Communicating Process Architectures 2017
K. Chalmers, J.B. Pedersen et al. (Eds.)
IOS Press, 2017
© 2017 The authors and IOS Press. All rights reserved.

What are Communicating Process Architectures? Towards a Framework for Evaluating Message-passing Concurrency Languages

Kevin CHALMERS [1]

School of Computing, Edinburgh Napier University

Abstract. What does it mean to be a message-passing concurrent language? This work attempts to build a framework for classifying such languages by judging four in regards to features and performance. Features of process calculi are used to evaluate Go, Rust, Erlang, and occam-pi. Furthermore, standard communication time, selection time, and multicore utilisation are examined. Although each of these languages use message-passing concurrency, their approaches and characteristics are different. We can start to build an initial classification based on message-passing type, language support, and feature support. Such classification allows an initial discussion of the suitability of the evaluation framework, whether it is useful, and how it can be expanded. Approximately 50 further languages have been identified as potentially supporting message-passing concurrency to further build up the classification.

Keywords. evaluation framework, Erlang, Go, Rust, occam-π

Introduction

In recent years, message-passing concurrency has gained wider adoption in mainstream programming languages. Message-passing is the ability to send data to a component which, under its own thread of control, decides how and when to handle the message. Such an approach to concurrency enables behavioural reasoning of a system using techniques such as Communicating Sequential Processes (CSP) [1], the π-Calculus, and other process calculi. Languages such as Google's Go, Mozilla's Rust, and Ericsson's Erlang – particularly in its Elixir flavour – have seen rapid adoption in projects ranging from the large-scale to the small-scale. As such, it can be argued that message-passing concurrency is becoming a tool with a growing exposure in the general software engineering community.

Suitable methods to evaluate message-passing languages has not enjoyed similar wider adoption. Unlike object-orientation and structured programming techniques before it, no clear set of metrics, benchmarks, and properties have been defined to allow software engineers to evaluate the suitability of a message-passing language for undertaking a particular task, nor enable easy understanding of where to address performance bottlenecks. Two key features for a software engineer, performance and maintainability, are not well understood in message-passing applications. Software metrics, design principles, and a common vocabulary exist for object-orientation, as do benchmark applications that allow software engineers

[1]Corresponding Author: *Kevin Chalmers, School of Computing, Edinburgh Napier University, 10 Colinton Road, Edinburgh.* Tel.: +44 131 455 2484; E-mail: `k.chalmers@napier.ac.uk`.

to understand the performance of a given language's produced executables. For message-passing applications, such a set of metrics, features, vocabulary, and benchmarks does not exist, highlighting a current gap in the software engineer's toolbox.

The work presented here is an attempt to bridge the gap in the message-passing software engineer's toolbox. It does so by proposing a set of possible message-passing language features, and some benchmark applications to allow extraction of properties seen as important to the message-passing language programmer. Language features allow us to define a set of principles and a common vocabulary for message-passing languages, much like object-orientation has ideas of specialization, generalization, and encapsulation. Properties relate to areas important to the use of the language in production code, such as message communication time and message selection time. Due to message-passing languages use in high-availability systems, the focus of the property extraction will be predominately on communication. Due to message-passing languages claiming inspiration from process calculi, this will be the main focus for drawing out language features. The produced features, properties, and benchmarks are by no means complete, nor should they be considered a definitive set to support in a language. Rather, this a first step towards a framework with an agreed consensus. The aim of such a framework is to provide the software engineer with the ability to make useful choices when working with message-passing concurrency languages.

This paper will explore three properties that can be used to evaluate a message-passing concurrent language: conformance to process calculi features; performance properties; and ease of use and the supporting APIs. Conformance to process calculi will be measured by examining which features the language provides that are defined in process calculi. Performance properties will focus on coordination ability. Ease of use and API support are subjective measures to an extent, and a rough classification of API features will be used. This defines the experimental area for this work, which will be described in more detail in Section 2.

1. Background and Related Work

In this section, a framework for evaluating message-passing concurrent languages is examined. As one of the premises of this work is around how a language is selected, an initial discussion on this area is presented.

1.1. Motivation

It has become clear that message-passing concurrency has become popular, in that popular languages such as Go and Rust have been released and been successful in commercial products such as Docker – which is written in Go. Redwine et. al. [2] defined a model for understanding software technology maturation, beginning with basic research, and ending with popularization. For message-passing concurrency, the basic research undertaken by Hoare [1] and Milner [3] into concurrency led to initial approaches as seen in occam and the Transputer, and via Limbo to the development of Go and Rust. This work proposes that to go from message-passing being a language feature to an understood architectural concept for software engineering, further work is required to define message-passing concurrency quality.

1.2. Programming Language Selection

One of the key premises of this paper is that we want to select between programming languages. Such a statement makes sense as we choose a language to meet our purposes to solve a particular problem in a particular domain. However, beyond classification of languages into types (e.g. imperative, functional, logical, etc.), or picking a language because it supports a very particular feature we are interested in, what measure can be used to select between programming languages?

Parker et. al. [4] have looked at the history of programming language selection in academia. Academic selection of programming languages is a widely studied and argued field, but in itself is not the focus of this work. One of Parker et. al.'s core beliefs is that today a language is mature when it starts to be taught in a university. This information is relatively easy to find for the message-passing languages mentioned previously: Go has a list of courses available (`https://github.com/golang/go/wiki/Courses`), Rust has been taught at the University of Pennsylvania (`http://cis198-2016s.github.io/`), and both the University of Kent (`https://www.cs.kent.ac.uk/ErlangMasterClasses/`) and the University of Oxford (`https://www.cs.ox.ac.uk/softeng/subjects/CPR.html`) provide courses in Erlang.

Although Parker et. al.'s selection criteria is based on university adoption, they draw two distinct reasons for selection: pragmatic or pedagogic. Of the former, criteria such as industry adoption and marketability are included. For the latter, avoiding complexity and problem-solving capabilities are included. In both instances, if applied to the objective of this work, then language selection could be based on the scale of industrial use, or it could be done based on its capability to solve problems.

Tharp [5] also examines programming language selection through the lens of teaching. Eight criteria are used to evaluate a languages suitability, four of which are useful to the discussion presented here:

- *the existence of control structures to support a preferred programming methodology.*
- *the existence of adequate diagnostic aids and other programming tools.*
- *its interfaces with existing software.*
- *the existence of literature and program libraries for the language.*

Tharp's other four criteria – inclusion of data types and structures to the subject area, availability on a given machine, interfaces with special equipment, and vendor support – are today less of a concern. The data types criteria could be argued for, but languages generally provide constructs to create types, and the criteria to interface with other software allows overcoming of most other issues.

From the selected four criteria, the first (existence of control structures) is a key criteria for selecting a message-passing language, and shall be revisited in the context of this work in Section 2. Debugging tool support and other tools are an important criteria which shall not be visited in this paper, and is left to future work. Interfaces with existing software is an interesting criteria. In 1982 (the year Tharp's work was published), the focus would have been very different to today. Today, interfacing with existing software consists of fundamentally two approaches: being able to call libraries written in another language (normally C), and the ability to send messages to another application, which will be generalised to communicating with a distributed system (which could be on the local machine).

Tharp's final criteria – essentially program library support – is important to many programmers today, and is therefore an important criteria to analyse.

Consideration has been made on concurrency in an object-oriented language. Nelson [6] has expressed communication models (asynchronous, synchronous, procedure invocation) and the costs associated with these models. The argument centres on an object-oriented language's ability via modularity to support concurrency and easily. This contradicts the work of...

1.3. Language Evaluation

Evaluating a programming is a difficult and subjective matter. The first question to ask if proposing such an evaluation are the types of measures that are important. A possible measure is the popularity of a language, seen in such places as the Tiobe language index (`http://www.tiobe.com/tiobe-index/`) which uses Internet search engines to determine a language's popularity. Other language popularity measures such as GitHub (`http://githut.info/`) also exist. Both of these measures place what could be considered mainstream languages (JavaScript, Java, C++, etc.) in the top ten, although languages such as Google's Go, Mozilla's Rust, and Erlang have seen increases in activity. The concept of popularity can be extended to include lines of code written, although older languages such as C will have an advantage. The problem with a simple popularity measure is it tells us little objectively about the suitability of a language to perform a task.

The understanding of programming as a practice is not a new one. As early as 1974, Yohe [7] expressed ideas of "good programming". Yohe states that language selection is a subjective decision, but that amount of material available is important – indicating that new languages have a harder time due to the wealth of material available online for existing languages. The key is finding the right language for the problem. This does raise a question about the problem that message-passing concurrency aims to solve. Is it improved multi-core utilisation? Is it improved reasoning around a system with multiple active components?

Possible methods of evaluating a languages suitability for performing a task can include the performance of executables, or the amount of code (Lines of Code or LoC) to solve a problem. The latter measure would be arguably the expressiveness of the language. To allow a fair measure, experts in each language would have to solve given problems, and the solutions compared. However, this could lead to competitions where single line solutions are compared, which doesn't necessarily equate to a more expressive language. Finally, the endurance or resilience of a language could be considered, with languages such as C still being used despite its relative age. But the implication here is that older languages are better, which is not a useful measure either.

AlGhamdi [8] has defined two types of language classification: qualitative analysis on a feature basis; and quantitative analysis via metrics on programs. It is noted that methods have not really been developed, although it is important for developers to compare languages subjectively to determine suitability. AlGhamdi's work is a checklist of features, with ideas such as language philosophy, readability, and portability being examined. A criteria of importance to software engineers is perhaps that of language cost: cost of learning the language, cost of writing code, cost of executing code, and cost of maintaining code being defined as measures to consider. The idea of cost match well with other quantitative measures such as LoC and executable performance.

Evaluations of programming languages have taken place. Belkhouche et. al. [9] compared Ada and Modula-2 by defining a set of seven criteria: formal definition, construct flexibility (orthogonality), machine independence, similarity to common languages, uniformity, extensibility, and abstraction and encapsulation. It can be argued that most modern languages are machine independent in their standardised form. It is also arguable that language similarity is a useful criteria as today it would mean any language which is not C like would be viewed negatively.

1.4. Languages Examined

Four languages have been selected for the initial evaluation. Two of the languages are older: Erlang and occam-π. Both were developed in the 1980s, but their philosophy in message passing is different. Erlang uses a mailbox system, and is referred to as an actor language. occam-π is a channel closely linked with CSP and therefore uses channels.

The other two languages are new. Go was released in 2009, and Rust in 2010. Go is supported by Google whereas Rust is supported by Mozilla. Rust is the least mature language examined, having undergone some major revisions since launch, and with some experimental features in the language. Both of these languages are proving popular at time of writing.

Rust is the only language to use operating system threads in its concurrency model, thus limiting the total number of active components. Erlang, Go, and occam-π all use an internal scheduler to support lightweight processes.

1.5. Objectives

The objective of this work is to examine a framework for evaluating message-passing languages. The results from the evaluation are of interest, but so is the suitability of the framework to undertake the evaluation. Therefore, and language comparisons and classifications at this stage should be seen as initial. The work therefore has two key objectives:

1. Define a framework for evaluating message-passing languages.
2. Apply this framework to the evaluation of some message-passing languages.

The rest of this paper is broken down as follows. In Section 2 the evaluation framework is defined, including process calculi features, benchmarks, and library support. In Section 3 the results of applying this framework on Erlang, Go, Rust, and occam-π are presented. Finally, in Section 4 some conclusions are drawn.

2. Evaluating Message-passing Languages

What is a message-passing language? This an important question to answer given the nature of the analysis presented. A language that supports message-passing is defined by the following three criteria:

1. the language *must* provide mechanisms to send a message between components as part of the core language features (e.g., keyword support and/or standard library) and not via an additional library.
2. the message must be sent in a manner so that the receiver can *choose* when to receive it; therefore, giving the receiver control over its internal state. A method invocation on an object is therefore not a message.
3. messages must be any structured data type supported in the language. Conversion to bytes, strings, or another data serialization technique is not considered message-passing.

These three criteria are all we require. The criteria immediately removes object-orientated languages such as Java, C#, and C++ when an object is a passive receiver of an invocation. The criteria also ignores the event handling mechanisms seen in such languages, which are also just method invocations on objects. The definition does not specify the need for channels – thus including Erlang and its related languages – and also does not require a component to be an active, first-order process – thus including Google's Go. The definition does not even state that a multithreaded or multicore supported runtime is necessary.

The requirement for inbuilt language support rather than external library support is also important, especially when combined with the need to send structured data. It is easy to create a message-passing mechanism with existing concurrency primitives, which has led to numerous libraries such as JCSP, PyCSP, etc. The definition also means that simple pipes between threads are not included, nor C signals, as they do not allow general structured data to be sent between components.

In this paper, the term language is used to mean both the actual language being discussed (keywords and form) as well as any supporting runtime that the language executes within.

With the definition in place, the criteria for evaluating the selected languages can be defined. Two separate classifications of criteria are used. The first set comes from the inspiration for many message-passing languages: process calculi. Two such calculi are used –

Communicating Sequential Processes (CSP), and the π-Calculus. CSP defines a number of core primitives and mechanisms and CSP itself was designed to be used as a programming language of sorts. The π-Calculus provides higher-order primitives, allowing consideration of channels and processes as first-class in their own right.

Although process calculi provide a rigorous set of primitives and mechanisms, they do not specify the properties many programmers are interested in: simplicity and efficiency. Simplicity is a somewhat subjective measure, but a LoC (Lines of Code) metric can be used to determine the relative complexity of solutions between languages. Efficiency relates to the performance of the language in certain conditions, and different efficiency measures can be measured using benchmarks. Both of these factors can provide a richer picture to compare the languages under test.

As such, a set of benchmark applications have been developed to allow the exploration of both the process calculi features, and the performance and efficiency metrics of the chosen message-passing languages.

2.1. Process Calculi Properties

From a study of process calculi, the following properties of interest have been identified.

Synchronous communication Component synchronization means for communication to occur the components must agree to perform the action and do so at the same time. Synchronous communication allows reasoning within process calculi, and therefore underpins correctness claims.

First-order channels Message-passing concurrency languages use one of two techniques: sending messages directly to a component; or sending messages using channels. A channel allows a component to abstract its interface, meaning a component only needs to know its own interface. External components are not known. Channels are the connector type in a message-passing architectural model.

Higher-order channels A higher-order channel can be communicated via another channel. That is, we have channels which have the communication type of a channel. Higher-order channels come from the π-Calculus, although they have been illustrated in CSP also. Higher-order channels allow reconfiguration of a system's communication infrastructure at runtime.

First-order processes A first-order process is where a specific process type is provided in the language. Our definition of a message-passing concurrent language means that an instance of this type is in control of the messages it receives, implying a thread of control for the component instance. A process type allows distinction from data types and functions in the language.

Higher-order processes Like higher-order channels, a higher-order process is one which can be communicated to other processes. This means a process must be a component that can be instantiated and assigned to a variable and the type of that variable communicable to other processes. Higher-order processes allow process ownership to be transferred, and process ownership is an important property to process calculi.

Parallel execution statement A key feature of process calculi is a statement to express concurrency. In the simplest terms, a parallel execution statement expresses that two processes will execute concurrently at the given point in the system definition, and the that the definition expressing the concurrency now has two executing behaviours which continue until they are both completed. Parallel execution statements can be further chained together to execute multiple processes concurrently.

In a language, a statement must be provided via a keyword or single standard library function call which states that a set of components should be created and spawned off with their own threads of control. We deliberately specify a set of components, as by definition the languages will have a method to spawn a single component. In our definition, we are interested in the creation and starting of a collection of components. Such a definition is important for the properties of indexed parallel and process ownership.

A further property of parallel execution is the idea that the calling process must wait for the statement to complete. This means that a parallel execution is a statement that does not complete until all of its spawned components complete. This concept is also important for process ownership.

Indexed parallel execution statement (parallel for) CSP also provides an indexed parallel expression which can be defined as a *parallel for* statement in a concurrency language. Although a parallel for is well understood, and provided in frameworks such as OpenMP, little support is found in languages. Again, the language requires some form of keyword support or standard library function to enable indexed parallel.

State ownership A key idea in concurrency is guarding of data races. Many languages do some through the use of immutable data, and others do so through strict encapsulation and ownership of data.

Process ownership In process calculi, a process is contained (owned) by its creating context. That is, if a process A launches a process B to execute concurrently with itself, process A cannot complete until process B has done likewise. Such behaviour is related to that of waiting for a parallel execution to complete, but can be generalised to include any spawned component in a concurrency language.

Process ownership is important when considering process mobility (higher-order processes). If a process is able to move to another context, the receiving process acquires ownership and should wait until the received process completes before it completes itself. The idea of process ownership also exists in the C++ threading standard, where by default a thread must be joined or moved to another thread to ensure it completes correctly.

Selection on incoming messages Another key feature of process calculi is choice on a set of events. For features this is split into selection on input and selection on output. This is because selection on both ends of a channel has been a difficult challenge. Separating the behaviour into two allows a better analysis of the selection feature. Keyword or standard library function is required for this feature.

Indexed selection Process calculi also provide an indexed selection operator. This is similar to the parallel for but is better described as *select from*. The operator is a short-cut for writing out each member of a select. The language must supply this in some manner that comes from the keywords or a simple method from the standard library.

Selection based on incoming value Another feature is the determination of behaviour based on the value communicated. Normally, this is used to work with arrays of channels (e.g. $c.5?x$) but in theory can be used as a general selection behaviour. The language must provide selection on incoming value at the point of selection, not after selection in an if statement or similar.

Guarded selection Conditional selection is provided in CSP with branches signified with if statements. The guard is a shorthand to indicate that for the given choice to be available the conditional must be met. The language must provide a mechanism to indicate when a given selection is allowed directly in the language.

Fair selection CSP and the π-Calculus do not specify which choice must be made if more than one is possible in a selection. In system verification, we are interested what would happen in either case, examining each possible trace in turn. In a real system, a decision must be made about how to resolve multiple ready conditions. Fairness means that no condition is prioritised each time the selection is made.

Selection with timer Although not an original feature of CSP or the π-Calculus, Timed CSP and the usefulness of timers is now an important feature. For a language to support timing it must allow some form of timeout or similar in a selection operation.

Other selection types This feature is added to explore which other selection types a language might support. The ability to follow a default branch is one such selection type, allowing a selection to be non-blocking. The language must provide this mechanism in the selection operation without some form of timeout being used.

Selection of outgoing messages Selection on output is also possible in process calculi, but is not commonly seen in implementation. The same requirements as input selection are necessary.

Multiparty synchronisation An event in CSP is a multi-party synchronisation, requiring all registered processes to engage in the communication. Typically, such behaviour is provided with a barrier or similar construct. The language must provide this construct in a similar manner to how channels are provided.

Selection on multiparty synchronisation As an event is multi-party and as an event can be selected, multiparty synchronisation means the ability to choose an event in a selection operation. The language must allow the a multiparty synchronisation to be used in such a block.

2.2. Benchmarks

Any benchmarking must provide the software engineer with information that is useful when selecting a language. The benchmarks considered here are aimed at message-passing concurrency features rather than computation power directly.

2.2.1. Communication Time

The Communication Time benchmark allows the measurement of communication / coordination time for the message-passing mechanism in the language. Its behaviour is simple, but its simplicity captures a number of important factors. Four components interact to output the natural numbers. The time taken to produce a single natural number is equivalent to four channel communications, thus allowing the calculation of the time taken to perform a single channel communication.

The Communication Time benchmark also servers two further purposes. Firstly, the ease of creating a communication ring in the language can be determined. Many message-passing languages rely on sending to a process via an identifier, which requires a component to know of its communication partner before initialisation. Such a dependency will make particular applications – such as the Communication Time benchmark – difficult to implement.

Secondly, the Communication Time benchmark enforces synchronization between the components. Some languages use asynchronous communication, meaning that measuring communication time is difficult. The enforced synchronization within the Communication Time benchmark will allow an actual communication / coordination time to be measured.

2.2.2. Select Time

Another common feature of message-passing languages is component behaviour being determined by the selection of incoming messages from a set of possibilities. The Select Time benchmark measures the time taken for a component to perform selection operation by sending to a single receiver from multiple senders.

This benchmark is not the traditional stressed alt benchmark as Erlang cannot support such a configuration. A process's mailbox will accommodate as many messages as possible until no memory is available. This means the Erlang mailbox is flooded, freezing the system for timing purposes and potentially hanging the machine. For a fair comparison of message selection, N writers are used to create a maximum number of processes no larger than that supported by the runtime.

2.3. Multi-core Support

With multi-core now being the norm, a concurrent language should utilise the hardware available as best as possible. A simple Monte Carlo π simulation allows testing speedup by increasing the number of workers available in the system. The Monte Carlo π benchmark will provide speedup values when using 2, 4, 8, and 16 processes.

2.4. Library Support

A somewhat subjective measure, but as indicated in the literature and important aspect when considering language choice. Library support is the least important aspect when considering just the message-passing features of a language, unless the library provides significant mechanisms to support the message-passing features. When it comes to determining which language to build a system with though, library support becomes very important.

3. Results and Discussion

In this section, we will examine the languages' process calculi features, benchmarking results, and try to discuss the library support features in an objective manner. Each language will be examined within each point, and any observations made commented on within the individual points.

3.1. Process Calculi Properties

Each feature will be examined in turn and the language support compared and summarised. The aim is to identify classifications and commonalities, as well as provide suggestions if certain features are lacking.

3.1.1. Synchronous Communication

Erlang Erlang is the only language examined which provides no synchronous communication as a basic feature. Although synchronous behaviour can be obtained via simple acknowledgement signals, by default Erlang does not support synchronous behaviour, adopting an asynchronous event model.

Go Go provides synchronous buffered channels of size 0 as default. To create a channel with a buffer requires a size parameter to be passed to the channel creation operation. Listing 1 illustrates channel creation code for Go.

```
// Normal channel creation
var a := make(chan int)
// Buffered creation
var c := make(chan int, n)
```

Listing 1: Creating a channel of buffer size `n` using Go.

Rust Rust's default channel type is asynchronous, but a synchronous channel is provided which can have a buffer size set during creation: a buffer size of 0 is a rendezvous channel. Listing 2 illustrates channel creation code for Rust.

```
// Create "infinitely buffered" channel.
let (tx, rx) = channel();
// Create a synchronous channel of buffer size n
let (tx, rx) = sync_channel(n);
```

Listing 2: Creating a channel in Rust.

occam-π occam-π operates on a purely synchronous channel model. No buffering is possible without a process in-between.

Summary Synchronous communication is the default in both Go and occam-π. Rust also supports synchronous communication but not in its default channels. Only Erlang provides no out-of-the-box synchronous behaviour. It is possible to implement using acknowledgement messages. Therefore, synchronisation can be seen as a common behaviour across the languages examined.

3.1.2. First-order Channels

Erlang As mentioned, Erlang uses a mailbox model to manage messages. As such, messages are sent directly to processes based on the process's ID. Erlang therefore has no channels.

Go A channel type is provided in Go, and the channel can be typed. Channels can be assigned to variables. The ends of the channel (writing and reading) can be captured if required.

Rust The channel type in Rust is explicitly assignable to writing end and reading end variables; `tx` and `rx` in Listing 2. `tx` has type `Sender` or `SyncSender`, and `rx` has type `Receiver`. Both of these end types can be typed.

occam-π occam-π has the ability to create typed channels, and the ability to capture the writing and reading ends of the channel. All of these channels can be assignable to a variable.

Summary As with synchronisation, all but Erlang provide a channel mechanism. The lack of channels in Erlang makes system setup more difficult as processes must know of each other to send messages. Order of process creation is therefore important. If any cyclical structures are in place, communication of process IDs is required at the start of an Erlang application. The channels in Go, Rust, and occam-π are similar with concepts of input and output ends in place. We can define Go, Rust, and occam-π collectively as channel-based languages.

3.1.3. Higher-order Channels

Erlang As Erlang has no first-order channels, it has no higher-order channels.

Go Channels in Go can be typed to communicate other channels. Listing 3 illustrates how this is achieved in Go.

```
// Create a channel to send channels.
var a := make(chan chan int);
// Create a normal channel and send it via the other channel.
var b := make(chan int);
a <- b;
```

Listing 3: Creating and using a higher-order channel in Go.

Rust As Go, Rust can also declare channels of types to send channel ends - either `Sender`, `SyncSender`, or `Receiver`. Listing 4 provides an example.

```
// Create two channels.
let (a_tx, a_rx) = channel();
let (b_tx, b_rx) = channel();
// Send one end of b down a.
a_tx.send(b_tx).unwrap();
```

Listing 4: Creating and using a higher-order channel in Rust.

occam-π Mobile records enable communication of channel end types in occam-π. Listing 5 provides an example code listing for such a channel communication.

```
CHAN TYPE INT.IO
  MOBILE RECORD
    CHAN INT in?:
:

PROC node(CHAN INT.IO! sent.int)
  INT.IO! int.c:
  INT.IO? int.s:
  SEQ
    int.c, int.s := MOBILE INT.IO
    send.int ! int.c
    ...
```

Listing 5: Creating and using a higher-order channel in occam-π.

Summary Each of the channel-based languages also support higher-order channels. Only in occam-π can it be strongly argued that the addition of mobility was a concious decision. Go and Rust have channels that can send any type, and a channel is just a type in these languages.

3.1.4. First-order Processes

Erlang The lack of first-order channels in Erlang is replaced by first-class processes. When a process is created, using the `spawn` function, the ID of the created process is returned. This allows messages to be sent to the process.

```
Pid = spawn(fun() -> io:format("Hello World~n") end).
```

Listing 6: Spawning a process in Erlang.

Go Goroutines are initiated using the Go command as shown in Listing 7. Goroutines are non-assignable to variables and are therefore not first class.

```
go func() {
    fmt.Println("Hello World!")
}
```

Listing 7: Starting a Goroutine.

Rust Although threads are created in Rust using the `thread::spawn` command (as illustrated in Listing 8), a `thread` object is not accessible. Rather, a call to `thread::spawn` returns a `JoinHandle` object, which allows the parent thread to join (wait) for the created child thread. Although Erlang provides a richer first-class process type, it is argued that Rust's thread creation mechanism does allow assignment of an value representative of the created thread, and therefore Rust provides first-class processes.

```
let child = thread::spawn(move || { println!("Hello World!"); });
```

Listing 8: Starting a thread in Rust.

occam-π Although occam-π has mobile process types as an extension feature, which allows assignment of processes to variables, this feature does not support standard process types. Mobile processes have some restrictions on their definition, and, as such, occam-π only provides partial support for first-order processes. More discussion is provided in Section 3.1.5 on higher-order processes.

Summary First-order processes are not a common feature. Only Erlang has the ability to work with assignable process variables. The first-class processes in Erlang are a replacement for channels allowing communication. It can be argued that a first-class process does not add any functionality. Sending signals is the best method of working with a process and having a variable of type process does not add anything that a channel does not.

3.1.5. Higher-order Processes

Erlang Erlang supports the communication of any type to a process's mailbox, including process IDs. In fact, this is a common mechanism to allow communication networks to be set up. Listing 9 provides a simple example.

```
A = spawn(...),
B = spawn(...),
A ! B.
```

Listing 9: Sending a PID in Erlang.

Go As Go does not support first-order processes it does not support higher-order processes.

Rust As Rust can communicate any type via its channel it likewise can send `JoinHandle` types. Therefore, Rust has higher-order process support.

occam-π As occam-π does have a `MOBILE PROC` type it can be argued that higher-order processes are supported. However, these processes are really continuations with check-pointing. Consider Listing 10.

```
PROC TYPE mobile.proc(CHAN INT in?):

MOBILE PROC my.proc(CHAN INT in?)
  IMPLEMENTS mobile.proc
  SEQ
    -- Do some work
    SUSPEND
    -- Process is checkpointed here.
    -- Restarting the process allows continuation

    -- Do some other work
```

```
:
```

Listing 10: `MOBILE PROC` Example in occam-π.

A higher-process in occam-π can be sent via a channel when it is suspended (with the `SUSPEND`) keyword. The process is check-pointed at this stage. When re-invoked, the process will continue at the suspension point. Process parameters are limited to channels or barriers, limiting the type of mobile processes that can be designed.

Summary Erlang supports higher-order processes much like Go, Rust, and occam-π support higher-order channels. This feature is actually required given the lack of channels in Erlang as it allows the creation of communication cycles. occam-π does have `MOBILE PROC` types that provide a different approach to higher-order processes, but is arguably a less versatile solution.

3.1.6. Parallel Execution Statement

Erlang Erlang only allows the spawning of individual processes. A helper function could be implemented to take a list of parameters to execute a collection of processes, but Erlang does not provide this explicitly.

Go Goroutines are started individually, and therefore Go does not provide a explicit parallel execution statement. Again, a helper function could be developed if needed.

Rust Rust, as Erlang and Go, only allows creation of a single thread at a time. Helper function implementation is possible.

occam-π The only language examined that provides a parallel execution statement is occam-π. Listing 11 provides an example.

```
PROC system()
  PAR
    proc.1()
    proc.2()
    ...
:
```

Listing 11: PAR execution in occam-π.

Summary Only occam-π provides a parallel execution statement. Erlang, Go, and Rust do not use a process as a synchronisation and all execution is detached from the initiating process. In occam-$/pi$ a parallel execution is another control block that needs to complete to allow continuation. However, Go and Rust have to use other primitives to simulate such behaviour if required.

3.1.7. Indexed Parallel Execution Statement

Erlang As Erlang provides no parallel execution statement it likewise provides no indexed parallel execution statement. A helper function could be developed to support such behaviour.

Go As Go has no parallel execution statement it has no indexed parallel execution statement. A `for` loop in Go can provide much of the behaviour required simply as illustrated in Listing 12. However, explicit indexed parallel is not supported.

```
for i := 0; i <= N; i++ {
    go my_func(i)
}
```

Listing 12: Using a `for` loop in Go with Gorountines.

Rust As with Go, indexed parallel execution can be simulated using a `for` loop. However, no explicit language support is provided.

occam-π occam-π supports indexed parallel execution via the replicated `PAR` construct. An example is provided in Listing 13.

```
PAR i = 0 FOR N
  my_func(i)
```

Listing 13: Replicated PAR in occam-π.

Summary Although occam-π provides specific syntax for indexed parallel execution, it is not difficult behaviour to replicate in the other languages. It is easy to see why languages do not implement additional syntax if it can be replicated. As such, it can be argued that explicit syntax is not necessary.

3.1.8. State Ownership

Erlang As Erlang is a functional programming language values are not referenced between contexts. As such, Erlang has state ownership because everything is copied.

Go Go has no explicit state ownership mechanisms, and is the only language examined that does not support such a feature. A pointer can be freely transferred between contexts by the programmer.

Rust Ownership of resources is an important factor in Rust, and although references are allowed, only one variable owns a given resource. References in Rust use borrow semantics, and this is enforced at compile time.

occam-π Originally occam only supported copy semantics. The introduction of occam-π allowed data to be *moved* if it is declared as `MOBILE`. The compiler checks that data is not referenced to avoid shared ownership and the possible data race that can occur.

Summary State ownership is a common feature, but it is Rust and occam-π that have explicit features to support correct behaviour. Erlang has state ownership via its functional nature. Rust and occam-π require additional care by the programmer to get the ownership behaviour correct, with the compiler ensuring that this happens.

3.1.9. Process Ownership

Erlang Although Erlang supports state ownership due to its functional nature, and supports first-order processes, it does not support process ownership. A returned process ID from a spawn is a handle to allow communication with the created process, and multiple processes may use this “address”. When a process is spawned, it runs separately to the creating process and is not contained within the spawning process’s state.

Go As Go does not support state ownership, and has no method of assigning goroutines to variables, there is no explicit method to maintain process ownership. There is also no implicit mechanism and a spawned goroutine is not considered part of the executing state of the creating process.

Rust Although Rust does support state ownership, and that a `JoinHandle` would likewise be owned by a particular variable, it is not the case that a thread must be joined before the handle goes out of scope. A Rust thread does not contribute to the state of the creating Rust thread. However, if the main thread exits before all spawned threads complete the Rust application will exit with an error. Waiting on the `JoinHandles` negates the error. So, although Rust has a number of features that would allow process ownership, they are not combined together to provide this functionality.

occam-π occam-π does not allow processes to be assigned to a variable, but any processes created in a `PAR` block is "owned" by that block insofar as the `PAR` does not complete until all child processes have. However, occam-π does allow processes to be spawned without a surrounding `PAR` block using the `FORK` keyword. An ownership statement can be made using a `FORKING` statement, so process ownership is maintained within occam-π.

Summary Process ownership is another feature that is not common, with occam-π being the only language to enforce such behaviour. Rust has many of the features and requirements that would indicate a form of process ownership, but these are detached from each other. It would be useful for Rust to enforce ownership via the compiler to avoid the exit error. Go and Erlang have a detached process model, with goroutines providing no handle thus being completely detached. We can grade process ownership features from strongest to weakest as: occam-π, Rust, Erlang, Go.

3.1.10. Selection on Incoming Messages

Erlang As Erlang has no channel construct it cannot choose between incoming messages. The mailbox contains all messages awaiting processing, and an Erlang process cannot determine its behaviour based on where a message comes from except using the value of the message as will be described below.

Go Selection between available channels is simple in Go. Selection is achieved using the `select` statement, which acts much like a case statement in C-like languages. Listing 14 provides an example.

```
func do_work(in0 <-chan int, in1 <-chan int) {
    select {
        case msg := <-in0 : // do some work
        case msg := <-in1 : // do some work
    }
}
```

Listing 14: Selecting on incoming channels in Go.

Rust Selection in Rust is currently in an experimental state, although it used to be in the standard library. The functionality is provided, but it is considered not necessarily stable. As such, a nightly build of Rust is required to perform selection. Listing 15 provides an example of the experimental select behaviour in Rust.

```
fn do_work(in0 : Receiver<i32>, in1 : Receiver<i32>) {
    select! {
        msg = in0.recv() => // do some work
        msg = in1.recv() => // do some work
    }
}
```

Listing 15: Selecting on incoming channels in Rust.

The `select!` macro utilises Rust's `Select` type. The `Select` provides methods to add a handle and to wait for one of the events. As such, it is more akin to the alternative techniques seen in JCSP.

The currently documented method to simulate the select is to use a busy system. While in a loop, each channel is tried to see if a receive is possible. This will have an overhead in comparison to a blocking select.

occam-π `ALT` allows selection from a set of incoming channels. An example is provided in Listing 16.

```
PROC do.work(CHAN INT in.0?, in.1)
  INT x:
  ALT
    in.0 ? x
    -- do some work
    in.1 ? x
    -- do some work
:
```

Listing 16: Selecting on incoming channels in occam.

Summary Each of the channel-based languages provides selection, with Rust being the only one to see this as an experimental feature. The syntax structure is also common between the languages, with the event being executed and then behaviour defined based on the event selected.

3.1.11. Indexed Selection

Erlang As Erlang has no channel or selection from channels, there is no capability for an indexed selection. The mailbox contains all messages, but it cannot be indexed across.

Go Go does allow a form of indexed selection behaviour via a for loop. Listing 17 provides an example.

```
func do_work(in []chan int) {
    for i := range in {
        select {
            case x := <- in[i] : // do some work
            default:
        }
    }
}
```

Listing 17: Indexed selection in Go.

The syntax is clunky, but it does provide indexed selection. The use of `default` essentially allows a fall-through which is not ideal. What we have is a busy select, and a surrounding loop would ensure an option is picked. Go can also support indexed selection via reflection, but would require some construction of the select object and also have an overhead due to reflection. The indexed select is therefore partially supported.

Rust Rust does not support indexed selection, although it can be replicated by building a `Select` structure one handle at a time. This makes the syntax similar to that of JCSP, and therefore is not considered a supported feature.

occam-π The ALT process can have FOR applied to it as SEQ and PAR. Listing 18 provides an example.

```
PROC do.work([N]CHAN INT in?)
  INT x:
  ALT i = 0 FOR N
    in[i] ? x
    -- do work
:
```

Listing 18: Indexed selection in occam-π.

Summary Of the channel-based languages, both Go and occam-π provide indexed selection, although Go's support is partial. Both are based on replication with a for. It can be argued that occam-π's syntax is cleaner and matches that of SEQ and PAR. Go code looks less elegant and is outcome of default selection types and looping, making the selection busy. Given the usefulness of an array or vector of channels, channel-based languages should consider this feature as important.

3.1.12. Selection Based on Incoming Value

Erlang The lack of selection on messages is replaced by selection on incoming values in Erlang. Listing 19 provides an example. The incoming values that are selectable on are limited to user defined atoms, and the selection is more a form of pattern matching than a choice.

```
do_work() ->
    receive
        a -> %% do some work,
        {b, X} -> %% do some work
    end.
```

Listing 19: Selecting on incoming values in Erlang.

Go Go cannot select based on incoming value. The first value available in any channel is the one that is read.

Rust As Go, rust cannot select based on incoming value. The first value available in any channel is the one that is read.

occam-π As the other channel-based languages, occam-π does not support selection based on value.

Summary Selection based on incoming value is used by Erlang to overcome the lack of channels with selection. The channel-based languages do not support selection based on value. These languages will use case statements of if statements to allow such behaviour, and so it is arguable whether it is necessary.

3.1.13. Guarded Selection

Erlang Erlang does allow guarded behaviour on selected incoming values. Listing 20 provides an example.

```
guarded() ->
    receive
        N when N > 42 -> do_work_a();
        N when N < 12 -> do_work_b();
        N -> do_work_c()
    end.
```

Listing 20: Guarded selection in Erlang.

Guarded selection is a form of pattern matching in Erlang. Each entry in the mailbox is tested against the patterns to determine which branch to take. If no pattern is matched, the next item in the mailbox is examined.

Go There is no support for guarded selection in Go, although the behaviour can be simulated by building a custom structure. As such, Go is not considered to support guarded selection.

Rust Rust has no mechanisms to support guarded selection. The functionality could be provided via selective building of the `Select` structure, but this would be required at each select point.

occam-π Guards can be placed directly into a `ALT` case. Listing 21 provides an example where a logical condition is used to determine which channels can be read from.

```
PROC guarded(INT n, CHAN INT in?)
  INT x:
  ALT
    n > 42 && in ? x
    do_work_a()
    n < 12 && in ? x
    do_work_b()
    in ? x
    do_work_c()
:
```

Listing 21: Guarded selection in occam-π.

occam-π's guards are separate to the channels. The logical guard is a condition that can be tested when the `ALT` is executed, thus leading certain branches to be inactive. This behaviour is different to the pattern matching approach in Erlang.

Summary Guarded selection is only available in Erlang and occam-π. In Erlang, the feature comes from pattern matching on the incoming value or other values that are known to the process. In occam-π guards are discrete logical statements separate from the incoming value of a channel, which would be dealt with via an if statement instead.

3.1.14. Fair Selection

Erlang A process receives messages in the order they are sent to the mailbox. This behaviour is not fair, as the asynchronous scheduling of processes means that any process can send numerous messages which will be processes first.

Go The `select` is fair in that when multiple possible channels are ready the chosen branch is selected randomly. This is not quite a priority fairness where the last selected branch is given lower priority, but is a fair solution.

Rust There is no indication on if the Rust `Select` is fair. Examining the source code indicates that there is no specific fairness mechanism implemented, meaning that the `Select` is not fair.

occam-π By default an `ALT` in occam-π is fair. occam-π also provides an unfair `PRI ALT` where branches are evaluated in order. Listing 22 provides an example.

```
PROC do.work(CHAN INT in.0?, in.1)
  INT x:
  PRI ALT
    in.0 ? x
    -- do some work
    in.1 ? x
    -- do some work
:
```

Listing 22: `PRI ALT` in occam-π.

Summary The fairness of selection is provided in both Go and occam-π. Erlang and Rust provide no such mechanism. In occam-π the fairness can be switched off by using a `PRI ALT`, whereas Go has no such facility. occam-π is therefore the only language that provides both options.

3.1.15. Selection with Timer

Erlang The `after` option in a `receive` statement to have a timeout. Listing 23 illustrates.

```
do_work(Timeout_ms) ->
    receive
        N -> do_more_work()
        after
        Timeout_ms -> do_less_work()
    end.
```

Listing 23: Erlang receive with timeout.

Go A timeout is available via the `time` package in Go. Multiple timeouts are possible in a `select` block, which is more versatile than the single approach in Erlang. Listing 24 is an example of Go's timeout facility.

```
func do_work(in chan int, timeout_ms int) {
    select {
        case n := <-in : do_more_work()
        case <-time.After(time.Millisecond * timeout_ms) : do_less_work()
    }
}
```

Listing 24: Go select with timeout.

Rust Prior to selection becoming an experimental feature, timeouts where available. The new experimental `Select` does not support a timer at present.

occam-π The `TIMER` enables timeouts in an `ALT`. As with Go, multiple `TIMER`s can be created, but the granularity of the timer is only milliseconds. Listing 25 provides an example.

```
PROC do.work(CHAN INT in?, INT timeout_ms)
  INT n, t:
  TIMER tim:
  SEQ
    tim ? t
    t := t + timeout_ms
    ALT
    in ? n
```

```
      do.more.work()
      tim ? AFTER t
      do.less.work()
  :
```

Listing 25: occam-π ALT with timeout.

Summary Timing is another common feature in message-passing languages. Rust is the only language without support, although it was previously supported and therefore likely to return. Go arguably provides the richest functionality, with different time granularity available. occam-π and Go support multiple timers, which Erlang does not. Depending on the application, the lack of multiple timers can be a weakness.

3.1.16. Other Selection Types

Erlang Erlang has no default branch in a `receive`. If no message is in the mailbox, and no timeout is set, the process has no mechanism to break out of a `receive` statement.

Go As Go's `select` is similar to a case statement in other languages, the `default` keyword has been used as another selection type. Listing 26 illustrates the syntax.

```
func do_work(in chan int) {
    select {
        case n := <-in : do_some_work()
        default : do_nothing()
    }
}
```

Listing 26: Default selection behaviour in Go.

Rust Rust at present only supports channels in a `select!`.

occam-π The `SKIP` (empty process) guard is used for a default branch in an `ALT`. Listing 27 shows how `SKIP` is used.

```
PROC do.work(CHAN INT in?)
  INT n:
  ALT
    in ? n
    do.some.work()
    SKIP
    do.nothing()
:
```

Listing 27: `SKIP` selection in occam-π.

Summary Both Go and occam-π provide a default or fall-through option in a selection operation. Rust and Erlang only provide a committed selection with no fall-through. Although Erlang has a "don't care" receive, this is still a committed input and will remove an item from the mailbox. Go and occam-π's non-committed selection provides more flexibility which should be considered, particularly in channel-based Rust.

3.1.17. Selection of Outgoing Messages

Erlang As Erlang messages are asynchronous there is no outgoing selection. A message is sent immediately to the receiving process's mailbox.

Go An output operation can be used within a `select`. Listing 28 provides an example.

```
func do_work(in <-chan int, out ->chan int, n int) {
    select {
        case n <- in : do_input_work()
        case out <- n : do_output_work()
    }
}
```

Listing 28: Selecting an output in Go.

Rust Rust does not support output selection.

occam-π occam-π does not support output selection.

Summary Only Go supports selection on an output. Considering the difficulty such behaviour has had in other libraries (for example JCSP), further investigation is required to see if the Go output selection works in all conditions. Output selection is useful, and if other languages can learn from Go's implementation then it is a feature that should be implemented.

3.1.18. Multi-party Synchronisation

Erlang Erlang provides no synchronisation between processes. At best, a send-ack pair would allow such behaviour. It can argued that Erlang is effectively an asynchronous language.

Go The `WaitGroup` provides multi-party synchronisation. An example is given in Listing 29.

```
func worker(out chan int, wg WaitGroup) {
    // Do some work
    out <- result
    wg.Done()
}
```

Listing 29: Using a `WaitGroup` in Go.

Typically the `WaitGroup` is used to synchronise completion of a goroutine. It provides a mechanism to wait for a set of goroutines to finish, much like a `PAR` statement does implicitly in occam-π.

Rust A `Barrier` type is provided by Rust. Listing 30 provides an example.

```
fn worker(out : Sender<i32>, bar : Barrier) {
    // Do some work
    out.send(result).unwrap();
    bar.wait()
}
```

Listing 30: Using a `Barrier` in Rust.

occam-π A `BARRIER` is also provided by occam-π. Listing 31 provides an example.

```
PROC worker(CHAN INT out!, BARRIER bar)
  INT result:
  SEQ
    -- Do some work
    out ! result
    SYNC bar
:
```

Listing 31: Using a BARRIER in occam-π.

Summary Only Erlang provides no group synchronisation mechanism. Go, Rust, and occam-π each provide a similar mechanism which can be described as a barrier. The operations possible on these barriers are almost identical, with waiting and resetting functionality provided. This is another common feature in the channel-based languages.

3.1.19. Selection on Multi-party Synchronisation

Erlang As Erlang has no multi-party synchronisation there is no selection possible.

Go A `WaitGroup` cannot be used in a `select` operation.

Rust A `Barrier` cannot be used in a `select!` operation.

occam-π A `BARRIER` cannot be used in a `ALT` operation.

Summary No language currently supports selection on multi-party synchronisation. It is possible to provide such behaviour (see JCSP), and perhaps it is a feature these languages should consider in the future.

3.1.20. Summary and Discussion

Table 1 summarises the feature list of Erlang, Go, Rust, and occam-π., and it is unclear if such code was an intentional design decision

Property	**Erlang**	**Go**	**Rust**	**occam-π**
Synchronous communication		✓	✓	✓
First-order channels		✓	✓	✓
Higher-order channels		✓	✓	✓
First-order processes	✓		✓	(partial) ✓
Higher-order processes	✓		✓	(partial) ✓
Parallel execution statement				✓
Indexed parallel execution statement				✓
State ownership	✓		✓	✓
Process ownership				✓
Selection on incoming messages		✓	(experimental) ✓	✓
Indexed selection		(partial) ✓		✓
Selection based on incoming value	✓			
Guarded selection	✓			✓
Fair selection		✓		✓
Selection with timer	✓	✓		✓
Other selection types		✓		✓
Selection of outgoing messages		✓		
Multi-party synchronization		✓	✓	✓
Selection on multi-party synchronization				

Table 1. Checklist of process-calculi features supported in Erlang, Go, Rust, and occam-π.

The classification of features indicates that two approaches to message-passing concurrency are present. The first is a mailbox system as seen in Erlang. A mailbox system leads to a collection of features which are different to the other three languages examined. The lack of a channel abstraction, and the subsequent features not provided, is the key difference between Erlang the the other languages examined.

occam-π has the most process calculi features, which is expected due to occam and CSP having strong links. occam-π is the only language that supports process ownership, although only partially. Guarded selection and parallel execution statements are also only provided in occam-π.

Go shares many of its features with occam-π, with only output selection being a unique feature. Go is the only language not to support state ownership, which arguably leads to potential data race problems.

Rust is the newest language examined, and it has some occam-π features than Go does not. It is the only channel based language to not fully support input selection. Such an exclusion is a possible weakness, as Erlang provides value selection from a process's mailbox.

Mobility is a common feature supported by all languages. Erlang and Rust both support higher-order processes by communicating a form of handle to a process or thread. For Erlang, the handle allows communication with the process via messages. With Rust, the handle allows the thread to be joined, so is not strictly a higher-order process. Each channel-based language supports higher-order channels. With Rust and Go this is an outcome of having channels typeable with anything supported by the language, which includes channels and/or channel ends. With occam-π, explicit work was undertaken to add higher-order channel capabilities. In either case, the ability to reconfigure communication infrastructure is the important outcome.

An aim of this work is an attempt to start classifying message-passing concurrency languages. Erlang does not have a similar feature list to thee other three languages, mainly due to the lack of a channel abstraction. Therefore, trying to synthesise a single set of common features is not practical. As such, we can separate message-passing concurrency languages into two criteria:

actor-based languages Erlang.
channel-based languages Go, Rust, occam-π.

An actor-based language uses a mailbox system to communicate between processes. Each process has a mailbox and the process checks the mailbox to receive incoming messages. Channel-based languages allow multiple inputs and outputs from a process, and decouple processes from one another because of this.

We can therefore define an initial hierarchy of message-passing concurrency language types as illustrated in Figure 3.1.20. The hierarchy is limited as only four languages have been examined, but it does enable some initial language classification.

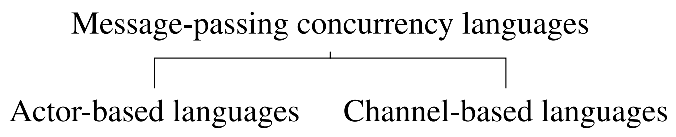

Figure 1. An initial hierarchy of message-passing concurrency languages.

Having such an initial classification does lead to questions on how to further evaluate message-passing concurrency languages. Actor-based approaches are fundamentally different to channel-based approaches, due primarily to the lack of a channel construct meaning that actor-based languages only have one stream of inputs.

It can also be argued that channel-based languages come in strict and non-strict forms. occam-π is a strict, channel-based message-passing language. It is strict due to it's adherence to state and process ownership. Rust is partially strict because of state ownership, where Go is a non-strict language.

We can also define some common features supported in channel-based languages:

1. Synchronous communication.
2. First-order channels.
3. Higher-order channels.
4. Selection on incoming messages.
5. Multi-party synchronisation.

Of course, other channel-based languages will need to be examined to determine if this is a common feature-set.

3.2. Benchmark Results

Each benchmark application is run on an Intel Core i7-4770K CPU running at 3.50GHz on a Linux OS with kernel 4.10. The following versions of languages were used:

- Erlang 18.
- Go 1.8.3.
- Rust 1.21 Nightly (to provide selection support).
- KRoC 1.6 for occam-π.

3.2.1. Communication Time

3.2.2. Selection Time

3.2.3. Multi-core Support

3.2.4. Summary

3.3. Library Support

Library support analysis is somewhat subjective, but some observations can be made: general library support, external library support, and package manager support.

3.3.1. General Library Support

Erlang Erlang is a commercial language initially developed by Ericsson, and is the second oldest language examined being launched in 1986. Its library reflects Erlang's usage as telephonic system language, and a language designed for large asynchronous messaging. The library has modules that support modern concepts in databases and networking, with a good deal of documentation.

Go Go is also a commercial language, developed by Google and initially launched in 2009. The Go standard libraries are as expected in a modern, commercially supported language, with support for cryptography and string handling as provided examples. Go is used in a number of Google's production systems, and this shows in the networking and service support.

Rust Rust is commercial, supported by Mozilla and appearing in 2010. Rust is aimed at system's programming, and therefore has support modules for working with memory and other low-level concerns. The library is also the smallest in comparison to the other commercial languages, and looks more like the C standard library with some additions such as collections and networking. The functional nature of the language also provides additions such as zip. Rust is still experimental in some areas, leading to a conclusion that the library is still not stable and may change in the future.

occam-π occam-π is a research language rather than a commercial language, and therefore its general library support does not compare to Erlang, Go, and Rust. It is also difficult to determine what could be considered the standard library, and what is additional, such as OpenGL support. occam is the oldest language examined, being first introduced in 1983.

Summary This measure is the most subjective of those examined, as library requirements change based on need. A simple measure is that Erlang, Go, and Rust are commercially supported, whereas occam-π is not. Therefore, general library support is better in Erlang, Go and Rust. Of these three, Rust has the smallest library, with some experimental features still present. Erlang and Go are difficult to differentiate between, and provide suitable support for their target audiences.

3.3.2. External Library Support

External library support is where the language or runtime provides mechanisms to import library code form other languages, such as C. It is not expected that the language does this automatically, and some work may be required by the programmer to interface. It is simply the ability to interface with external library code that is important.

Erlang Native Implemented Functions (NIFs) allow Erlang to call external C code. A wrapper interface is required, but it is possible to return basic types from a C library to Erlang.

Go `cgo` allows Go to interface with C. It is also possible to export Go functions to be used by C. `cgo` also supports pointers to a certain extent, which Erlang does not.

Rust C code can be called directly in Rust. The `unsafe` keyword allows C functions to be used directly if required. As a system's programming language, Rust can also replace C as the language being called by others if a suitable library is produced.

occam-π The C Interface (CIF) allows occam-π to call C functions. Wrapper code is required, but as with Erlang and Go the majority of functionality is possible with basic types supported.

Summary Each language provides interfacing with C. Rust takes this a step further with C code usable directly in Rust code. It is difficult to differentiate between Erlang, Go, and occam-π in this regard.

3.3.3. Package Manager Support

Modern languages come with package manager systems to allow simple installation of user-supplied code. The package manager normally provides build facilities also. Examples include Python's pip.

Erlang Package managers exist for Erlang, Hex being the common one. There is no official package manager, and therefore one has to be installed separately.

Go `go get` downloads and installs packages and comes with the Go installation. `dep` is an upcoming dependency management tool that is a pre-alpha phase. With both of these tools, Go will have a complete package manager experience provided with the official implementation.

Rust `crate` is the official package manager for Rust. Dependencies can be defined in build files and fetched and compiled as required. Rust code can define external crate requirements also. Rust therefore has a fully integrated package manager.

occam-π There is no package management system available for occam-π.

Summary Package management is fully functional in Rust, and almost fully functional in Go. Erlang has package managers available but nothing official. occam-π has no such support. Go and Rust are the newest languages evaluated and therefore it makes sense that they have such support. occam-π is non-commercial and therefore package management is not provided. It could be argued that occam-π provides everything when downloaded, and therefore no package management is needed.

3.3.4. Summary and Discussion

It is obvious that occam-π does not fair well in this area primarily due to having no commercial support. Rust and Go fair well as they are both newer languages with commercial support. Go probably has the best library support overall as it has a reasonable package manager and a large general library. Rust has the better package manager, but its system's programming aim means its library is understandably smaller. Erlang also has a reasonably sized library. Each language also provides C interfacing.

Erlang has library features that make it useful for telephonic systems, messaging applications, and other Internet applications. The library is reasonably broad, and package managers are available. It can be argued that Erlang provides a good level of library support, but in comparison to Go it is a lacking.

Go provides a rich library, has a versatile package manager, and is commercially supported. If library support is key, then Go is arguably the best choice.

Rust's simplicity makes it similar to C. The language is small, the standard library is small, but with modern features. The additional package manager is useful. Rust is a good C replacement if that is required.

occam-π is a research language, and as such it cannot be compared to the other three. C interfacing is provided, and a fairly extensive library of code is provided. Really, the only lacking element is a package manager.

If we consider the aim of this paper where message-passing and the related elements are important, then these library features are of little importance. None of the languages extend the message-passing features with their libraries. Determining where a language is useful is the important information from this section.

4. Conclusions

A framework for evaluating message-passing concurrent languages has been proposed, and this framework applied to four languages: Erlang, Go, Rust, and occam-π. The framework has allowed some distinction between languages to be made, but more work is required to fully explore what a software engineer may find useful. The evaluation has led to some observations that are summarised in the following sections.

4.1. Erlang

Erlang is an asynchronous, actor-based language, providing message-passing via a a mailbox system. The language can be deemed suitably mature, with good library support. An internal scheduler means only small processes are required, and therefore no limit based on operating system thread load. One point of note is the mailbox being susceptible to "denial-of-service" attacks via flooding. The functional nature of the language leads to state protection. Erlang also has the fewest process calculi features, but this is mainly due to no channel being available.

4.2. Go

Go has a good library, and could be argued as the language with most commercial features. It has good library support, and a package manager to further support development. It also has a good range of process calculi features, with only occam-π having more. The performance of the language is also comparable to occam-π. The internal scheduler also leads to small, lightweight processes. The limitations come from the support features from process calculi. A lack of a parallel, indexed parallel, and a usable indexed selection make some applications difficult to write.

4.3. Rust

Rust is a small system's language, which has features more akin to C than the other languages examined. The channel and thread mechanism are based on operating system facilities rather than an internal scheduler. This makes Rust the slowest language from a communication benchmarking point-of-view, and limits the active number of components. Rust has a reasonable set of process calculi features, but the lack of a non-experimental select is a limitation. The package manager and state protection mechanisms are useful, although the latter does mean more thought being put into the code that is written.

4.4. occam-π

occam-π is a research language, which puts it at a disadvantage in a number of areas in comparison to the other languages evaluated. In particular, library support and package management is limiting. However, the occam-π is the most adherent to process calculi features and has the best communication performance.

4.5. Language Use

From these observations an initial classification of usage can be defined:

Erlang is useful for asynchronous systems with numerous processes communicating in a dynamic manner.
Go is useful for developing commercial applications with lightweight message-passing facilities.
Rust is a system's programming language with better protection than C, that supports concurrency with a channel construct.
occam-π is a research language useful for exploring process calculi ideas in given contexts.

These are broad definitions, and each language can be used in a number of different scenarios. The classification is meant to distinguish the most suitable usage of each language, rather than stating that it is their only purpose.

4.6. Future Work

Only four languages have been examined thus far to gain an initial analysis of the approach. Approximately 50 languages have been identified which allude to message-passing concurrency support, although it is likely that some of these may do so via libraries or have a different interpretation of message-passing. Some key languages that will be examined next are:

- Ada
- D
- Ocaml
- Scala

Many of the other languages identified seem to have small user-bases, and may be simple research languages. Ada, D, Ocaml, and Scala are popular, although only Ada is really know as having message-passing capabilities.

References

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

[2] S. T. Redwine and W. E Riddle. Software Technology Maturation. In *Proceedings of the 8th International Conference on Software Engineering ICSE '85*, pages 189–200, 1985.

[3] R. Milner. A Calculus of Communicating Systems, 1986.

[4] K. R. Parker and B. Davey. The History of Computer Language Selection. In *IFIP Advances in Information and Communication Technology*, volume 387, pages 166–179. Springer Berlin Heidelberg, 2012.

[5] A. L Tharp. Selecting the Right Programming Language. *ACM SIGCSE Bulletin*, 14(1):151–155, 1982.

[6] M. L. Nelson. Considerations in Choosing a Concurrent/Distributed Object-oriented Programming Language. *ACM SIGPLAN Notices*, 29(12):66–71, 1994.

[7] J. M. Yohe. An Overview of Programming Practices. *ACM Computing Surveys*, 6(4):221–245, dec 1974.

[8] J. AlGhamdi and J. Urban. Comparing and Assessing Programming Languages : Basis for A Qualitative Methodology. In *SAC '93 Proceedings of the 1993 ACM/SIGAPP symposium on Applied computing: states of the art and practice.*, pages 222–229, New York, New York, USA, 1993. ACM Press.

[9] B. Belkhouche, L. Lawrence, and M. Thadani. A Methodical Comparison of Ada and Modula-2. *Journal of Pascal, Ada & Modula-2*, 7(4):13–24, 1988.