without any changes to the JVM or the compiler. Classes that are required to participate in transactions must implement *@atomic* interface. This rather limits the use of DSTM2 as does not support existing libraries and native classes.

This limitation is overcome in the Deuce framework [15] which does not require marking of atomic classes. Instead, an *@Atomic* annotation is used at the method to level to indicate atomic transactions. Deuce requires execution with a specialised Java agent loaded via the *-javagent:* option. This enables Deuce to instrument classes as they are loaded to the JVM.

Other JVM languages such as Clojure and Scala[3] also provide support for STM.

### 2.1.3 Message-Passing Concurrency

Message-passing concurrency is rooted in theoretical models such CSP [11], join calculus [6] and the Actor model [1]. One of the fundamental principles of message-based concurrency is mutual exclusion [1]. The only way

[1] `http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html`
[2] `http://akka.io/`
[3] http://nbronson.github.io/scala-stm/

individual processes can communicate with each other is by exchanging immutable messages. The 'share nothing' principle makes message-passing paradigm an attractive alternative to shared-state concurrency. Firstly, there is no need for synchronisation and, secondly, the model lends itself well to distributed computing [8]. It is therefore unsurprising that there have recently been efforts to implement message-based concurrency models for the JVM platform.

The Scala Actor library [8] is one of the most prominent examples. It introduces an actor-based concurrency model for asynchronous message passing with two types of actors; thread-based and event-based. Thread-based actors waiting to receive a message suspend the underlying thread and keep the full execution stack. In comparison, waiting event-based actors maintain its execution state as a continuation closure. A terminating closure throws a special type of exception which allows the stack of the executing thread to be unwound and reused by another actor. This approach improves scalability as continuation closures have a smaller memory footprint than threads.

Another interesting implementation of the Actor model on the JVM is JCoBox [23]. Every object in the JCoBox framework belongs to a CoBox - an isolated actor-like component. Concurrency in CoBoxes is realised by control flows called tasks which are cooperatively scheduled. Therefore, there is always only one active task in a CoBox which gains exclusive control of the local state. Importantly, each task is owned by its CoBox and is not allowed to 'leave' during its lifetime. Tasks differentiate between *local* and *far references.* A *local reference* refers to an object in the same CoBox as the task belong to and, intuitively, *far references* reference objects outside the current CoBox. A method invocation on a *far reference* object is executed asynchronously by creating a new task in the corresponding CoBox and returning a *Future.* To ensure state isolation during communication JCoBox recognises three types of objects. *Standard* objects are passed by reference and are treated as *far references* by tasks in different CoBoxes. *Immutable* objects are not conceptually bound to any CoBox and are treated as *local references* in all CoBoxes. *Transfer* objects are always *local* and are deep-copied if passed to another CoBox. This model conforms to the mutual exclusion principle which is essential for the Actor model [1]. In fact, Schäfer and Poetzsch-Heffter [23] point out how difficult it is to achieve state isolation in the object-oriented (OO) paradigm as synchronous method calls, the primary communication mechanism in OO languages, clearly break the 'share nothing' principle. For example, the aforementioned Scala actors support both asynchronous message passing and synchronous method calls. There are also no restrictions on what constitutes a valid message. These issues are acknowledged by Odersky [22] who recommends not to mix the two paradigms and send only immutable messages.

The enormous interest in messages-passing concurrency is well documented by the growing amount of actor-based frameworks for the JVM. First attempts date back to 1998 when M. Astley started working on ActorFoundry [2] - a simple model where actors are mapped directly to thread and both deep-copy and 'by reference' messaging is supported. More recent efforts include Killim [26] which provides continuation based actors similar to the event-based actors in Scala. However, the Kilim framework uses a special post-processor to weave an additional 'fibre' parameter to the actor execution flow. The fibre collects information about stack frames, effectively building up a continuation. This enables the execution stack to unwind naturally as opposed to an abrupt termination by throwing an exception (Scala approach). A natural stack return is noted to be faster by almost two orders of magnitude than unwinding the stack via exceptions [26]. Jetlang[4] and Functional Java[5] are amongst other notable implementations of message-passing concurrency.

[4]http://code.google.com/p/jetlang/
[5]http://functionaljava.org/

### 2.1.4 Dataflow Concurrency

This paradigm provides a declarative, data-driven and fully deterministic concurrency model [3] whose theoretical foundation lies in KPN [12]. Dataflow concurrency supports only three operations; *create* to define dataflow variables, *wait* to read the variable value when it becomes available and *bind* to assign a value to a variable (this can happen only once per variable's lifetime). Independent processes are treated as 'black boxes' which execute as long as their required input in the form of dataflow variables is available (read operations). If an unbound variable is encountered the process stops and waits until the value becomes available. Processes can also provide output by binding dataflow variables.

Dataflow concurrency is referred to as the dead paradigm [3]. Hence, there are relatively few implementations of this concurrency model available on the JVM. One of the early efforts was the Flow Java framework [5] which extended Java with single assignment variables and futures to support dataflow concurrency. More recent projects include the Akka framework[6] and Ozma [4]. Both of these project are inspired by the Oz programming language[7].

## 2.2 Concurrent Data Structures

The traditional approach to implementing concurrent algorithms uses mutual exclusion (locks) to isolate critical sections [21]. However, locking comes at a price. Coarse-grained locking reduces performance while fine-grained locking increases complexity and the likelihood of deadlocks.

Considering the availability of the atomic *read-modify-check-write* operation as a hardware primitive, non-blocking, lock-free algorithms provide an attractive alternative as they typically outperform their locking counterparts [20].

This part of the research focuses on non-blocking, lock-free algorithms that can implemented in Java using atomic variables available in the *java.util.concurrent.atomic* package.

### 2.2.1 Lock-free Algorithms

Treiber [29] introduces a lock-free algorithm for LIFO concurrent singly linked list which has become commonly known as the Treiber stack. The list is represented by a sequence of nodes; each holding its value and a pointer to the next node. The Treiber stack uses CAS operations to modify the top node in the sequence (called *head*). Note that in Java the *head* node can be maintained as an instance JUC's *AtomicReference* which provides the atomic CAS operation.

While very simple the Treiber stack presents a bottleneck under high loads as the number of failed CAS operations on the *head* increases dramatically. Hendler et al. [9] introduce an algorithm for a lock-free stack which overcomes the scalability issue and performs well under both low and high loads. The algorithm is based on a simple observation that performing a *push* followed by a *pop* does not change the internal state of stack. Therefore, these two operations can be eliminated and the value exchanged without modifying the stack's structure. Hendler et al. [9] proposes the use of an elimination array in which threads that have previously lost CAS races on the stack can collide and exchange the value directly.

[6]http://doc.akka.io/docs/akka/snapshot/scala/dataflow.html
[7]http://www.mozart-oz.org/

Lock-free algorithms are not limited to LIFO stacks. Much work has been done on other data structures such as queues, sets and hash tables [19][18].

# Chapter 3

# System Analysis and Design Decisions

## 3.1 Core Concepts

### 3.1.1 Combinator Definition

One of the key concepts of the Wire Calculus [25] declares that well-formed semantics of syntactic expressions can be depicted as "black boxes" with a boundary on either side. Each expression is described by an LTS with an associated sort `(k,l)` which intuitively prescribes the number of wires along the left and right boundary respectively.

In object-oriented design, the black box can be represented as a class with no public methods. The only way to communicate with this class is through two sets of ports on the left and right boundaries. These ports effectively define the public interface of the black box. A black box with two boundaries shall be referred to as a combinator.

### 3.1.2 Boundaries and Ports

A boundary can be thought of as an ordered collection of ports that provide a means of communication between combinators. It is important to emphasise that combinators can only communicate over complementary boundaries and strictly on one-to-one basis (see Figure 3.1). This notion is distinct from public methods which can be invoked from any external object holding the corresponding reference.

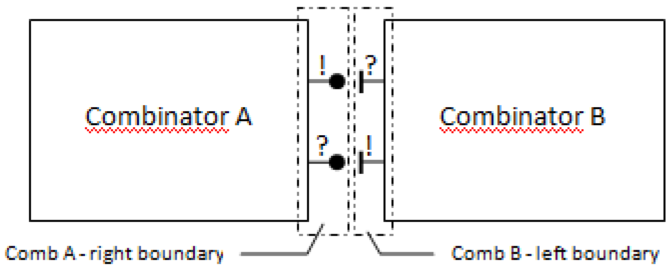


Figure 3.1: Two combinators communicating via complementary boundaries

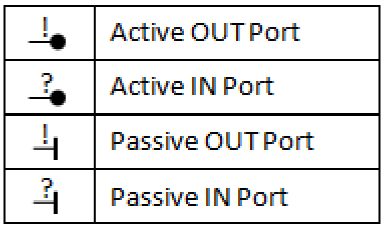
Ports are defined by three essential properties:

- Port Control Type
- Port Data Flow
- Port Data Type

The first property, Control Type, recognizes two types of ports; Active and Passive. Connection can only be established between exactly one active and one passive port. The active port dynamically initiates communication while the passive port waits to be invoked. In the graphical representations active ports are depicted as a line ending with a filled circle while passive ports end with a horizontal line (see Table 3.2).

The Data Flow property specifies whether data is sent (Data Flow OUT) or received (Data Flow IN) through the port. Graphically, Data Flow types are represented by annotations over ports. The exclamation mark (!) denotes an OUT port while the question mark implies an IN port (see Table 3.2). For two ports to be compatible, one must send data (OUT) while the other must receive data (IN).

The final property, Data Type, specifies the type of data that is allowed to pass through the port. To ensure type safety between connected ports, the data type of the sending (OUT) port must be assignable from the data type of the receiving (IN) port.



| | |
|---|---|
| !● | Active OUT Port |
| ?● | Active IN Port |
| !┤ | Passive OUT Port |
| ?┤ | Passive IN Port |

Figure 3.2: Graphical Representations of Four Distinct Ports Types

Table 3.2 shows the four different types of ports which can be created by combining the Control Type and Data Flow properties. These, together with the Data Type, define the basic rules for combinator composition. Two ports are connectable only if they have opposing Control Type, opposing Data Flow and compatible Data Types. These rules propagate to boundaries where two boundaries are said to be complementary if they contain an identical number of ports and all ports at the corresponding indices are connectable.

### 3.1.3 Composition

The Wire Calculus defines two coordination operators; vertical composition along a common boundary `(cut)` and a non-communicating parallel composition `(ten)` [25]. The behaviour of these operators is defined with LTS using the following structural operational semantics (SOS):

$$\frac{P \xrightarrow[c]{a} Q \quad R \xrightarrow[b]{c} S}{P;R \xrightarrow[b]{a} Q;S}\,\texttt{(cut)} \qquad \frac{P \xrightarrow[b]{a} Q \quad R \xrightarrow[d]{c} S}{P \bigotimes R \xrightarrow[bd]{ac} Q \bigotimes S}\,\texttt{(ten)}$$

In the JCC framework, these two coordination operators shall be defined in terms of vertical and horizontal compositions of two combinators as shown in Figure 3.3.

Both the vertical and the horizontal composition operations result in the creation of a new combinator which encapsulates the semantics of the two original combinators. The boundaries exposed by the new combinator depend on the type of composition. The left boundary of a new combinator obtained through horizontal composition `(cut)` mirrors the left boundary of the left operand. The right boundary of the right operand becomes the right boundary of the new combinator. The inner boundaries of the two operands are connected provided they are complementary. As for vertical compositions, no boundaries are connected. The boundaries of the new combinators are simply defined as aggregates of the corresponding boundaries of the operands (see Figure 3.3).

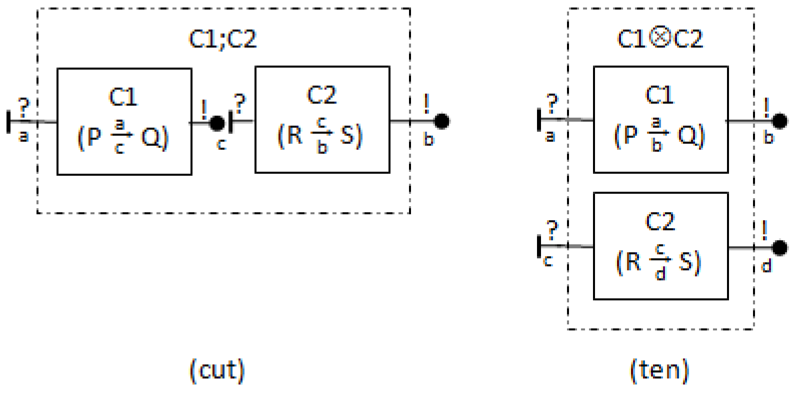


Figure 3.3: Graphical Representation of the `(cut)` and `(ten)` SOS rules for combinators

Constructing combinators in such a formal way provides a sound foundation for reasoning about their behaviour and identifying bisimilarities. This will help with optimisation of complex data structures in the future.

### 3.1.4 Communication

In the Wire Calculus, connected components globally synchronise on a signal passing through their boundaries. The JCC framework emulates this concept by sending messages (or requests for such messages) through the connected ports of combinators. Typically, a message is produced by a single combinator known as the Producer and has one or many intended Consumers (other combinators). When the Producer sends out a message all the intended Consumers need to synchronise on that message. If one fails to do so then all of them must fail. This concept is similar to the 'all-or-nothing' proposition used in database transactions and defines clear rules for combinator synchronisation over messages.

With the notion of failures in place, the JCC framework also provides facilities for dealing with such failures. When a combinator sends a message there are generally two types of failure that can occur: *transient* or *permanent*. Transient failures are typically caused by interference from other synchronisation signals (e.g. a lost race to CAS shared memory location). In such a case the combinator should back off and retry later. In contrast, permanent failures result in an invalidation of the message and cannot be therefore handled locally. The only reasonable course of action is to notify the original message producer of the failure and drop the message. Causes of permanent failures can be manifold but the Join Wire[1] provides a good example. An attempt to join unequal messages results in a permanent failure and invalidation of the messages.

### 3.1.5 Messages and Failure Propagation

The 'all-or-nothing' requirement for synchronisation over messages together with the possibility of failures requires a sophisticated system for failure handling. The approach taken in the JCC framework is based on the concept of uniqueness of all messages propagating through a composition of combinators at any time. As shown in Figure 3.4, the Copy Wire[2] sends a new, unique message to every outgoing port. However, each of the new messages encapsulates the original one; effectively creating a shared state amongst all of them. Similarly, the Join Wire outputs a new message encapsulating all incoming messages.

[1]The Join Wire is a concrete combinator that receives multiple messages through its IN ports, attempts to merge them and synchronises over the resulting join message on its right boundary.

[2]The Copy Wire is a concrete combinator that provides atomic synchronisation of multiple system branches by sending copies of an incoming message through its active OUT ports.

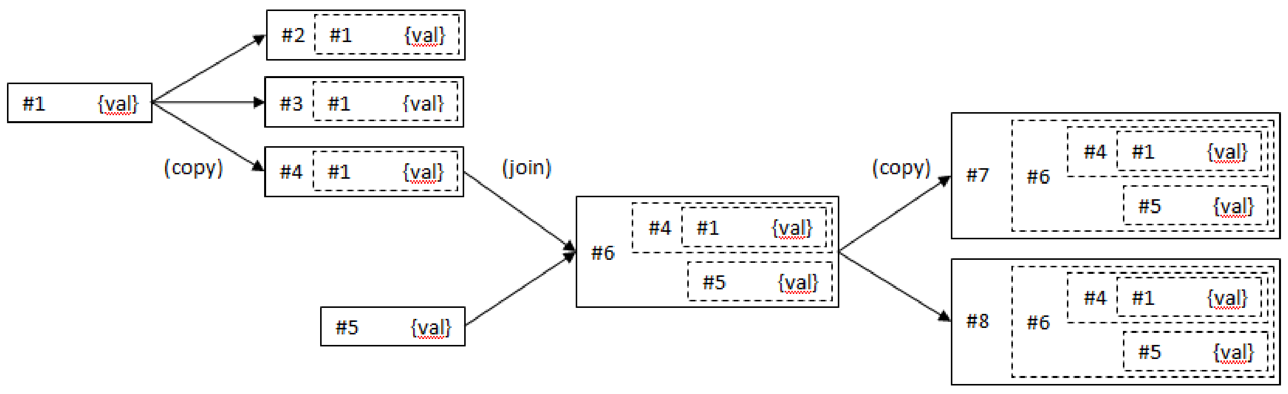


Figure 3.4: Visualisation of messages travelling through Copy and Join Wires

The creation of shared state amongst related messages provides a very powerful mechanism for determining the current message status.

**Definition 1** (Validity). For a (top-level) message to be valid all encapsulated messages must be valid.

**Definition 2** (Invalidation). If a message is invalidated all associated messages must become invalid as well. For example, referring to the diagram in Figure 3.4, if message #8 is invalidated then all (directly or indirectly) encapsulated messages (#6,#5,#4 and #1) also become invalid. Notice that message #7 now encapsulates invalid messages and therefore, in accordance with Definition 1, is no longer valid either. The same statement is true of messages #2 and #3 as they encapsulate now invalid #1.

**Definition 3** (Full Acknowledgement). When a Consumer receives a message it must (locally) acknowledge its receipt. However, the content is not available for processing until the message becomes fully acknowledged. For a top-level message to be fully acknowledged all encapsulated messages must be also fully acknowledged. For an encapsulated message to be fully acknowledged all its (direct or indirect) top-level wrappers must be acknowledged at least locally.

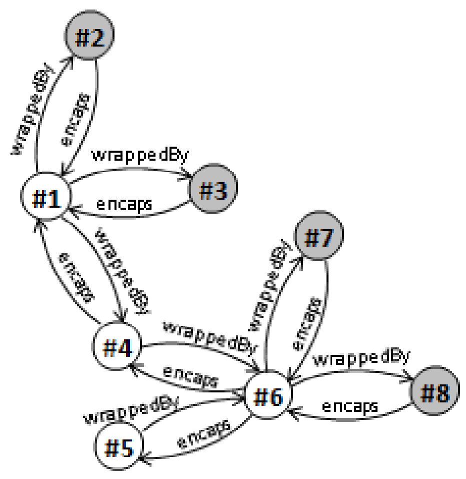


Figure 3.5: Relationships between messages in Figure 3.4 represented as a graph

The diagram in Figure 3.4 can be translated into a directed graph shown in Figure 3.5 where nodes (messages) are connected through two properties; *wrappedBy* and *encaps*. The graph shows that the *wrappedBy* is an inverse of property *encaps* and both are transitive. This has implications for implementation of the system as it shows that a message needs to only know about its immediate neighbours.

## 3.2 Semantics of Wires

### 3.2.1 Identity Wires

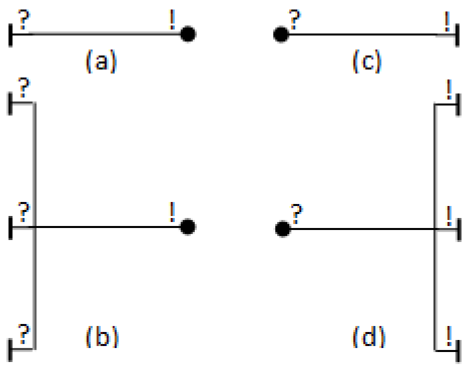


Figure 3.6: Graphical representation of the Push Adaptor Wire (a,b) and the Pull Adaptor Wire (c,d)

Identity wires receive a synchronisation request on one of its passive ports and propagate it through the active port on the other boundary. There are two types of identity wires in JCC as shown in Figure 3.6. The Push Adaptor Wire receives a message though a passive port and pushes it on via the other boundary. Conversely, the Pull Adaptor Wire receives a pull request on a passive port and fetches a message through its active port on the other boundary. Figure 3.6 shows both types of identity wires with one as well many passive ports. This is to emphasise how these wires can be used in a system. For example, if multiple producers need to push messages on a single stack the Push Adaptor Wire provides the required connector.

### 3.2.2 Permute Wire

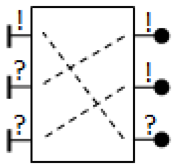


Figure 3.7: Graphical representation of the Permute Wire

The semantics of the Permute Wire are identical to those of the identity wires. However, there is an equal number of ports on both boundaries and each port on the left is internally connected to complementary port on the right. The main benefit of the Permute Wire is that it allows for reordering of connected ports from one boundary to the other.

### 3.2.3 Reverse Wire

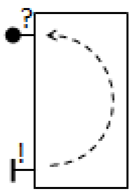


Figure 3.8: Graphical representation of the Reverse Wire

Similarly to the Permute Wire the passive ports of the Reverse Wire are internally connected to the corresponding active ones. The main difference is the Reverse Wire has all its ports on a single boundary and has therefore the ability to change the direction in which synchronisation requests are travelling. Figure 3.8 shows a simple Reverse Wire with only two ports but there could be more. For example, if a combinator has three ports then a Reverse Wire with six ports can be constructed. The top three are complements of the combinator boundary while the bottom three are identical to the combinator's boundary. Internally, port 1 is connected to port 4, 2 to 5 and 3 to 6.

### 3.2.4 Synchronisation Wire

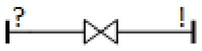


Figure 3.9: Graphical representation of the Synchronisation Wire

The Synchronisation Wire introduces a rendezvous style of synchronisation over a message. It receives a message through one passive port and a pull request through the other. These signals must wait for each other in the Synchronisation Wire and are released when the message is successfully exchanged.

### 3.2.5 Copy Wire

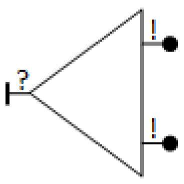


Figure 3.10: Graphical representation of the Copy Wire

The Copy Wire enables atomic synchronisation of multiple system branches over a single message. The semantics dictate that a message coming in through the passive port is propagated to all active ports on the opposite boundary. The independent outgoing signals are all required to complete (succeed/permanently fail) before another synchronisation request is allowed to enter the Copy Wire.

### 3.2.6 Join Wires

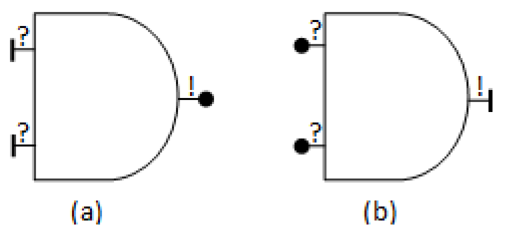


Figure 3.11: Graphical representation of the Join Push Wire (a) and the Join Pull Wire (b)

The main purpose of the Join Wire is to merge multiple incoming synchronisation signals into one. In the JCC framework this equates to a number of messages being 'wrapped' into a single message and propagated through the opposite boundary. However, this operation is allowed only if all the incoming messages are meaningfully equal. An attempt to join unequal messages results in a permanent failure of all incoming signals. Figure 3.11 shows that there are two types of Join Wires; Push and Pull. While the Join Push Wire (a) passively waits for all messages to arrive, joins them and actively pushes the result through the active port on the right boundary, the Join Pull Wire (b) actively propagates a pull request coming from the right through the active ports on the left and subsequently attempts to join the messages it receives in reply.

### 3.2.7 Choice Wires

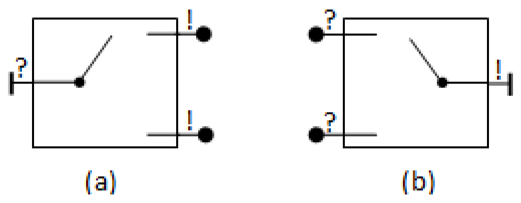


Figure 3.12: Graphical representation of the Choice Push Wire (a) and the Choice Pull Wire (b)

The Choice Wire is inspired by the non-deterministic choice operator introduced in the Wire Calculus. However, the JCC framework implementation of choice is biased rather than non-deterministic. For example, upon the receipt of a message through the passive port on the left the Choice Push Wire (see Figure 3.12.a) attempts to propagate the signal via the first (top-most) active port on the right. If the first port fails transiently then the second port is attempted and so on. These synchronisation attempts continue in a loop until synchronisation via one of the port succeeds or fails permanently. The Choice Pull Wire shown in Figure 3.11.b provides the same functionality for pull requests.

# Chapter 4

# Development of the JCC Framework

## 4.1 Development Environment

### 4.1.1 Project Settings

The project uses Apache Maven[1] to automate the build process. While Maven is typically associated with management of external references (which there are not any in this case) it also provides a project structure recognised by many IDEs. This helps greatly with portability of the project.

The Maven Compiler Plugin is configured to target Java 1.7. However, features introduced in Java 1.7 are not used in the source code. Therefore, it is possible to compile the framework for Java 1.6 as well.

### 4.1.2 Version Control and Source Code Management

The project uses Git[2] for version and revision control of the source code. The repository with a number of branches can found at `https://github.com/cifa/MScProject`.

## 4.2 Core Classes

### 4.2.1 Port

As described in the System Analysis, the Port class is defined in terms of the Control Type, Data Flow and Data Type properties. The Control Type and Data Flow are implemented as simple enumerations. As for the Data Type property, Port is defined as a generic class over type T which corresponds to the data type of messages allowed to travel through the port. However, the use of type erasure in Java generics makes the implementation more complex as generic types are not available at runtime. To solve the type erasure issue the JCC framework combines generics with reflection as shown in the following code snippet.

```
public class Port<T> {
    private final Class<T> portDataType;
    ...
    private Port(Class<T> portDataType, ...) {
        if(portDataType == null) {
            throw new IllegalArgumentException("Port Data Type cannot be null");
        }
        this.portDataType = portDataType;
        ...
    }
```

[1] `http://maven.apache.org/`

[2] `http://git-scm.com/`

```
    ...
}
```

By requiring a reference to *Class<T>* in the constructor and storing it in an instance variable, it is possible to refer to the generic type T at runtime using methods of the *Class<?>* class such as *isAssignableFrom(Class<?> cls)*. This solution enables JCC to ensure type safety when connecting ports and passing messages at runtime.

An astute reader will have noticed that the constructor in the code snippet above is declared *private*. In fact, rather than public constructors, the Port class provides three static factory methods for creating valid active and passive ports as shown below.

```
public static <T> Port<T> getActivePort(Class<T> portDataType, DataFlow portDataFlow) {...}

public static <T> Port<T> getPassiveInPort(Class<T> portDataType, PassiveInPortHandler<T>
    handler) {...}

public static <T> Port<T> getPassiveOutPort(Class<T> portDataType, PassiveOutPortHandler<T>
    handler) {...}
```

As the name suggests the *getActivePort* method creates a new port of *ControlType.ACTIVE* with the other two properties supplied as parameters. Internally, active ports provide methods for sending and receiving messages. The second method, *getPassiveInPort*, creates a port of *ControlType.PASSIVE* and *DataFlow.IN*. Similarly, the *getPassiveOutPort* creates a passive port of *DataFlow.OUT*. Passive ports introduce the notion of port handlers; code that is executed when a passive port is invoked by its active counterpart. These handlers come in two flavours as shown in the code snippet below.

```
PassiveOutPortHandler<T>() {
    @Override
    public Message<? extends T> produce() {...}
};

PassiveInPortHandler<T>() {
    @Override
    public void accept(Message<? extends T> msg) {...}
};
```

Both *PassiveOutPortHandler* and *PassiveInPortHandler* are abstract classes whose concrete implementations must override the *produce* and *accept* methods respectively. This is a very important concept for combinators as their behaviour is typically defined in implementations of the aforementioned methods.

The Port class also provides methods for sending and receiving messages as well as a method for connecting two ports. The *connectPorts* methods carries out the compatibility checks highlighted in the System Analysis before initializing an instance variable of the active port to hold reference to the connected passive port. This enables the active port to call the passive port's handler when communication is initiated.

### 4.2.2 Boundary

The Boundary class is implemented as a simple wrapper over a *List* of ports. All methods in the class use the default access modifier which limits their visibility to the core package only. As the Boundary class provides no additional functionality it might be refactored away in the future versions of JCC with lists of ports denoting left and right boundaries associated directly with the *Combinator* class.

### 4.2.3 Combinator

*Combinator* is implemented as an abstract class encapsulating the logic for sending and receiving messages via its boundaries as well as providing functionality for combining combinators both horizontally and vertically.

There are two abstract methods that concrete combinators must implement, *initLeftBoundary* and *initRightBoundary*, both of which return a list of ports (*List<Port<?>>*). It is important to note that these methods are not called from the *Combinator* constructor during initialisation. Rather, the boundaries are initialised lazily when required for the composition process. This means that any instance variables of the concrete combinators can be safely used in port construction.

Both vertical and horizontal composition is achieved by invoking the *combine* method of the *Combinator* class:

```
public Combinator combine(final Combinator other, CombinationType combinationType) throws
    IllegalCombinationException {...}
```

The *combine* method takes two parameters; the right operand for the composition operation and the type of composition required. *CombinationType* is implemented as a simple enumeration (HORIZONTAL, VERTICAL). As shown in the method signature above *IllegalCombinationException* is thrown if the composition fails. This can happen for two reasons. Firstly, horizontal composition fails if the inner boundaries of the composed combinators are not complementary or, secondly, one or both of the combinators has already been involved in another composition.

Typically, messages flow through combinators from left to right. For example, the Join Wire receives a message on the left and sends out the copies on the right. While this feels natural it is likely that the flow direction will be reversed in some scenarios. The following enumeration is introduced to cater for this possibility:

```
public enum CombinatorOrientation {
    LEFT_TO_RIGHT, RIGHT_TO_LEFT;
}
```

The *Combinator* class defines two constructors. The no-argument constructor initialises the combinator with the default *LEFT_TO_RIGHT* orientation. The other constructor takes a specific *CombinatorOrientation* value as its parameter. While this greatly improves the flexibility of the JCC framework it does not place any additional burden on developers as combinators only need to be implemented for the default *LEFT_TO_RIGHT* orientation. If they are subsequently initialised with the opposite orientation the *Combinator* superclass switches the boundaries while automatically maintaining the internal behaviour.

### 4.2.4 Message

The Message class implements the *java.util.concurrent.Future* interface which underpins many of the requirements identified in the System Analysis. While messages are not typical asynchronous computations, retrievability of their content is dependent on other asynchronous tasks; namely acknowledgement of all associated messages. That means that the *get()* and *get(long timeout, TimeUnit unit)* methods can block and even throw an exception if the message is later invalidated. Message invalidation corresponds to the *cancel()* method of the *Future* interface. The *isCancelled()* method returns true if the message has been invalidated while the *isDone()* returns true if the message has been invalidated or fully acknowledged. Messages maintain their current status in an instance variable of type *java.util.concurrent.atomic.AtomicInteger*. The status can be *ACTIVE* (0), *CANCELLED* (1) or *FULLY_ACKNOWLEDGED* (2). Each message starts off as active and changes its status exactly once during its lifetime to either cancelled or fully acknowledged. As specified in the System Analysis, complex messages must know about messages they directly encapsulate or are directly encapsulated by. These are held as collections in instance variables and used for recursive propagation of the status during invalidation and acknowledgement processes.

The concept of message invalidation and full acknowledgement provides an interesting challenge as message consumers often need to retrieve the message content in order to decide whether the message is valid. However, the content should remain inaccessible until the message is fully acknowledged (by which time it cannot be invalidated any more). Obviously, this is a Catch-22 situation which cannot be resolved without a compromise. JCC takes a pragmatic approach to this issue by providing the following static method in the Message class:

```
public static <T> boolean validateMessageContent(final MessageValidator<T> validator, final
    Message<? extends T>... msgs)
```

The *validator* parameter must be an implementation of the interface shown below:

```
public interface MessageValidator<T> {
    boolean validate(final T... contents);
}
```

To use this method in a realistic scenario let us say that a combinator receives messages A and B (of type Double) with temperature measurements. These are considered valid only if they are less than two degrees apart. The following code snippet shows a possible implementation of such a condition:

```
boolean valid = Message.validateMessageContent(
        new MessageValidator<Double> {
            boolean validate(final Double... contents) {
                return Math.abs(contents[0] - contents[1]) < 2;
            }
        }, msgA, msgB);
```

The validator defines a predicate under which the messages are considered valid and can be locally acknowledged. If the predicate fails both msgA and msgB are automatically invalidated. The *contents* parameter of the *validate* method is initialised with the content of the messages; in this case with the actual temperature values carried by msgA and msgB. The overall result of the *validateMessageContent* method call corresponds to result returned by the validator. While this approach works there are certain shortcomings. The assumption is that the values of messages available in the *validate* method remain read-only and cannot be passed outside the validator. However, these restrictions are defined only as a convention since they cannot be directly enforced in Java

Another important feature of the Message class is the notion of callbacks. As previously mentioned, every message becomes either invalid or fully acknowledged during its lifetime and it is conceivable that the producer of the message might wish to be notified of this status change. The JCC framework defines the following interface whose implementation may be (optionally) passed to the Message constructor:

```
public interface MessageEventHandler<T> {

    void messageInvalidated(Message<T> message, T content);
    void messageFullyAcknowledged(Message<T> message);
}
```

If provided, the callback reference is held in the message and the corresponding method is invoked when the message status change occurs. This provides additional flexibility as message producers can maintain a greater level of control over the messages they send. For example, it is possible to implement a custom failure handling system or even timeout (cancel) messages if they are not acknowledged/invalidated within a certain time limit.

### 4.2.5 Combinator Thread Pool

The *CombinatorThreadPool* class encapsulates an instance *ExecutorService* obtained through the *Executors.newCachedThreadPool()* factory method. This pool creates threads as needed, but reuses previously constructed ones if available. Introducing the thread pool for combinators such as the Copy Wire, which execute multiple asynchronous tasks, dramatically improves their performance.

### 4.2.6 Backoff Component

This class implements a lightweight backoff scheme which, depending on the number of available processors, uses either busy-wait (multi-core) or thread suspension (single-core) to wait for a random period of time. The backoff period increases exponentially every time the *backoff()* method is called.

The Backoff component is used extensively by combinator wires to handle transient failures such as *CombinatorFailedCASException* which can be typically resolved by backing off and retrying later.

## 4.3 Implementation of Wires

### 4.3.1 Push and Pull Adaptor Wires

As all wires, the *PushAdaptorWire* combinator implements the two abstract methods for boundary initialisation. The implementation of the right boundary with one active port is straightforward:

```
@Override
protected List<Port<?>> initRightBoundary() {
    List<Port<?>> ports = new ArrayList<Port<?>>();
    ports.add(Port.getActivePort(adaptorDataType, DataFlow.OUT));
    return ports;
}
```

The behaviour of the wire is defined in the implementation of *PassiveInPortHandler* which is common for all passive ports on the left. The handler uses the Backoff component in its retry loop (if required) while trying to propagate the message through the active port on the right:

```
@Override
protected List<Port<?>> initLeftBoundary() {
    PassiveInPortHandler<T> inHandler = new PassiveInPortHandler<T>() {
        @Override
        public void accept(Message<? extends T> msg) {
            Backoff backoff = null;
            while(true) {
                try {
                    sendRight(msg, 0);
                    // success -> return
                    break;
                } catch (CombinatorTransientFailureException ex) {
                    if (retryOnTransientFailure) {
                        // back off and retry
                        if(backoff == null) {
                            backoff = new Backoff();
                        }
                        try {
                            backoff.backoff();
                        } catch (InterruptedException e) {
                            // assoc msg must have been invalidated - clear flag
                            Thread.interrupted();
                        }
                    } else {
                        // allowed to fail transiently -> re-throw
                        throw ex;
                    }
                }
            }
        }
    };
    List<Port<?>> ports = new ArrayList<Port<?>>();
    for(int i=0; i<noOfPushPorts; i++) {
        ports.add(Port.getPassiveInPort(adaptorDataType, inHandler));
    }
    return ports;
}
```

Properties such as *noOfPushPorts* and *retryOnTransientFailure* are specified through the class constructor.

The implementation of *PushAdaptorWire* follows the same principles.

### 4.3.2 Permute and Reverse Wires

Both the Permute and Reverse Wires extend the *AbstractUntypedWire* combinator which provides generic methods for building interconnected ports:

```
protected <T> Port<T> getPort(PortDefinition<T> def, Side side, int connectedPortIndex) {...}
protected <T> Port<T> getComplementaryPort(PortDefinition<T> def, Side side, int
    connectedPortIndex) {...}

protected enum Side {LEFT, RIGHT}
```

As shown in the code snippet above, the core port properties are specified via the *PortDefinition* class. Predictably, the *getComplementaryPort* methods creates a port that is connectable to the one specified by the definition. When the resulting port is passive the remaining two arguments are used to create the handler. Passive ports handlers are implemented as inner classes in *AbstractUntypedWire*. The following code highlights the concept:

```
private class IdentityPassiveOutPortHandler<T> extends PassiveOutPortHandler<T> {

    private final int portIndex;
    private final Side side;

    IdentityPassiveOutPortHandler(int portIndex, Side side) {
        this.portIndex = portIndex;
        this.side = side;
    }

    @Override
    public Message<? extends T> produce() throws CombinatorFailureException  {
        if(side == Side.LEFT) {
            return (Message<? extends T>) receiveLeft(portIndex);
        } else {
            return (Message<? extends T>) receiveRight(portIndex);
        }
    }
};
```

The implementation of the *produce* and *accept* (not shown) is straightforward as the main purpose of both the Permute and Reverse Wire is to simply forward synchronisation requests.

When constructed the Permute Wire requires arrays of *PortDefinitions* and *Integers* of an equal length. The ports on the left boundary are obtained via the *getPort* method in the order defined by the *PortDefinitions* array and connecting to a port on the right identified by the values of the corresponding element of the array of *Integers*. The ports on the left are constructed in a similar way using the *getComplementaryPort* method.

The Reverse Wire is even simpler as it only requires an array of *PortDefinitions*. Ports are initialised on the left boundary by iterating twice over the *PortDefinitions* array as shown below:

```
@Override
protected List<Port<?>> initLeftBoundary() {
    List<Port<?>> ports = new ArrayList<Port<?>>(portsDefinitions.length * 2);
    // add the top ports first
    for(int i=0; i<portsDefinitions.length; i++) {
        ports.add(getPort(portsDefinitions[i], Side.LEFT, portsDefinitions.length + i));
    }
    // ... and then add the complementary ones
    for(int i=0; i<portsDefinitions.length; i++) {
        ports.add(getComplementaryPort(portsDefinitions[i], Side.LEFT, i));
    }
    return ports;
}
```

Evidently, the first *portsDefinitions.length - 1* ports match the interface specified in the array with the remaining ports creating a complementary interface below. The internal connections of the ports follow the
*portIndex <–>portIndex + portsDefinitions.length* pattern where *portIndex < portsDefinitions.length*.

### 4.3.3 Synchronisation Wire

The *SynchWire* has one passive port on each boundary. Access to each of these ports is synchronised over an instance of the JUC's Reentrant Lock. This ensures that only two threads can simultaneously enter the *SynchWire*; one carrying a message on the left and the other requesting a message on the right. The exchange is carried out via an instance of the JUC's *Exchanger* which blocks the first thread until the other one arrives. A maximum waiting time can also be specified. In such a case the *SynchWire* fails transiently if the exchange cannot be carried out within the given time limit.

### 4.3.4 Copy Wire

The *CopyWire* has a single passive IN port on the left while the number of active OUT ports on the right is specified through the class constructor. The implementation of the left port handler is shown in the following code snippet:

```java
@Override
protected List<Port<?>> initLeftBoundary() {
    List<Port<?>> ports = new ArrayList<Port<?>>();
    ports.add(Port.getPassiveInPort(dataType, new PassiveInPortHandler<T>() {
        @Override
        public void accept(final Message<? extends T> msg) throws CombinatorFailureException {
            mutexIn.lock();
            try {
                permanentFailure = false;
                transientFailure = false;
                final CountDownLatch copyStart = new CountDownLatch(1);
                final CountDownLatch copyComplete = new CountDownLatch(noOfCopyPorts);
                // start all copy runners (threads)
                for(int i=0; i<noOfCopyPorts; i++) {
                    // create copy message by encapsulating the original one
                    Message<T> copyMsg = (Message<T>) new Message<>(msg);
                    CombinatorThreadPool.execute(new CopyRunner(copyMsg, i, copyStart,
                        copyComplete));
                }
                copyStart.countDown(); // runners must execute when all wrappers initialised
                copyComplete.await();  // wait for all runners to complete
                // check for problems
                if(permanentFailure) throw PERMANENT_EXCEPTION;
                if(transientFailure) throw TRANSIENT_EXCEPTION;
            } catch (InterruptedException e) {
                // this shouldn't really happen
                throw msg.isCancelled() ? PERMANENT_EXCEPTION : TRANSIENT_EXCEPTION;
            } finally {
                mutexIn.unlock();
            }
        }
    }));
    return ports;
}
```

Instances of the JUC's *ReentrantLock* and *CountDownLatch* are used to ensure the synchronous nature of the Copy Wire. Each outgoing signal is executed as an independent task represented by an instance of *CopyRunner*; an inner class show below:

```java
private class CopyRunner implements Runnable {
    private final CountDownLatch copyStart, copyComplete;
    private final int portIndex;
    private final Message<T> msg;

    public CopyRunner(Message<T> msg, int portIndex, CountDownLatch copyStart, CountDownLatch
        copyComplete) {
        ... init variables ...
    }

    @Override
    public void run() {
        try {
                copyStart.await();
                Backoff backoff = null;
                while(true) {
                    try {
                        sendRight(msg, portIndex);
                        break;  // success
                    } catch(CombinatorTransientFailureException ex) {
                        if(retryOnTransientFailure) {
                            // back off and retry
                            if(backoff == null) backoff = new Backoff();
                            backoff.backoff();
                        } else {
                            throw ex;    // re-throw the transient exception
                        }
                    }
                }
            } catch (CombinatorPermanentFailureException | InterruptedException e) {
                permanentFailure = true;     // permanent failure
            } catch (CombinatorTransientFailureException ex) {
                transientFailure = true;     // only possible with retry off
            }
            copyComplete.countDown();
        }
    }
```

### 4.3.5 Join Push Wire

The Join Push Wire use the JUC's *CyclicBarrier* to synchronise threads coming through the passive ports on the left. Each of these ports is associated with a *ReentrantLock* to ensure only one thread is allowed to enter at any time. Before it calls the *await()* method on the common barrier each incoming thread places its message into an allocated slot in a *AtomicReferenceArray<Message<T>>*. The *CyclicBarrier*'s support for executing a *Runnable* command once per barrier point, after the last thread arrives, but before the threads are released, is utilised to carry out the join procedure. In fact, the *JoinPullWire* class implements the *Runnable* interface and the logic for joining messages resides in its *run()* method. If the join procedure is successful the resulting join message is sent out as an independent task using the *CombinatorThreadPool.execute(new Runnable{...})* method. If the join operation fails all the original messages are invalidated and the threads released from the barrier throw *CombinatorPermanentFailureException*.

Note that part of the message invalidation process is invoking the *interrupt()* method on its carrying thread. If this happens while the thread is waiting at the barrier the *InterruptedException* is thrown and the barrier is broken. All the other waiting threads (if any) are also released via *BrokenBarrierException*. The Join Pull Wire handles these exceptions locally by resetting the barrier and determining the type of failure for each of the released threads. The threads whose messages are invalid throw *CombinatorPermanentFailureException* while the remaining ones fail transiently with *CombinatorTransientFailureException*.

### 4.3.6 Join Pull Wire

The Join Pull Wire is essentially opposite of the Copy Wire and, as such, it uses analogous techniques to provide the required functionality. The passive port on the right is synchronised over a Reentrant Lock and independent tasks are spawned to propagate the message pull request through the passive ports on the left boundary. The implementation of the inner class used to execute the pull requests is shown in the following excerpt:

```
private class JoinRunner implements Runnable {
    private final CountDownLatch runnersComplete;
    private final int portIndex;

    public JoinRunner(int portIndex, CountDownLatch complete) {
        this.portIndex = portIndex;
        this.runnersComplete = complete;
    }

    @Override
    public void run() {
        Backoff backoff = null;
        while(!permanentFailure) {
            try {
                Message<T> msg = (Message<T>) receiveLeft(portIndex);
                joinMessages.set(portIndex, msg);    // set msg for this port
                noOfPulledMsgs.incrementAndGet();
                // success -> return
                break;
            } catch (CombinatorTransientFailureException ex) {
                if(!retryOnTransientFailure || permanentFailure) break;
                // back off and retry
                if(backoff == null) {
                    backoff = new Backoff();
                }
                try {
                    backoff.backoff();
                } catch (InterruptedException e) {
                    // shouldn't happen -> just clear the interrupt flag
                    Thread.interrupted();
                }
            } catch (CombinatorPermanentFailureException ex) {
                permanentFailure = true;
            }
        }
        runnersComplete.countDown();      // runner done
    }
}
```

A *CountDownLatch* is used to synchronise the completion of all *JoinRunner*s while a *AtomicReferenceArray<Message<T>>* stores successfully fetched messages for the forthcoming join procedure. The Join Pull Wire also uses an instance variable of type *AtomicInteger* to monitor whether all *JoinRunner*s completed successfully. The join procedure is attempted only if the *noOfPulledMsgs* equals to the *noOfJoinPorts*. If some messages are unobtainable or the join procedure fails all fetched messages are invalidated and the initial request on the right boundary is released with *CombinatorPermanentFailureException*.

### 4.3.7 Choice Push and Pull Wires

The implementation of the Choice Push Wire is reasonably straightforward. Similarly to the Copy Wire, there is a passive IN port on the left boundary and an arbitrary number of active OUT ports on the right. When a message-carrying thread enters the wire via the left port it is forwarded out through the first (top) port on the right. If the port fails transiently the Choice Wire catches the exception and forwards the message on to the next port. This process is repeated in a loop until one of the ports succeeds or the message fails permanently. If the last (bottom) port on the right fails transiently the process is restarted from the first port.

The Choice Pull Wire follows the same process for pull requests as shown in the code snippet below:

```
@Override
protected List<Port<?>> initRightBoundary() {
    List<Port<?>> ports = new ArrayList<Port<?>>();
    ports.add(Port.getPassiveOutPort(dataType, new PassiveOutPortHandler<T>() {
        @Override
        public Message<? extends T> produce() throws CombinatorPermanentFailureException {
            int portIndex = 0;
            while(true) {
                try {
                    Message<? extends T> msg = (Message<? extends T>) receiveLeft(portIndex);
                    return msg;
                } catch(CombinatorTransientFailureException ex) {
                    if(++portIndex == noOfChoices) {
                        portIndex = 0;
                    }
                }
            }
        }
    }));
    return ports;
}
```

## 4.4 Implementation of Data Combinators

The data combinators developed as part of this project are implementations of well-known lock-free algorithms which are used to test performance of the JCC framework.

### 4.4.1 Treiber Stack

This *Combinator* provides an implementation of a lock-free stack introduced by Treiber [29]. As opposed to other implementations, which typically repeat CAS operations until they succeed, the *TreiberStack* combinator fails transiently with *CombinatorFailedCASException* if it loses a CAS race. Hence, CAS failures can be detected by enclosing combinators and handled flexibly. This flexibility a key ingredient for composition of more complex data structures such as the elimination stack introduced by Hendler et al. [9].

The *push* and *pop* operations are implemented as a passive IN port on the left and a passive OUT port on the right respectively.

### 4.4.2 Michael and Scott Queue

The *MSQueue* combinator implements a lock-free queue algorithm proposed by Michael and Scott [20]. Similarly to the *TreiberStack* combinator, the *enqueue* and *dequeue* operations fail transiently rather than retrying CAS attempts in a loop.

### 4.4.3 Elimination Exchanger

The *EliminationExchanger* combinator is intended for composition with the Treiber Stack to provide an implementation of the scalable lock-free stack algorithm presented by Hendler et al. [9]. The Elimination Exchanger maintains a dynamically resizeable array of JUC's *AtomicReference* slots enabling synchronisation of push and pop requests. Each request is represented by a node which carries the value of the request itself as well as a hole waiting to be filled during the exchange. First, a request selects a slot and checks its state. If it is empty the request node is CASed into the slot and awaits fulfilment. However, if the slot is occupied by a matching node (push matches with pop and vice versa) the executing request claims the matching node by CASing the slot back to null. Once claimed the hole of the matching node is filled with the value of the current request node and vice versa. This represents a successful exchange.

If any of the CASes fail or the slot is occupied by a non-matching request a failure count is incremented and the process repeated (possibly on a different slot). Too many failures cause the slot array to increase in size. Conversely, if a request node CASed into a slot is not claimed within a time limit the slot array size may be decreased.

The implementation of the *EliminationExchanger* combinator is based closely on the JUC's *Exchanger* class.

# Chapter 5

# Building Complex Concurrent Data Structures with JCC

## 5.1 Creating Components

A rich set of Wires and Data Combinators provided by JCC enables software developers to create concurrent data structures and encapsulate them into regular Java components. Figure 5.1 shows how a concurrent elimination stack component can be easily developed with the aid of the JCC framework. Note that a more detailed description of the elimination stack construct in JCC is provided in the next chapter.

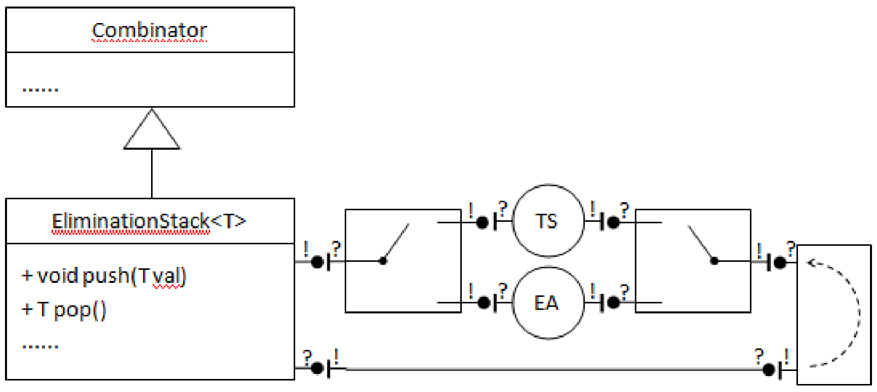


Figure 5.1: Diagram showing an implementation of the Elimination Stack algorithm using the JCC framework

The following excerpt from the elimination stack component shows how the JCC combinators are wired in the constructor of the class:

```
public EliminationStack() {
    ChoiceSendWire<Object> sendChoice = new ChoiceSendWire<>(Object.class, 2,
        combinatorOrientation);
    ChoiceReceiveWire<Object> receiveChoice = new ChoiceReceiveWire<>(Object.class, 2,
        combinatorOrientation);
    TreiberStack<Object> stack = new TreiberStack<>(Object.class, combinatorOrientation);
    EliminationExchanger<Object> eliminationExchanger = new EliminationExchanger<>(Object.
        class, combinatorOrientation);

    Combinator eliminationStack = sendChoice.combine(
        (stack.combine(eliminationExchanger,CombinationType.VERTICAL)),CombinationType.
            HORIZONTAL).combine(receiveChoice, CombinationType.HORIZONTAL);

    ReverseWire reverse = new ReverseWire(eliminationStack.
        getPortDefinitionCompatibleWithRightBoundary(),combinatorOrientation);
    AdaptorPullWire<Object> pull = new AdaptorPullWire<>(Object.class, 1,
        CombinatorOrientation.getOpposite(combinatorOrientation));

    this.combine((eliminationStack.combine(pull, CombinationType.VERTICAL)
        .combine(reverse, CombinationType.HORIZONTAL)), CombinationType.HORIZONTAL);
}
```

The *combinatorOrientation* is a protected final variable exposed by the *Combinator* superclass. In this case it refers to the default *LEFT_TO_RIGHT* orientation. Note how the *ReverseWire* obtains the compatible boundary definition from the *Combinator* resulting from the previous composition (*eliminationStack*). This is a powerful mechanism that greatly enhances the framework's flexibility. Another interesting point is the construction of the *AdaptorPullWire* with the opposite orientation to the rest of the structure. This essentially reverses the Data Flow property of the wire and swaps the boundaries around which is exactly what is required in this scenario. Yet again, this demonstrates the great flexibility of JCC.

With the combinator wiring in place implementation of the rest of the component is pleasantly simple:

```
@Override
protected List<Port<?>> initLeftBoundary() {
    return Collections.emptyList();
}

@Override
protected List<Port<?>> initRightBoundary() {
    List<Port<?>> ports = new ArrayList<Port<?>>();
    ports.add(Port.getActivePort(Object.class, DataFlow.OUT));
    ports.add(Port.getActivePort(Object.class, DataFlow.IN));
    return ports;
}

@Override
public void push(T value) {
    sendRight(new Message<Object>(Object.class, value), 0);
}

@Override
public T pop() {
    return (T) receiveRight(1).get();
}
```

The component exposes no ports on the left boundary and uses two active ports on the right to connect with the rest of the composition as shown in Figure 5.1. Subsequently, the implementation of *push* and *pop* becomes trivial.

# Chapter 6

# Correctness and Performance Testing

## 6.1 Correctness Testing

This section focuses on ensuring that the combinators' behaviour conforms to the specification defined in the System Analysis. The JCC combinators were tested both individually and as part of more complex structures.

### 6.1.1 Individual Wire Combinators

Combinator wires can be treated as black boxes that typically receive a synchronisation request via one boundary, process it and output it through the other boundary. As such they can be tested as independent units where specific input on one boundary is expected to generate certain output on the other.

In order to test behaviour of individual wires, producer and consumer combinators were developed. The producer is a simple implementation of the *Runnable* interface which, when executed, sends a predefined number of random *Integer* messages through a single active port on the right boundary. The consumer combinator receives messages via a passive port on the left and prints out their content. It also keeps count of all messages received.

Figure 6.1 shows two simple compositions that were used to evaluate the behaviour exhibited by the *PushAdaptorWire* (a) and *CopyWire* (b) combinators.

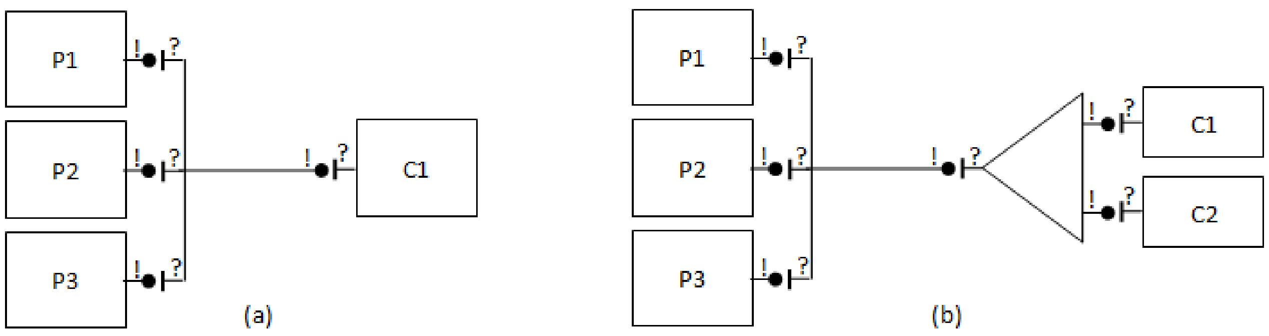


Figure 6.1: Compositions for testing individual wires

First, the *PushAdaptorWire* combinator was tested with three producers; each generating 10 messages. As for the consumer, the expectation was to receive all 30 messages. Once the correctness of the *PushAdaptorWire* was established it was possible use it as part of the structure shown in Figure 6.1 (b) where the main focus was on assessing the behaviour of the *CopyWire* combinator. In this case, each message sent by the producers was expected to be received by both consumers. Furthermore, the semantics of the Copy Wire dictate that all copy branches must complete before another request is processed. Therefore, the value of each message was expected to be output by both consumers in succession.

Similar unit tests were carried out for other combinator wires to ensure that all of them exhibited the behaviour specified in the System Analysis. More details can be found in the *testing* package of project.

### 6.1.2 Complex Structures

While unit-testing represents the first step towards ensuring correctness, combinators must also be tested in more complex structures to guarantee that compositions behave as expected. Figure 6.2 shows a graphical representation of a complex message-passing structure that was used to verify the behaviour of combinator wires with emphasis on failure handling and notification callbacks.

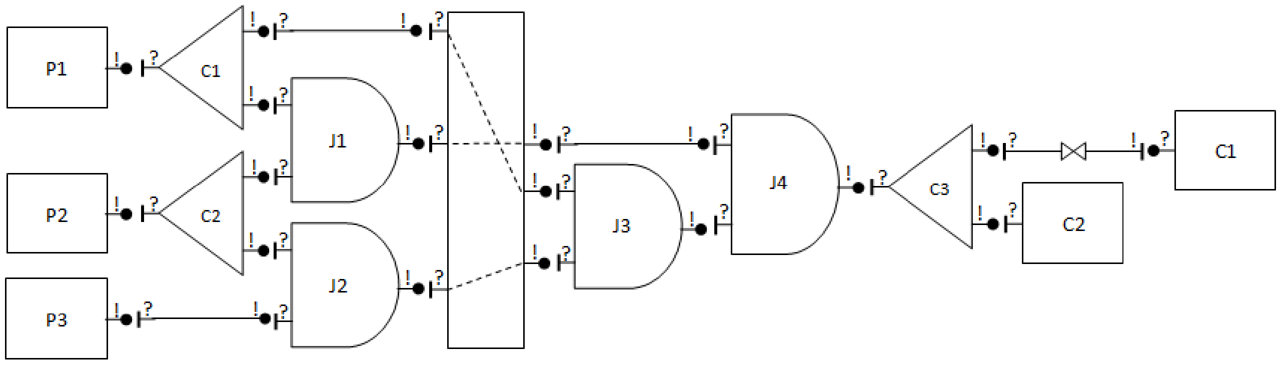


Figure 6.2: Complex message-passing structure created in JCC

In this scenario, producers P1, P2 and P3 implement the *MessageEventHandler* interface to receive notifications when their messages are fully acknowledged or invalidated. They keep sending random *Integer* messages between 0 and 99 until ten messages has been fully acknowledged. The structure of Copy and Join Wires means that all three producers must generate equal values in the right order for the final join message to reach the last Copy Wire before consumers C1 and C2. Even then the message can be invalidated as both consumers use separate *MessageValidators* to decide whether to process the message. The active consumer (C1) only accepts values divisible by 2 while the passive one (C2) requires multiples of 3. That means only multiples of 6 satisfy both consumers.

One of the issues detected when testing this structure was the possibility of a deadlock when messages were invalidated while waiting at passive ports of the Join Wires. Consider the following scenario:

1. P1 produces message #1{2}, P2 produces #2{3} and P3 produces #3{4}
2. C1 copies message #1 and sends out messages #4(#1{2}) and #5(#1{2})
3. Message #4 blocks at the top passive port of J3 waiting for the other join message to arrive. Similarly, message #5 is blocked at J1. Message #3 is also now waiting at the bottom port of J2.
4. C2 now copies message #2 and sends out messages #6(#2{3}) and #7(#1{3})
5. Message #6 reaches J1 and tries to join it with #5. This attempt fails as the messages are not equal and both are invalidated as result.
6. Subsequently, messages #4 and #7 are also invalidated as they were associated with #5 and #6 respectively.

7. Message #7 fails to reach J2 as it is already invalid.
8. As result both copy messages sent out by C2 failed and returned. C2 therefore completes with a failure and releases the P2 thread.
9. However, C1 now hangs as the invalid message #4 is still waiting at J3.
10. P2 produces a new message #8{5}
11. C2 copies #8 and sends out messages #9(#8{5}) and #10(#8{5})
12. Message #9 now waits at J1 while J2 fails to join #10 and #3
13. With its thread released after the failure of J2, P3 produces a new message #11{6}
14. Message #11 arrives at J2 and blocks. But with both C1 and C2 now hanging, no progress is possible - DEADLOCK

In response to this issue, the *currentCarrier* variable of type *Thread* was introduced in the *Message* class. This variable is updated with a reference to *Thread.currentThread()* every time the message passes through a port. This makes it possible to call the *interrupt()* method on the carrying thread as part of the invalidation process. As a result, *InterruptedException* is thrown if a message is invalidated while the carrying thread is blocked (e.g. waiting at the barrier used by the Join Push Wire). The *InterruptedException* is then caught locally and re-thrown as *CombinatorPermanentFailureException*. This prevents the occurrence of deadlocks and ensures liveness.

## 6.2 Performance Testing

### 6.2.1 Tested Data Structures

The first data structure created using the JCC framework was a Backoff Treiber stack shown in Figure 6.3. As mentioned in the Development section, failed CASes on the stack (TS) present themselves as transient failures. These are handled by the Push and Pull Adaptor Wire which deploy the Backoff component and then retry.

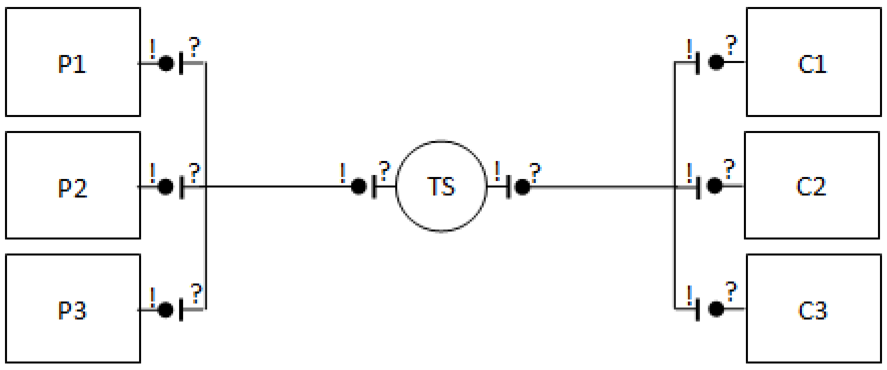


Figure 6.3: Graphical representation of a Backoff Treiber Stack created in JCC

The Elimination Stack shown in Figure 6.4 was the other JCC data structure tested for performance. In this case, transient failures produced by the stack (TS) are handled by the Switch Wires with failed requests forwarded on to the elimination layer (EA).
The aforementioned JCC data structures were tested against a Treiber Stack (without backoff) and Elimination Stack coded by hand. The implementation of these algorithms can be found in the *testing.nocombinators* package of the project.

### 6.2.2 The Producer-Consumer Benchmark

The Producer-Consumer benchmark creates an equal number of active producers and consumers; each executing on an independent thread. The objective is to transfer 10 million messages over the stack in the shortest

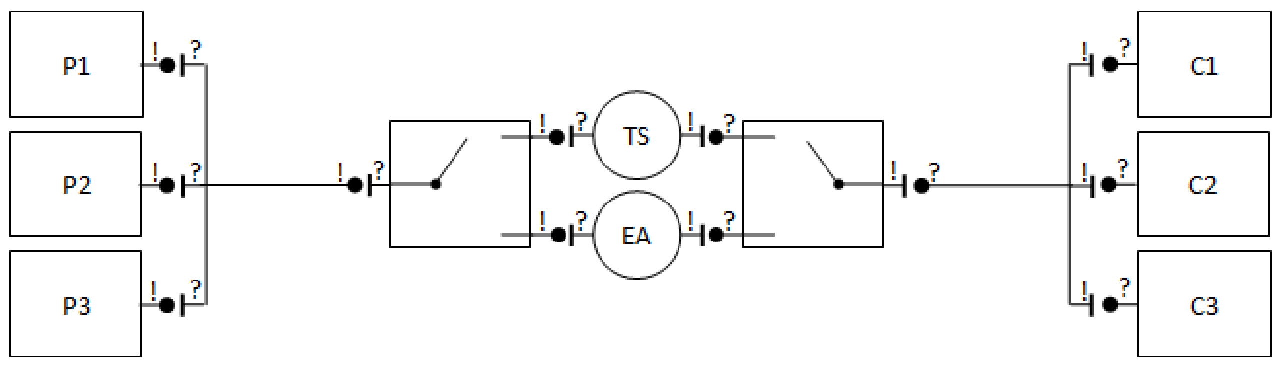


Figure 6.4: Graphical representation of an Elimination Stack created in JCC

possible time.

Test A was set up with 10 producers and 10 consumers; each producing/consuming 1,000,000 messages. This environment represented a low level of contention on the stack.

Test A was set up with 1,000 producers and 1,000 consumers; each producing/consuming 10,000 messages. This environment represented a high level of contention on the stack.

### 6.2.3 Testing Environment

The tests were carried out on an Intel(R) Core(TM) i5-3210M CPU @ 2.50 GHz with 8GB RAM running Windows 7 (64-bit), and using Eclipse Juno IDE with JDK 1.7.0_05.

### 6.2.4 Test Results

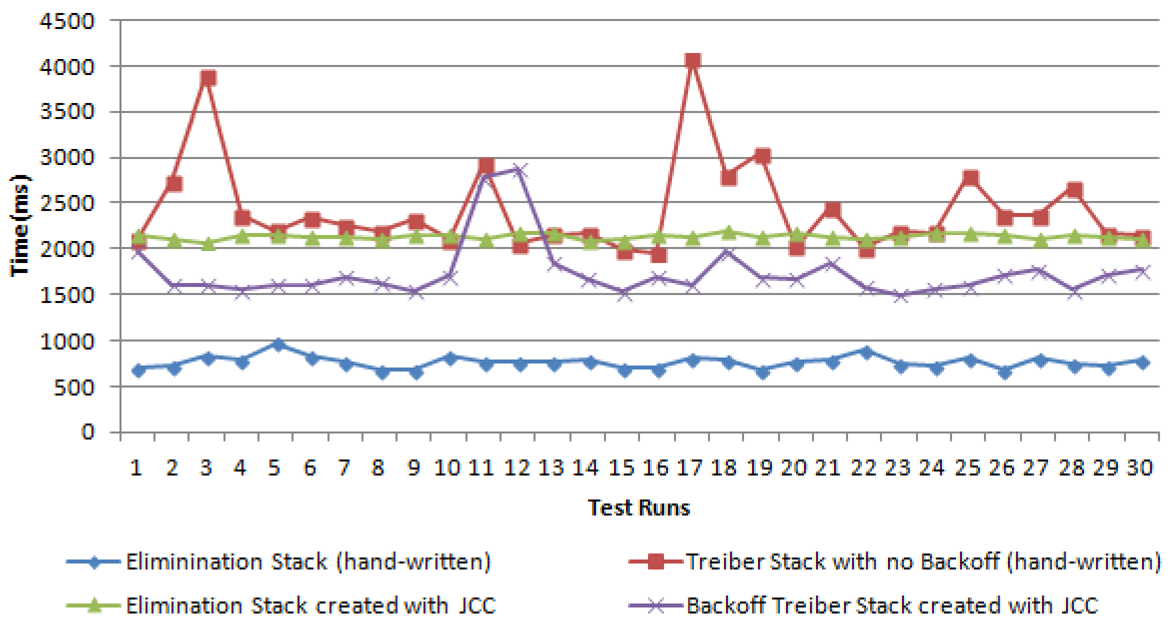


Figure 6.5: The Producer-Consumer Benchmark Results for Test A - 10 Producers & 10 Consumers

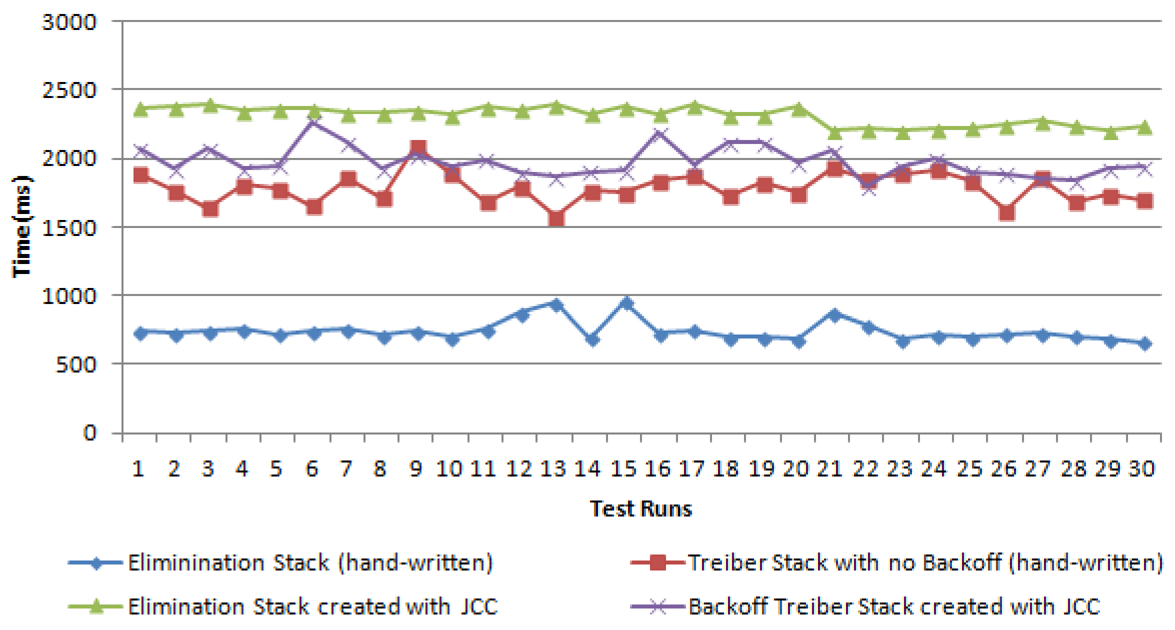


Figure 6.6: The Producer-Consumer Benchmark Results for Test B - 1,000 Producers & 1,000 Consumers

As shown in Figures 6.5 and 6.6 the hand-coded Elimination Stack easily outperformed the other stack implementations in both tests with consistent execution times around 750 ms. Interestingly, both JCC stacks outperformed the hand-coded Treiber Stack without backoff in Test A under low contention but were slower in Test B under high contention. This is a paradox as both the Backoff Treiber Stack and Elimination Stack were expected to perform even better in Test B. One possible explanation could be better compiler optimisation of the hand-written code but this result necessitates further investigation.

However, perhaps most interestingly, the out of the two stacks created with JCC framework the Backoff Treiber Stack consistently outperformed the Elimination Stack. This could suggest that the implementation of the Elimination Exchanger does not perform as expected. Yet, the hand-coded version of Elimination Stack uses exactly the same algorithm and performs very well. It also seems unlikely that the worse-than-expected performance is caused by the Switch Wires as their internal implementation is quite simple. Therefore, this performance issue will require further analysis.

Overall, the unoptimised structures produced by JCC performed 2-3 times slower that the hand-coded Elimination Stack. This can be attributed to additional method calls in the framework and creating a *Message* object for each value sent. Also, unwinding the execution stack by throwing exceptions may have negative implications for the framework's performance. However, this level of performance was expected at this early stage of JCC development and should not be considered a failure.

Detailed performance results can be found in Appendix B.

# Chapter 7

# Future Work

## 7.1 Further Development of the JCC Framework

A number of Wires and Data Combinators were developed as part of this project. However, there is a scope for many more. For example, the Copy Wire provides a synchronous means of communication where the incoming thread is blocked until all copy sub-processes complete. It is possible to imagine scenarios where asynchronous communication would be preferable. Yet, wires providing such functionality are not currently available.

More work also needs to be done with regards to defining precise rules for message passing. JCC currently works well with immutable messages such as numbers. However, there are many unsolved issues when it comes to passing mutable objects. For example, what does it mean to copy or join mutable objects? This is definitely an area which requires further research.

## 7.2 Optimisation and Code Generation

With JCC's formal background in the Wire Calculus, it shall be possible to formalise and reason about the behaviour of individual combinators. This could lead to identifying common patterns and bisimilarities in complex structures. A new compiler preprocessor, that could perform analysis and generate optimised Java code before compilation, could be developed.

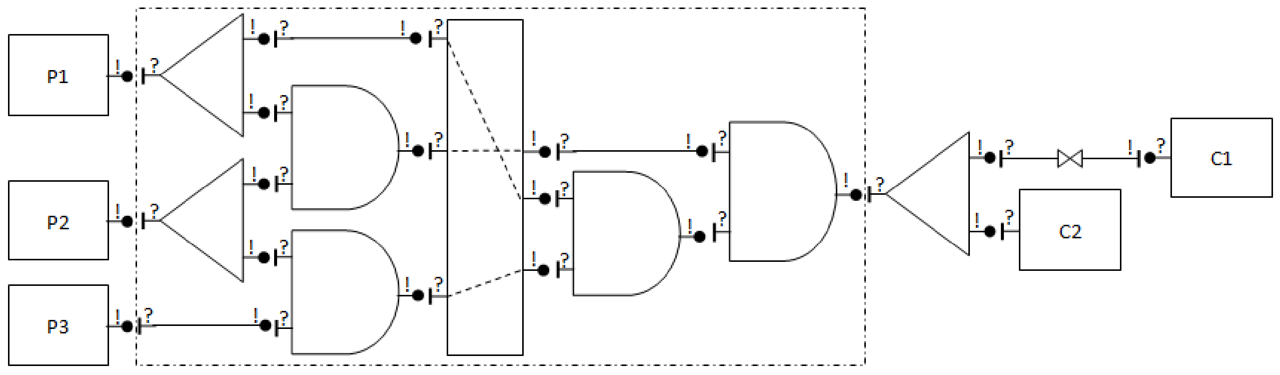


Figure 7.1: Graphical Representation of an unoptimised composition of combinators

Figures 7.1 and 7.2 show an example of possible optimisation of one of the structures defined in this project.

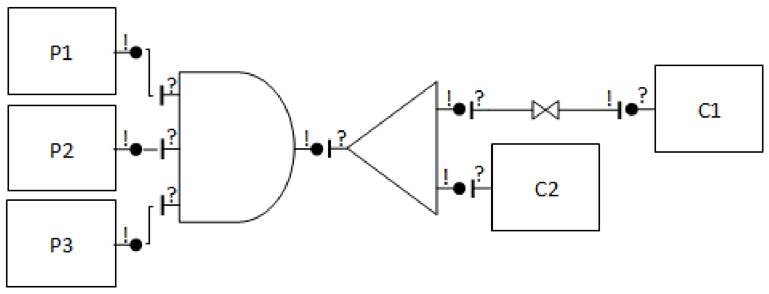


Figure 7.2: Graphical Representation of the composition after optimisation

The complex composition demonstrated in Figure 7.1 can be reduced to a simple Join Wire with three passive ports as shown in Figure 7.2.

## 7.3 Graphical Environment

It is envisaged that, in the future, the JCC framework might provide a sophisticated graphical environment for building complex structures. In principle, this tool could be similar to modern GUI builders. Developers could drag and drop combinators onto a working area and position them next to each other to create horizontal and vertical combinations. The combinator properties such as data type and orientation might be specified via a property panel. The resulting visual representation of JCC structures could look similar to the diagrams shown in this project.

# Chapter 8

# Evaluation

## 8.1 Project Management

The initial set of objectives was created with a very little knowledge of the work involved and, as a result, the work plan presented in the first project brief (see Appendix A) proved unrealistic later on. The original GANTT chart suggested using the waterfall model as the project management methodology with the main activities following each other in a sequence. However, this turned out to be impossible as the precise requirements for the JCC framework could not be established at the beginning of the project.

As a result, the project switched to a rather more agile management methodology where progress was reviewed continuously and features and requirements were adjusted regularly during the weekly project meetings (see Appendix C for details). This change also had a profound effect on the development phase of the project. Rather than gathering all the requirement in advance and then implementing the final product, an incremental approach with prototypes was adopted. For example, an early prototype of JCC did not include the notion of failures, messages were not implemented as futures and the Copy Wire simply took the incoming message and sent down the copy ports. Obviously, this behaviour was undesirable. However, creating these simplified models proved instrumental in arriving at the current solution.

Reflecting on the overall management process, it is obvious that numerous things did not go to plan and many lessons were learnt. However, considering the research nature of the project, it was perhaps unrealistic to expect that the initial work plan would be adhered to in the first place. Most importantly, the majority of the initial objectives were met and a functional product was delivered. Therefore, the project shall be considered successful.

## 8.2 Developed Product

While the JCC framework in its current state represents a functional high-level abstraction over threads it should be considered as no more than a sound foundation for future research. There are many areas that require further investigation and improvement before the framework becomes a useful tool for software developers. Some possible directions for further development are proposed in the previous chapter (Future Work).

Upon reflection, other issues also became obvious. One of the most serious problems stems from the way consumers synchronise over messages. One of the rules states that the message content is not retrievable until all associated messages have been acknowledged. This can lead to deadlocks when associated messages are processed by a single consumer. Consider, for example, the composition in Figure 8.1 where consumer C1 implements the handler for its passive port as follows:

```
...
... = new PassiveInPortHandler<Integer>() {
        @Override
        public void accept(Message<Integer> msg1) {
            Message<Integer> msg2 = (Message<Integer>) receiveLeft(0);
            if(msg1.get() == msg2.get()) {
                ...
            }
        }
    };
...
```

While this might seem a reasonable approach the system actually deadlocks when the operands of the *if* condition are evaluated. The call to *msg1.get()* blocks and waits for the other associated message (msg2) to be also acknowledged before returning the message content. However, this never happens as the call to *msg2.get()* is never reached. As highlighted in the Development chapter, utilizing a *MessageValidator* offers a partial solution. Unfortunately, this has its own drawbacks regarding uncontrollable access to 'dirty' data.

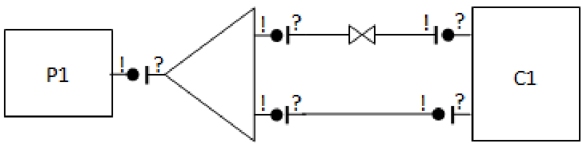


Figure 8.1: Diagram showing a deadlock scenario in JCC

Other potential issues include the growing complexity of the *Message* class which might impact negatively on the framework's performance as more memory must be allocated at construction time. This problem could be tackled by constructing *Message* pools and reusing discarded messages. Yet another performance issue arises from associating producers and their messages via the *MessageEventHandler* interface. Without the direct association, the producers can stay localised and only the messages become visible to whole system. This enables the Java Runtime Environment to manage the state of producer objects faster and more efficiently by using registers and processor-local caches.

Yet, despite the aforementioned shortcomings, the framework achieves its main objective by demonstrating that complex concurrent data structures can be developed by composing well-defined combinators in a controlled way.

## 8.3 Personal Evaluation

On a personal level, this project was a very rewarding experience. The most challenging aspect was the attempt to take purely theoretical concepts from the Wire Calculus and incorporate them into a concrete framework of combinators. Trying to achieve this goal meant investing a great amount of time and energy into the project from the very beginning.

However, this challenge was tackled with a great amount of enthusiasm and determination. While, at times, the workload seemed overwhelming, there was always a sufficient level of commitment to overcome the difficulties. As a result of this hard work, many lessons were learnt and there was a great sense of achievement at the end.

# Chapter 9

# Conclusion

The main objective of this project was to establish whether development of a framework of concurrent combinators was a feasible proposition. This aim was achieved through implementation of the JCC framework which clearly demonstrates the power of composability by providing a rich set of primitive combinators that allow for the composition of more complex concurrent data structures. While the framework provides a functional proof of concept it also exhibits a number of shortcomings in terms of expressiveness and performance. However, it is important to emphasise that the aim of this project was not to deliver a fully-fledged framework for instant use in the industry. While this project made good progress on a long journey towards a new, innovative paradigm for simpler and safer development of concurrent software there is still a significant amount of work to be done before JCC becomes a useful tool for software developers.

# Bibliography

[1] G. Agha. Actors: a model of concurrent computation in distributed systems. 1985.

[2] M. Astley. The actor foundry: A java-based actor programming environment. *University of Illinois at Urbana-Champaign: Open Systems Laboratory*, 1998.

[3] J. Bonér. Exploring alternative concurrency paradigms on the jvm. `http://vimeo.com/20059154`, June 2009. [Online; accessed 27-July-2013].

[4] S. Doeraene and P. Van Roy. A new concurrency model for scala based on a declarative dataflow core. 2013.

[5] F. Drejhammar, C. Schulte, P. Brand, and S. Haridi. *Flow Java: declarative concurrency for Java*. Springer, 2003.

[6] C. Fournet and G. Gonthier. The reflexive cham and the join-calculus. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 372–385. ACM, 1996.

[7] B. Göetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java concurrency in practice*. Addison-Wesley, 2006.

[8] P. Haller and M. Odersky. Actors that unify threads and events. In *Coordination Models and Languages*, pages 171–190. Springer, 2007.

[9] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 206–215. ACM, 2004.

[10] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *ACM SIGPLAN Notices*, volume 41, pages 253–262. ACM, 2006.

[11] C. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[12] G. Kahn, D. MacQueen, et al. Coroutines and networks of parallel processes. 1976.

[13] M. Kaku. Tweaking moore's law and the computers of the post-silicon era. `http://www.geek.com/chips/theoretical-physicist-explains-why-moores-law-will-collapse-1486677/`, Apr. 2012. [Online; accessed 10-August-2013].

[14] T. Kelly, Y. Wang, S. Lafortune, and S. Mahlke. Eliminating concurrency bugs with control engineering. *Computer*, 42(12):52–60, 2009.

[15] G. Korland, N. Shavit, and P. Felber. Noninvasive concurrency with java stm. In *Third Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG)*, 2010.

[16] D. Lea. The java.util.concurrent synchronizer framework. *Science of Computer Programming*, 58(3):293–309, 2005.

[17] E. Lee et al. The problem with threads. *Computer*, 39(5):33–42, 2006.

[18] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 73–82. ACM, 2002.

[19] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275. ACM, 1996.

[20] M. M. Michael and M. L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, 1998.

[21] M. Moir and N. Shavit. Concurrent data structures. *Handbook of Data Structures and Applications*, pages 14–47, 2007.

[22] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala, 2/e.* Artima Series. Artima Press, 2010.

[23] J. Schäfer and A. Poetzsch-Heffter. Jcobox: Generalizing active objects to concurrent components. *ECOOP 2010–Object-Oriented Programming*, pages 275–299, 2010.

[24] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[25] P. Sobociński. A non-interleaving process calculus for multi-party synchronisation. In *ICE '09*, 2009.

[26] S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for java. *ECOOP 2008–Object-Oriented Programming*, pages 104–128, 2008.

[27] V. Subramaniam. Programming concurrency on the jvm: Mastering synchronization, stm, and actors author: Venkat subramaniam, publisher: Pra. 2011.

[28] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):202–210, 2005.

[29] R. K. Treiber. *Systems programming: Coping with parallelism.* International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

[30] A. Turon. Reagents: expressing and composing fine-grained concurrency. In *ACM SIGPLAN Notices*, volume 47, pages 157–168. ACM, 2012.

# Appendices

# Appendix A

# Project Brief

# 2013 Electronics and Computer Science MSc Project Brief

| | | | | | |
|---|---|---|---|---|---|
| **Name** | **Ales Cirnfus** | **ID no** | **25728067** | **Email** | **ac6g12@soton.ac.uk** |

| | | | |
|---|---|---|---|
| **Supervisor** | **Dr Julian Rathke** | **Date of 1st meeting** | **03/06/2013** |
| **Co-supervisor** | ---------------------------- | | |
| **Project title** | **JCC: Java framework for building concurrent data structures using combinators** | | |

**Background**

Demands on hardware performance grow progressively as we build more complex systems. For a long time, these requirements have been met by advances in chip speeds and transistor density; phenomenon known as Moore's Law. However, these exponential increases in performance are no longer sustainable due to the limits set by the laws of thermodynamics and quantum physics. As we begin to reach these limits, performance boosts are often delivered through parallelism (e.g. multi-core processors, multi-processor computers). This shift in hardware architectures has a fundamental effect on the way software is written. Software developers must embrace concurrent programming to take advantage of the power offered by parallelism. Yet, there is a broad consensus that developing robust concurrent systems in general-purpose object-oriented languages is fraught with difficulties. However, there is much less consensus between both researches and practitioners when it comes to the question of solving this problem. Some say that concurrency models based on threads, locks and shared memory are so fundamentally broken they cannot be fixed. Yet others believe that it is possible to build safe and efficient high-level concurrency models on top of threads.

**Aims**

The main aim of this project is to develop a framework of Java Concurrent Combinators (JCC); an extensible set of building blocks that can be combined both vertically and horizontally to create complex concurrent data structures with minimum effort. The core combinators should encapsulate common concurrency patterns (e.g. optimistic retry loop, backoff scheme) and provide implementations of general concurrent algorithms (e.g. lock-free stacks and queues). Communication between combinators should be provided through a message passing system with particular emphasis on atomicity and failure handling. The core building blocks provided by the framework should be subsequently composed into complex concurrent data structures and tested for correctness and performance.

This project builds on the ongoing research into process algebra carried out by Dr Pawel Sobocinski and includes ideas presented by Kristian Elliott in his MEng Computer Science dissertation.

**Expected Outcomes**

Showing, through the creation of a high-level concurrency framework, that thread-based concurrency can be used effectively in a safe and efficient fashion. The proposed framework shall alleviate many problems commonly associated with thread-based concurrency, while also providing a high degree of composability and expressiveness with additional safety and correctness guarantees (type safety, message atomicity, lack of deadlock).

| | |
|---|---|
| **Does your project involve laboratory work?** | [✓ **NO**] / **YES** |
| **I have completed a risk assessment form** | [✓ **NO**] / **YES** |
| **Does your project involve human subjects?** | [✓ **NO**] / **YES** |
| **I have completed an ethics application** | [✓ **NO**] / **YES** |

**<u>Main Objectives and Deliverables</u>**

**Concurrency Research Report**

This report should provide an overview of different concurrency models and frameworks built on top of them. In particular, it should focus on thread-based concurrency and frameworks executable on the Java Virtual Machine.

- Find and review academic journals relating to concurrency models and frameworks
- Read suitable literature to gain a good understanding of concurrency support provided in Java
- Assess the relevance and implications of the findings to the proposed product

**Concurrent Data Structures and Algorithms Research Report**

This report should review concurrent data structures and common concurrency patterns that are typically used when implementing these structures. The report should also focus on different ways of thread synchronisation (blocking with locks, lock-free, wait-free) through which these structures can be implemented.

- Find and review academic journals relating to design and implementation of concurrent data structures
- Review the benefits and trade-offs offered by different ways of thread synchronisation
- Consider Java features relevant to different thread synchronisation practices

**Feasibility Study**

This study should determine if it is feasible, in the view of the previously carried out research, to achieve the primary aim of this project – creating the Java Concurrent Combinators framework which will provide a high degree of composability and expressiveness as well as safety and correctness guarantees.

- Determine whether the Java language provides sufficient features for implementation of the proposed product
- Determine whether there are any legal implications associated with the implementation of the proposed product
- Establish whether there are any other factors that could seriously impede delivery of the required product within the given timescale

**System Analysis Report**

In this part the project boundaries should be clearly defined. Using a range of fact-finding approaches, problem analysis should be carried out and both functional and non-functional requirements should be gathered.

**Design Report**

This report should contain details of all design decisions and proposed solutions including UML diagrams and other documentation such as test plans.

**Implementation Notes**

This part should provide a record of what went on during the implementation stage including a list of encountered problems and their solutions. All prototypes should be mentioned and briefly documented.

- Specify the development methodology
- Document all prototypes developed during this stage
- Keep a logbook of all encountered problems
- Document solutions to the encountered problems and justify why a particular approach was used

**Test Report**

This section should document how testing was carried out, elaborate on the results of the tests and establish if the product provides the expected levels of expressiveness, composability and performance, as well as the aforementioned safety and correctness guarantees.

- Describe testing techniques used in this project
- Document unit testing during implementation stage and state how it helped in delivering fully functional combinators

**Evaluation and Reflection Report**

This part should reflect on the way the project was managed and carried out. It should state what went well and what did not go so well. Could/should something have been done differently? What could be done in the future to make the product better?

- Reflect on project management
- Evaluate the research conclusions
- Evaluate the final product and state its strengths and weaknesses
- Identify future work

## Work Plan and Milestones

| **Week #** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ***week beginning :*** | **3/6** | **10/6** | **17/6** | **24/7** | **1/7** | **8/7** | **15/7** | **22/7** | **29/7** | **5/8** | **12/8** | **19/8** | **26/8** | **2/9** |
| **Concurrency Research** | ■ | ■ | ■ | ■ | ■ | ■ | | | | | | | | |
| **Data Structures Research** | ■ | ■ | ■ | ■ | ■ | ■ | | | | | | | | |
| **Milestone** – Literature review complete | | | | | | ✕ | | | | | | | | |
| **Feasibility Study** | | | | ■ | | | | | | | | | | |
| **System Analysis** | | | | | ■ | | | | | | | | | |
| **System Design** | | | | | | ■ | ■ | | | | | | | |
| **Implementation** | | | | | | | ■ | ■ | ■ | ■ | | | | |
| **Testing** | | | | | | | | | ■ | ■ | ■ | | | |
| **Milestone** – Product implemented and tested | | | | | | | | | | | ✕ | | | |
| **Evaluation** | | | | | | | | | | | | ■ | | |
| **Final Report Composition** | | | | | | | | | | | ■ | ■ | ■ | ■ |
| **Milestone** – demonstrate to supervisor/examiner | | | | | | | | | | | | ✕ | | |
| **Milestone** – dissertation draft complete | | | | | | | | | | | | | ✕ | |
| **Final Corrections** | | | | | | | | | | | | | | ■ |
| **Milestone** – Hand-in | | | | | | | | | | | | | | ✕ |

Description and dates of milestones:

| | | |
|---|---|---|
| 1. | 14th July | Research of relevant topic complete and a literature review written |
| 2. | 18th August | Delivery of a fully functional and tested product |
| 3. | 19th - 31st August | Demonstration of the complete product to the supervisor and second examiner |
| 4. | 26th August | A complete draft of the dissertation |
| 5. | 6th September. | Hand in final copy of dissertation |

# Appendix B

# Performance Testing Results

## Test A results in milliseconds

| Test Number | Eliminination Stack (hand-coded) | Treiber Stack with no Backoff (hand-coded) | Elimination Stack created with JCC | Backoff Treiber Stack created with JCC |
|---|---|---|---|---|
| **1.** | 709 | 2113 | 2146 | 1970 |
| **2.** | 724 | 2733 | 2115 | 1598 |
| **3.** | 835 | 3894 | 2063 | 1598 |
| **4.** | 789 | 2365 | 2156 | 1551 |
| **5.** | 977 | 2203 | 2151 | 1611 |
| **6.** | 835 | 2342 | 2126 | 1601 |
| **7.** | 760 | 2252 | 2137 | 1694 |
| **8.** | 686 | 2187 | 2108 | 1618 |
| **9.** | 680 | 2322 | 2157 | 1539 |
| **10.** | 826 | 2105 | 2143 | 1702 |
| **11.** | 762 | 2944 | 2103 | 2773 |
| **12.** | 762 | 2061 | 2163 | 2867 |
| **13.** | 771 | 2159 | 2181 | 1849 |
| **14.** | 785 | 2163 | 2091 | 1663 |
| **15.** | 697 | 1997 | 2094 | 1527 |
| **16.** | 699 | 1961 | 2154 | 1695 |
| **17.** | 807 | 4083 | 2127 | 1610 |
| **18.** | 802 | 2805 | 2182 | 1966 |
| **19.** | 685 | 3035 | 2137 | 1679 |
| **20.** | 761 | 2027 | 2169 | 1661 |
| **21.** | 789 | 2464 | 2134 | 1852 |
| **22.** | 897 | 2009 | 2113 | 1584 |
| **23.** | 746 | 2200 | 2124 | 1502 |
| **24.** | 731 | 2180 | 2180 | 1563 |
| **25.** | 805 | 2808 | 2176 | 1591 |
| **26.** | 682 | 2373 | 2153 | 1713 |
| **27.** | 807 | 2364 | 2107 | 1766 |
| **28.** | 758 | 2667 | 2148 | 1549 |
| **29.** | 730 | 2160 | 2136 | 1708 |
| **30.** | 791 | 2146 | 2111 | 1770 |
| **Average (ms):** | **769.6** | **2437.4** | **2136.17** | **1745.67** |
| **Median (ms):** | **762** | **2227.5** | **2137** | **1662** |
| **Average (%):** | **100.00** | **316.71** | **277.57** | **226.83** |
| **Median (%):** | **100.00** | **292.32** | **280.45** | **218.11** |

## Test B results in milliseconds

| Test Number | Eliminination Stack (hand-coded) | Treiber Stack with no Backoff (hand-coded) | Elimination Stack created with JCC | Backoff Treiber Stack created with JCC |
|---|---|---|---|---|
| **1.** | 744 | 1888 | 2370 | 2067 |
| **2.** | 732 | 1763 | 2384 | 1923 |
| **3.** | 746 | 1638 | 2399 | 2072 |
| **4.** | 756 | 1809 | 2351 | 1922 |
| **5.** | 723 | 1778 | 2362 | 2269 |
| **6.** | 744 | 1653 | 2363 | 2269 |
| **7.** | 755 | 1857 | 2338 | 2113 |
| **8.** | 712 | 1716 | 2340 | 1929 |
| **9.** | 749 | 2090 | 2348 | 2028 |
| **10.** | 700 | 1888 | 2318 | 1941 |
| **11.** | 764 | 1685 | 2374 | 1992 |
| **12.** | 876 | 1794 | 2358 | 1900 |
| **13.** | 952 | 1576 | 2391 | 1868 |
| **14.** | 695 | 1763 | 2329 | 1906 |
| **15.** | 955 | 1747 | 2373 | 1910 |
| **16.** | 728 | 1840 | 2329 | 2191 |
| **17.** | 753 | 1872 | 2389 | 1951 |
| **18.** | 697 | 1731 | 2322 | 2120 |
| **19.** | 702 | 1825 | 2318 | 2117 |
| **20.** | 684 | 1747 | 2380 | 1969 |
| **21.** | 872 | 1935 | 2206 | 2061 |
| **22.** | 790 | 1856 | 2217 | 1796 |
| **23.** | 682 | 1888 | 2206 | 1946 |
| **24.** | 720 | 1919 | 2223 | 2001 |
| **25.** | 701 | 1841 | 2225 | 1905 |
| **26.** | 725 | 1622 | 2248 | 1891 |
| **27.** | 735 | 1857 | 2273 | 1856 |
| **28.** | 710 | 1685 | 2240 | 1835 |
| **29.** | 684 | 1732 | 2206 | 1924 |
| **30.** | 665 | 1700 | 2240 | 1937 |
| **Average (ms):** | **748.38** | **1789.83** | **2314.03** | **1976.27** |
| **Median (ms):** | **730** | **1786** | **2333.5** | **1943.5** |
| **Average (%):** | **100.00** | **239.17** | **309.21** | **264.08** |
| **Median (%):** | **100.00** | **244.66** | **319.66** | **266.23** |

# Appendix C

# Progress GANTT Chart

## Work Plan and Milestones vs. Actual Completed Work

| **Week #** / ***week beginning :*** | **1** **3/6** | **2** **10/6** | **3** **17/6** | **4** **24/7** | **5** **1/7** | **6** **8/7** | **7** **15/7** | **8** **22/7** | **9** **29/7** | **10** **5/8** | **11** **12/8** | **12** **19/8** | **13** **26/8** | **14** **2/9** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Concurrency Research** | ■ 🟩 | ■ 🟩 | ■ 🟩 | ■ | ■ | ■ | | | 🟩 | 🟩 | 🟩 | | | |
| **Data Structures Research** | ■ | ■ | ■ | ■ | ■ | ■ 🟩 | 🟩 | 🟩 | | | | | | |
| **Milestone** – Literature review complete | | | | | | X | | | | | | | 🟩X | |
| **Feasibility Study** | | | 🟩 | ■ | | | | | | | | | | |
| **System Analysis** | | | | 🟩 | ■ | | 🟩 | | | | | | | |
| **System Design** | | | | 🟩 | 🟩 | ■ | ■ | 🟩 | | | | | | |
| **Implementation** | | | | | | 🟩 | ■ 🟩 | ■ | ■ 🟩 | ■ 🟩 | 🟩 | 🟩 | | |
| **Testing** | | | | | | | 🟩 | 🟩 | ■ 🟩 | ■ 🟩 | ■ 🟩 | 🟩 | | |
| **Milestone** – Product implemented and tested | | | | | | | | | | | X | 🟩X | | |
| **Evaluation** | | | | | | | | | | | | ■ 🟩 | | |
| **Final Report Composition** | | | | | | | | | | | ■ 🟩 | ■ 🟩 | ■ 🟩 | ■ |
| **Milestone** – demonstrate to supervisor/examiner | | | | | | | | | | | | X | 🟩X | |
| **Milestone** – dissertation draft complete | | | | | | | | | | | | | X 🟩X | |
| **Final Corrections** | | | | | | | | | | | | | ■ | ■ 🟩 |
| **Milestone** – Hand-in | | | | | | | | | | | | | | X 🟩X |

Description and dates of milestones:

| | | | |
|---|---|---|---|
| 1. | 14th July | (planned) | Research of relevant topic complete and a literature review written |
| | 16th August | (completed) | |
| 2. | 18th August | (planned) | Delivery of a fully functional and tested product |
| | 22nd August | (completed) | |
| 3. | 19th - 31st August | (planned) | Demonstration of the complete product to the supervisor and second examiner |
| | 27th August | (completed) | |
| 4. | 26th August | (planned) | A complete draft of the dissertation |
| | 31st August | (completed) | |
| 5. | 6th September | (planned) | Hand in final copy of dissertation |
| | 5th September | (completed) | |