Discourje: An embedded protocol language.

Reducing multi-threading complexity through an embedded protocol language as Clojure macros.

Ruben Hamers

Student: 851851876
Date: 08/07/2019

DISCOURJE: AN EMBEDDED PROTOCOL LANGUAGE.

REDUCING MULTI-THREADING COMPLEXITY THROUGH AN EMBEDDED PROTOCOL LANGUAGE AS CLOJURE MACROS.

Thesis

by

Ruben Hamers

in fulfillment of the requirements for the degree of

Master of Science
in Software Engineering

at the Open University, faculty of Management, Science and Technology
Master Software Engineering.

Student number:	851851876	
Course code:	IMA0002	
Thesis committee:	Prof. dr. M. C. J. D. van Eekelen (chairman),	Open University
	Dr. S. Jongmans (supervisor),	Open University

0.1. Abstract

Writing correct concurrent software programs is becoming increasingly difficult in traditional programming languages like C and Java, as a result of the increasing number of CPU cores on chips. One way of simplifying concurrent programming is to use domain-specific languages for interaction protocols, which offer high level abstractions for low level synchronization concepts and separation of concerns for coordination and business logic. A considerable downside to protocol languages, however, is that usability is hampered by a number of issues: they require special syntax, they may be coupled to a specific development environment and require additional compilation steps.

This thesis presents an attempt of improving usability by integrating a domain-specific language for interaction protocols, called Discourje, into Clojure through its powerful meta-programming facilities. Discourje does not introduce special syntax, compilation steps and is not coupled to any specific IDE.

By using Clojure's macro system to introduce new protocol macros, Discourje becomes part of the Clojure language. Without the need for a dedicated compiler and a specific development environment. Additionally, since Discourje is implemented as Clojure macros, there is no need to learn any new, special syntax for a developer; it is simply Clojure code. Finally, to further improve usability, Discourje mimics the API of a Clojure core library for asynchronous communication and synchronization, called Clojure.core.async, to minimize migration effort. This allows programmers to gradually adopt Discourje.

Clojure programs that use only Discourje to implement communications between threads are guaranteed to be free of communication deadlocks. Another advantage of Discourje and using Clojure is the primacy of immutability: All native Clojure data types are immutable by default. An immutable data model is robust in multi-threaded environments since concurrent updates and race conditions cannot occur. When Discourje is used with Clojure data types only, freedom of data races is guaranteed.

The Discourje run-time system contains a validation engine that, at run-time, monitors communications between threads to detect protocol violations. In benchmarks, Discourje is roughly ten times slower than Clojure.core.async communication and synchronization abstractions, which is similar to other run-time verification tools. Moreover, the benchmarks yield insight into possible optimizations, three of which we describe in this thesis.

0.2. SAMENVATTING

Het schrijven van correcte concurrent software wordt steeds lastiger in traditionele programmeertalen als C en Java, als gevolg van het alsmaar toenemende aantal CPU kernen op chips. Een manier om het schrijven van concurrent software gemakkelijker te maken is door gebruik te maken van domein-specifieke programmeertalen voor interactie protocollen. Deze talen bieden high level abstracties voor low level synchronisatie mogelijkheden en bevorderen het scheiden van de verantwoordelijkheden tussen business logica en coördinatie. Een aanzienlijk nadeel van deze talen is dat de gebruiksvriendelijkheid belemmerd wordt door een drietal kwesties: Deze talen brengen extra syntaxis met zich mee, zijn soms gekoppeld aan een specifieke ontwikkel omgeving (IDE) en vereisen additionele compilatie stappen.

Deze thesis presenteert een poging om de kwesties rondom gebruiksvriendelijk op te lossen door een domein-specifieke protocol taal, genaamd Discourje, in Clojure te ontwikkelen met behulp van Clojure's meta-programmeer faciliteiten. Discourje brengt geen extra syntaxis, speciale compiler of afhankelijkheid aan een specifieke IDE.

Discourje wordt deel van de Clojure syntaxis met behulp van nieuwe protocol macro's door het gebruik van Clojure's macro systeem. Zonder noodzaak van een speciale compiler. Bovendien, sinds Discourje geïmplementeerd is als macro's hoeft een programmeur geen extra syntaxis te leren; het is simpelweg Clojure code. Tenslotte, om gebruiksvriendelijkheid nog verder te promoten bootst Discourje de API van een Clojure core library voor asynchrone communicatie na, om de stap voor het migreren zo klein mogelijk te houden. Dit biedt programmeurs de mogelijkheid om Discourje geleidelijk te implementeren.

Clojure programma's die gebruik maken van Discourje voor het implementeren van communicatie tussen thread zijn gegarandeerd vrij van communicatie deadlocks. Daarnaast bied Discourje door het gebruik van Clojure ook een voordeel met betrekking to immutability: Alle standaard Clojure data types zijn immutable. Een immutable data model is robuust in een multi-threaded programma omdat concurrent updates en race-condities niet kunnen optreden. Als Discourje wordt gebruikt met enkel Clojure data types, wordt vrijheid van data-races gegarandeerd.

De run-time engine van Discourje bevat een validatie engine die, run-time, de communicatie tussen threads doet monitoren om schending van het protocol te detecteren. Uit benchmarks blijkt dat Discourje ongeveer tien keer langzamer is dan Clojure.core.async communicatie en synchronisatie abstracties, wat vergelijkbaar is met andere run-time validatie tools. Ook gaven de benchmarks inzicht in mogelijke verbeteringen, waarvan er drie worden beschreven in deze thesis.

Contents

1

Introduction

In recent years, in software industry, it has become increasingly clear that concurrency and parallelism are important. Since the end of Moore's Law, CPU clock speed no longer grows, and hardware developers try to increase throughput by adding more CPU cores to chips, as they move away from monolithic CPU architectures [SL15].
This shift in paradigm has a significant impact on the software industry since. Now that typical hardware consists of multiple CPU cores, software must become more concurrent to fully make use of their potential parallel throughput. When software is designed to use only one CPU, it will not be able to take full advantage of future hardware improvements. This shift in paradigm can also be seen in the advent of new programming languages like; Clojure[1], Go[2] and Rust[3].

Ever since this shift in paradigm, there has been a lot of research on how to fully utilize multiple CPU cores. A recurring question in this research is how processes and interprocess-communication are managed. To find an answer to this question, there have been numerous studies on so called 'coordination' or 'protocol' languages [JA18; JA13; Arb04; Hon+11; NY14; HVK98; Cop+15; HYC16; Zai+14; Van18]. A protocol language is a supplemental language to express synchronization and communication patterns among concurrent entities (threads, processes, services, etc.) and offers high level abstractions that take care of low level synchronization concepts, like locks and semaphores. These protocol languages provide several software engineering advantages, like forcing programmers to separate business logic from protocol implementation. This allows for better reuse, maintainability and modularity of both the former and the latter.

Despite protocol languages offering separation of concerns regarding business logic and coordination, implementing and maintaining specifications written in protocol languages can become tedious. First, programmers must learn the syntax of protocol languages. Second, some protocol languages allow for integration with only one integrated development environment (IDE). This IDE might not be the preferred IDE for the programmer. Finally, the protocol language might introduce multiple compilation steps, using different compilers,

[1]https://clojure.org/
[2]https://golang.org/
[3]https://www.rust-lang.org/

which may be disruptive to the preferred build process. These inconveniences make it hard to use and adopt protocol languages in mainstream software industry.

1.1. Scientific Contribution & Research Questions

This thesis makes two main contributions. First, the premise of this thesis is that to increase usability and adoption of a protocol language, it should be part of, and integrated in, a general purpose programming language (GPL) itself. This eliminates the need for a programmer to learn new syntax and use a specific IDE. Currently, no GPL with high level protocol abstractions exists. Yet, there do exist programming languages with powerful meta-programming facilities and macro systems that can be used to extend a GPL with new abstractions. A mature and highly wanted GPL[4] that fits this profile is Clojure[FH11; MHB18; JR14]. With the macro system in Clojure, protocol abstractions and constructs can be created as new language features in terms of macros. This is an entirely new approach to implementation of protocol languages and one of the main contributions of this thesis.

Second, by using Clojure and taking advantage of it's native immutable data model, this thesis makes a contribution regarding data race freedom of protocol implementations. An immutable data model is robust in multi-threaded environments since concurrent updates and race conditions cannot occur. However, Existing protocol languages[Hon+11; Arb04] are typically compiled to, or used with, programming languages with mutable data, for instance Java or C. This mutable data is used for communication, and thus senders and receivers own a reference to the same data element, which can lead to race conditions.

More precisely, the research question that is answered in this thesis is:

How to embed protocol specifications into Clojure macros to provide a seamless syntax/IDE experience between GPL and protocol language, while taking advantage of Clojure's immutable data model?

This question can be broken down to the following sub questions:

RQ 1 ***How to create Clojure macros for embedded protocol specifications while taking advantage of immutability characteristics?***

RQ 2 ***How do these macros perform?***

RQ 3 ***Is it possible to increase performance by moving macro execution from run-time to compile-time and what are trade offs?***

Research Question 1: To provide a programmer with a seamless syntax and IDE experience, Discourje is developed. Discourje is a Clojure library that tries to improve usability of a protocol language through three components: First, Discourje presents a domain specific language, as Clojure macros, to define protocol specifications while conforming to Clojure's syntax. This means there is *no special syntax* while using Discourje and, since the DSL is implemented as Clojure macros there is *no need for a dedicated compiler* or *unnecessary*

[4]https://insights.stackoverflow.com/survey/2019

coupling to any specific development environment. The second component of Discourje is an API that offers functions for asynchronous communication and synchronization. The API is designed to mimic Clojure.core.async's API, which is a Clojure library for asynchronous communication, to minimize the effort of migrating to Discourje. The API also offers setup and configuration features. Finally, the third component, is a run-time validation engine that verifies *all* communication and synchronization done through Discourje's API complies with the pre-specified protocol specification.

Discourje is designed to operate on native Clojure data types, that are all immutable by default, for communication and synchronization. When Discourje is used with native Clojure data types only, it *guarantees data race freedom.* Discourje also offers the option to use Java objects for communication, that are often mutable by default. Discourje is unable to guarantee data race freedom on Java data types.

Research Question 2: In order for the macros to serve as a realistic alternative to built-in Clojure concurrency constructs, specifically Clojure.core.async, their performance must be comparable. Investigating how the macros perform compared to Clojure.core.async is an important step in order to assess whether an embedded DSL for protocols truly is a realistic alternative for programmers: no matter how much more usable an embedded DSL is, if performance is bad, adoption will be low (i.e., decent performance is a prerequisite for high usability).

Four types of performance benchmarks were performed for Discourje. Two benchmarks involve running examples used in this research; the one buyer protocol[5] and the two buyer protocol [HYC16]. Additionally, two scalable benchmark types are performed; pipeline and scatter & gather or master-slave pattern.

Research Question 3: The results of the benchmarks must be investigated to see if performance can be improved by moving macro compilation to compile-time instead of run-time. Since Clojure will compile the macros before they are able to be executed, it is possible to move expensive logic from run-time to compile-time. The disadvantage of this approach is the increased compile-time. If compile time is long, programmers might not use the macros. In contrast, if compile time is fast, but run-time performance is low, adoption by programmers could also remain low.

The benchmarks showed that switching from the currently active and monitored communication of the validation engine, to the next pre-specified communication is slow. This switching process involves a query on the protocol specification to find the next permitted communication. This query is done at run-time, which can be moved to compile time through Clojure's macro system. This removes the performance hit of querying the next permitted communication in the protocol specification each time a new communication must be activated for monitoring.

[5]https://github.com/scribble/scribble-language-guide/tree/master/defineprotocol

1.2. CLOJURE COMMUNITY CONTRIBUTION

Discourje was presented as public library in a talk [6,7] during 'Dutch Clojure Days' (DCD) 2019[8]. DCD is a large annual international gathering of Clojure enthusiasts and practitioners in the Netherlands. It is a free, non-profit conference, fully organized by, and for the community with a full day of (technical) talks. After the talk some developers recognized the lack of validation in Clojure.core.async synchronization abstractions and thought Discourje was a great idea. They also understood the advantage Clojure offers towards designing a DSL for Discourje through it's homoiconic property and native immutable data types.

1.3. RESEARCH OVERVIEW

The structure of this research is as follows. The chapter 2 will provide background information on protocol languages, where they originated from, and why they are important to this research. This chapter will also describe the need for meta-programming facilities and Clojure, and it includes a short summary of related work.
Additionally chapter 2 will explain the Clojure programming language in more detail. It will illustrate why immutability is such an important characteristic and explains Clojure's syntax.
Finally chapter 2 will provide more detailed information on protocol languages and fundamental concepts involving them. It mainly targets two protocol languages with a major difference in form of representation, textual vs. visual.
Chapter 3 presents the design for Discourje. It describes running examples from the protocol literature, the adopted programming model and Clojure.core.async. Chapter 4 describes the design of the Discourje DSL in detail. Chapter 5 presents the Discourje API and shows implementation examples. Chapter 6 explains the run-time system of Discourje and how validation of communication works. Chapter 7 describes performance benchmarks, their results and possible improvements. Chapter 8 will summarize the conclusions of this research and discuss future work.

[6]https://www.youtube.com/watch?v=Vf6lfrX5caw&t
[7]https://speakerdeck.com/rhamers/discourje-automatically-validated-message-exchange-patterns-in-clojure
[8]https://clojuredays.org/

2

Research context

This chapter sets the scene and describes the context of this research. It touches on the problem space and provides background information.

2.1. Background

Protocol languages have historically been researched and used for web service coordination [Hon+11]. Modern software industry requires highly parallel and distributed systems to support business processes. To let such independent processes communicate with each other, some form of message-passing middleware is typically used.

The purpose of protocol languages is to provide a method to separate the coordination of these processes from business process implementation. They allow developers to pull away some complexity from business processes and encapsulate coordination into its own domain. A protocol language is suitable to control conversations (the act of sending messages back and forth) between processes all following a specific workflow, i.e., a protocol. This protocol may enforce a specification of senders and receivers of messages, message types, and data structures, to enact the conversations through communication channels. A protocol is able to keep conversations between processes separate from each other and can ensure compliance between processes. Without an encompassing concept as a protocol, communication between processes is difficult to implement. Processes would not know what other parties exist, what kind of data will be sent, and which communication channels are used [Hon+11].

2.1.1. State of the art

This subsection will briefly describe two main influences of this research. Details appear in Chapter 4.

Scribble & Pabble

A coordination language called Scribble [Hon+11] utilizes the π-calculus and its type theory regarding session types to provide simple coordination processes. An upgraded version of Scribble called Pabble [NY14] also allows for parametrization (number of participants) of protocols to exploit the benefits of multi-party session types. Pabble greatly increases the expressiveness and modularity of the Scribble language.

REO & PR

Another such language, called Reo [Arb04], takes a different approach by allowing the specification and implementation of the protocol to be created through graphical syntax. These graphical constructs (directed graphs) represent connectors through which processes can interact with each other. Users are able to compose arbitrarily complex communication structures using a small set of primitive connectors.
Additionally, Reo also allows for parametrization of protocols [Van18]. This is however not included in the graphical syntax. To do this, programmers can use a Reo-to-Pr translator to translate the Reo to a textual representation. Subsequently, programmers are able to manually modify the textual output to implement parametrized protocols. The modified textual output can then be converted into Java.

2.1.2. USAGE

Using a protocol language to coordinate processes at a high level reduces coupling and separates coordination for business processes. This means that if the business process or protocol specification needs maintenance or modifications, the one will not affect the other.
Another useful area for protocol languages is at a much lower level. More specifically, they are useful for coordinating business logic, i.e., source code responsible for the correct function of elicited requirements. The use of protocol languages at this low level of abstraction arises from the fact that *Moore's Law* stagnates.

In the past, software developers could rely on Moore's Law [SL15] to increase the performance speed of their software over time. A program that was not fast enough at the start of development, would be fast enough at the end of the development cycle. Just by the increase of raw computing power of the hardware. Since the end of Moore's Law in roughly 2004, this is no longer possible. CPU clock speed has settled, and we will most likely not see a great increase in the future. Instead, hardware developers have found another way to increase throughput. By increasing the amount of chips (cores), instead of one monolithic architecture of a single chip, there are now multiple chips working cohesively to be able to increase throughput on CPUs.
Protocol languages are used for managing synchronization and communication between threads. They can enforce the same coordination between threads, at a low level of abstraction, as they can between business processes at a high level of abstraction.

2.1.3. CHALLENGES

When developers decide they want to utilize a protocol language for coordination, they are faced with two main challenges. The first one concerns usability: Protocol languages require a programmer to do additional implementation work. The second challenge arises when protocol languages are used at a low level of abstraction and operate on shared memory.

USABILITY

What existing protocol languages have in common is not only the separation of business logic and coordination, but also *an additional syntax for coordination logic, extra compile steps, separate IDE's and if possible, a custom parametrization step.* When programmers choose to utilize a protocol language for coordination, they are faced with extra work which poses a threshold. In more detail, a protocol language is a supplemental language, and the protocol specification written in it must be developed and maintained in parallel with business logic. The programmers must learn a specialized syntax for the protocol language and apply it to their application correctly. Programmers first have to learn the semantics of the protocol language; only then are they able to apply a correct coordination scheme to their application. A consequence of an additional syntax is that the protocol language authors might not offer integration with the programmers' preferred IDE. This can also be perceived as a disadvantage and further increases the threshold for using a protocol language. Another aspect of a specialized syntax is that there will most likely be an additional compilation procedure. Furthermore, a more advanced protocol language might offer parametrization capabilities which increases the learning curve of the syntax and could pose even more additional steps to development. With all these factors taken into consideration, comprehensibility when using a protocol language decreases.

SHARED MEMORY

Using a protocol language at a low level of abstraction presents an obstacle. At a low level of abstraction, protocols are supposed to comprehensively coordinate all low level code. However the data being communicated is likely mutable since traditional protocol languages are intended to be for Java or C. Thus, protocol language designers must find an efficient and safe manner to act on shared mutable state to avoid data races. Existing protocol languages try to solve this problem in different ways. First, by copying all data between sender and receivers to make sure no reference points to the same data, but this introduces a significant performance hit. Second, utilizing static analysis tools to try and detect race conditions early on, and throw compile-time errors or warnings. With immutable data structures in Clojure, copies are transferred instead of references in an efficient manner. Senders and receivers operate on different data, thus an immutable data model will provide for an even more robust environment when used in combination with a protocol language.

2.1.4. SOLUTIONS

To counteract the problem regarding usability, a more practical way for defining coordination through protocol languages is essential. Having a GPL with coordination constructs and abstractions without additional development steps, may increase usability for protocol languages. Unfortunately, no existing programming language offers such capabilities.

META-PROGRAMMING

Luckily there are several general purpose languages that offer powerful meta-programming facilities. Using the meta-programming functions, one could implement protocols as extended languages features. In C, for instance, there is a macro system that allows programmers to modify system behavior during compilation time. A programmer can define functions and objects and replace elements in a preprocessing step during compilation. C++ offers an intricate meta-programming programming feature through its templating system.[1] There are also plenty of dynamic (dynamically typed) languages like JavaScript, Python and Ruby that have *eval* functions to use strings and produce results. The most well-known powerful macro system in a GPL is perhaps in LISP.

[1] https://gcc.gnu.org/onlinedocs/cpp/

LISP

LISP is a functional language and has a very small syntax. Any new or complex behavior is added through the use of macros. This is achieved through another very important characteristic of LISPs namely, there is no difference between code and data. This property is called *Homoiconicity* and means that the syntax has the same structure as data. Through clever usage of this property, programmers are able to add new behavior through macros, even at run-time. Code can be transformed at run-time to change its behavior.

CLOJURE

A programming language that offers an **immutable data model** and has a powerful macro system like a LISP for implementing **protocol specification constructs as extended language features**, is Clojure. With immutable data there cannot be concurrent update problems, equality has meaning (data is immutable and thus identity is preserved) and as a result, reasoning about data and possible states is simplified. Immutable data can always be shared between threads since they do not allow modification. Thus, if data is immutable, protocol languages will be able to coordinate the business logic without restriction or concern of accessing memory which is already being used.

Clojure is a functional language which runs on both the Java Virtual Machine (JVM) and the Common Language Runtime (CLR). Having two large and commonly used target platforms greatly increases adoption by programmers. Clojure has specifically been designed to support a high level of concurrency. Native Clojure data types are therefore immutable. Since Clojure runs on a host (JVM or CLR), a specific goal when designing the language was to allow seamless interoperability, for example with Java. This provides programmers with the option to use popular Java libraries directly in Clojure.

Clojure falls in the Lisp family tree which brings all advantages any LISP has. The syntax is small, it is homoiconic and functional. Clojure is not a pure functional language (like Haskell) since it offers interoperability with other languages like Java, which support assignment. This means exploiting Java objects in Clojure code could very likely still result in race conditions. The language author, Rich Hickey justifies this by saying that any *real program requires dangerous things*, and thus he will not forbid it.[2][3] Plus, Clojure is dynamically typed where Haskell is static. To help programmers with shared mutable state, Clojure offers Software Transactional Memory (STM) which acts in the same manner as database transactions work. A simple example of this is a counter which is incremented from two threads. If both threads are accessing the same counter, one of them will retry until it is able to increment the counter. More on the Clojure STM will be described in chapter 3.

Using Clojure, protocol and coordination constructs could be implemented as macros and offer programmers the advantages of protocol languages. Clojure's characteristic of immutability will alleviate concerns about shared data.

[2] https://www.youtube.com/watch?v=nDAfZK8m5_8&t=4184s

[3] https://github.com/dimhold/clojure-concurrency-rich-hickey/blob/master/ClojureConcurrencyTalk.pdf

2.2. Clojure programming language

The main motivation for choosing Clojure as target programming language in this research is its inherited properties from LISP. Clojure was publicly released in 2007. It is a mature language with significant adoption in the software industry. Clojure runs on major target platforms like the Java Virtual Machine (JVM) and the Microsoft Common Language Runtime (CLR). Clojure can also be compiled to target JavaScript platforms.

2.2.1. Heritage

LISP stands for ***LIS***t ***P***rocessing. It was invented by John McCarthy in 1958 and was the first major adopted *functional programming* language. LISP is based on lambda calculus and thus offers a simple notation and syntax. The syntax of LISP is so small that any complex computation must be added through macros. (More on macros later, or see appendix A.) Clojure is based on LISP. It is derived from the LISP language and therefore inherits many its properties. There are some minor differences between LISP and Clojure. Clojure has additional data structures like *Vectors*, *Maps* and *Sets*, where LISP has only *Lists*. Clojure provides different data types to allow programmers to choose one that fits for example performance requirements.

2.2.2. Immutability

One major difference Clojure has compared to LISP is that native Clojure data types are all immutable. The language was originally designed to be very robust in multithreaded programs/environments. Once a value is set, it cannot be changed. Clojure simply creates a new value including the changes. Immutability is a property of any native Clojure data-structure. Clojure does this by clever list manipulation and linking. When an element to a list is added, there will simply be a new head which links to the rest of the list. This is reliable since all types are immutable and thus the rest will never change. When it is absolutely necessary to update a certain variable, Clojure's Software Transactional Memory will automatically make sure there are no concurrent update problems. When there are multiple updates occurring at the same time the STM will block the updates and make them retry later. Nonetheless, the programmer has to consider side-effects from retrying.

2.2.3. Interoperability and host symbiosis

Another distinct characteristic of Clojure is that it runs on a host platform. This also was a very important requirement when the language was originally designed. Running on a host platform like the JVM offers multiple advantages like free:
garbage collection, Memory and threading model, stack, type system, exception handling, (just in time) compilation, security and seamless integration, ad-hoc interoperability.
Through interopability any Java library can be called straight from Clojure.

2.2.4. Syntax

This subsection will provide information on the syntax of Clojure and LISP in general. First the notation will be explained followed by the concept of *homoiconicity*. This will serve as an introduction to macros. Unfortunately, macros are *not* free in terms of usability and readability, they bring a certain amount of complexity which will also be explained. This section will end with an example explaining interopability features.

READ EVAL PRINT LOOP

Clojure offers a dynamic development environment through a Read Eval Print Loop (REPL). With the REPL small code snippets can quickly be defined and tested. All following Clojure code snippets can be entered in the REPL.

NOTATION

The LISP notation is famous for its pre-fix notation and stacked parentheses. In contrast, imperative (C-derived) programming languages are written with in-fix notation. A Java function notation is defined as follows:

Listing 2.1: Java function definition

```
private int sum(int x, int y, int z) {
        return x + y + z;
}
```

The function can be called like this:

Listing 2.2: Java function call

```
int six = sum(1,2,3);
```

A function in Clojure is defined as following:

Listing 2.3: Clojure function definition

```
(defn sum[x y z]
        ( + x y z ))
```

The function can be called and set its return value like this:

Listing 2.4: Clojure function call

```
(def six (sum 1 2 3))
```

A slightly more complex example is when the order of arguments matter like in a formula: $(x + y) * z$

The Java function looks very similar to the notation of the formula:

Listing 2.5: Complex Java function definition

```
private int formula(int x, int y, int z) {
        return (x + y) * z;
}
```

In Clojure the function is written like this:

Listing 2.6: Complex Clojure function definition

```
(defn formula[x y z]
        ( * z (+ x y)))
```

In Clojure, anything that is put as the first argument in a *list* can be treated as a function call. In Java the + symbol is an operator which is part of the Java syntax, in Clojure the + symbol is just a macro. Clojure functions and macros do not need to have a textual name. A function name like *-=*a* would work just fine (but is not very descriptive). This trait has a profound impact on the concept of homoiconicity.

HOMOICONICITY

LISP is famous for its powerful macro system. A well known property for any LISP is *code is data and data is code*. The technical term for this property is *Homoiconicity*, and means that the structure of the syntax is the same as data (structures). Consider the following simple example to demonstrate homoiconicity:

Listing 2.7: Clojure function call

```
(+ 1 2 3)
```

Listing 2.8: Clojure data structure

```
(1 2 3)
```

These structures look very similar, but differ in their meaning. The list in listing 7 includes a function call + with three arguments. The list in listing 8 is a data structure with three numbers. If the data structure from listing 8 would be executed, the program throws an exception for no knowing the *1* function.

This is a *very* simple example of homoiconicity but it describes the property very well. It demonstrates that the structure of a function call is the same as a data-structure. There is however a possibility to 'execute' the data-structure from listing 8, without evaluating the first argument as a function call. This can be done with *quoting*. Anything that is quoted is interpreted as mere data. Consider the following example.

Listing 2.9: Clojure quote

```
'(1 2 3)
=> (1 2 3)
```

This looks like a function call but the quote tells the compiler to interpret the rest as data and not as a function call. This means the 1 in the list will not be treated as a function call and the compiler will simply evaluate the list as a data-structure. The quoted data structure can be evaluated by surrounding it with the eval function:

Listing 2.10: Clojure evaluate quote

```
(eval `(+ 1 2 3))
=> 6
```

The attentive reader might have spotted the use of a different kind of quote. To make matters slightly more complex, there are different kind of quotes in Clojure. There is the *quote* (’) and *syntax quote* (‘). The normal quote looks straight and narrow, and the syntax quote is rotated. The syntax quote is known as 'backquote' or 'backtick.' The difference between the two quotes is that the 'normal' quote tells the compiler to evaluate anything following it as data. In contrast, the syntax quote tells the compiler to try to resolve the quoted structure as code in the current namespace. Consider the following example:

Listing 2.11: Clojure syntax quote

```
1 '(sum 1 2 3)
2 => (sum 1 2 3)
3
4 `(sum 1 2 3)
5 => (user/sum 1 2 3)
```

The use of the normal quote in the first line allows the compiler to evaluate the list as a data structure with a symbol as first value and rest as integers. The syntax quote on line 3 tells the compiler to resolve sum which points to the user namespace. When data is quoted with the syntax quote, it can also be *unquoted*.

Consider the following definition of a variable called *five* which is initialized by the integer 5. After inserting *five* in a list the compiler produces the following result.

Listing 2.12: Evaluate

```
(def five 5)
=> #'user/five
`(1 2 3 4 five 6 7)
=> (1 2 3 4 user/five 6 7)
```

The quoted list is evaluated as a data-structure and the variable *five* refers to the *five* variable which is defined in the *user namespace*.
Only after unquoting *five* in the list, will the compiler insert the value of *five*.

Listing 2.13: Unquoting

```
`(1 2 3 4 ~five 6 7)
=> (1 2 3 4 5 6 7)
```

The correct unquoting of the sum method form the previous example looks like this:

Listing 2.14: Unquoted Sum

```
`(4 5 ~(sum 1 2 3))
=> (4 5 6)
```

There exists however another form of unquoting called *unquote splicing*. Unquote splicing allows all elements of the unquoted collection to be merged with the invoking collection. The collection will be 'flattened'. Consider the following example of a quoted list, with an unquoted nested list in an attempt to insert its values. Unquote splicing is done with the @-symbol.

Listing 2.15: unquoted nested list

```
`(1 2 3 ~(list 4 5 6))
=> (1 2 3 (4 5 6))
```

This might not be the desired behavior. The nested list *(4 5 6)* is inserted as another list. In order to tell the compiler to interpret only the values of the list and insert them into the main list, the syntax splicing form can be used.

Listing 2.16: Unquote Splicing

```
`(1 2 3 ~@(list 4 5 6))
=> (1 2 3 4 5 6)
```

Unquoting provides the compiler with information about what content of the data structure to interpret as data, and what to interpret as code. The concept of quoting and unquoting plays a major role when writing macros.

MACROS

Clojure macros [JR14] are made from the coherence between *quoting*, *unquoting* and *splicing* of data and executable code and evaluating them correctly. A more intricate example, which demonstrates the importance of quoting and unquoting content in a macro definition is the following. The following example defines a macro called *unless* which only evaluates its content when the expression evaluates to false [FH11]. For a more detailed description on Clojure compilation and macros see Appendix A. Consider the following definition of the *unless* macro.

Listing 2.17: Unless

```
(defmacro unless [condition & body]
  `(if (not ~condition)
     (do ~@body)))
```

The macro takes two arguments called *condition* and *body*. The *syntax-quote* provides the compiler with information about the *if* form acting as a template for the succeeding expression. The *condition* block will be inserted by unquoting. The *body* argument will be inserted as an expression in the *do* block. The *body* block gets unquoted and the @-symbol allows the compiler to interpret anything in the *body* expression to be evaluated as code. Both the *if, not, and do* forms are native to Clojure.

Invoking the unless macro with some simple arguments produces the following results:

Listing 2.18: Invoking Unless

```
(unless true (println "I will not be printed."))
=> nil
(unless false (println "I will be printed."))
I will be printed.
=> nil
```

Notice the use of the ampersand in the definition of the macro. It denotes a parameter, in this case *body*, as a 'rest-param' which allows the compiler to evaluate all following arguments.

Listing 2.19: Ampersand

```
(unless false
        (print "I ")
        (print "will ")
        (print "be ")
        (print "printed."))
I will be printed.
=> nil
```

Although the *unless* macro example is only slightly more complex than unquoting the variable *five* in the list, the correct placement and usage of quoting, unquoting and splicing is very important.

COMPLEXITY

Clojure offers a macro to help programmers with evaluating macro definitions. A macro can be *expanded* to show its interpreted structure. It produces the following result when macro expanding the unless macro.

Listing 2.20: macroexpand unless call

```
(macroexpand
  '(unless false (println "I will be printed.")))

=> (if (clojure.core/not false)
     (do (clojure.core/println "I will be printed.")))
```

The entire macro gets syntax-quoted to allow macroexpand to inspect the definitions. The *if* and *do* forms are native to Clojure, *not* comes from the Clojure.core namespace. The values inserted by the macro call *condition* = *false* and *body* = *(println "I will be printed.")* are inserted into the syntax.

Macro expanding can help with debugging macro definitions. It allows for inspection of the generated syntax to improve readability and give the programmer the opportunity to repair bugs. When a macro definition is not written correctly it can be very difficult to determine the fault. Clojure is well-known for throwing surprising exceptions.

INTEROP

Clojure is able to seamlessly interact with Java as a result of its host symbiosis. Consider the following example of the Java [string].toUpperCase() call.

Listing 2.21: simple interop

```
(.toUpperCase "hello world")
=> "HELLO WORLD"
```

The toUpperCase() is not a Clojure function but comes straight from Java. Clojure provides a number of convenient macros to interact with Java like the .. macro which inserts . in between all its arguments as if it were an in-fix notation Java call:

Listing 2.22: System.GetProperties.get(”os.name”)

```
(.. System (getProperties) (get "os.name"))
=> "Windows 10"
```

Another useful macro for interopability is the *doto* macro which calls all function arguments on the target object.

Listing 2.23: Doto

```
(doto (new java.util.HashMap) (.put 1 "hello") (.put 2 "world"))
=> {1 "hello", 2 "world"}
```

2.3. PROTOCOL LANGUAGE FUNDAMENTALS

This chapter provides information on the fundamentals of protocol languages mainly focusing on Scribble[Hon+11; NY14] and Reo[Arb04; JA18]. It provides simple examples for both languages and explains the concept of parametrization.

2.3.1. SCRIBBLE

Scribble is a *textual* protocol language designed to provide a formal and intuitive interface for specifying coordination schemes. Scribble utilizes π-calculus and multi-party session types. In this context, a session is a set of interactions that form a conversation. A session is set up between multiple parties by a shared name, which acts as a public point for interaction. Session channels are then generated and shared by all parties to facilitate communication [HYC16].

Scribble allows developers to implement interaction behavior by using a generated protocol specific conversation API, which is a high-level message passing interface. Scribble relies on three crucial properties for message transport. First, all send actions are non-blocking which provides asynchrony of conversations. Second, the order of messages sent between participants in a single conversation is preserved. Third, a message in transmission is never lost or tampered with. Scribble provides several kinds of interactions between participants. A simple interaction might be a message from *A* to *B*. Consider the following example for a simple 'hello world' implementation in Scribble.

Listing 2.24: Scribble hello world

```
import Message;

protocol HelloWorld {
        role You, World;
        greet(Message) from You to World;
}
```

In this example, a conversation involves two participants taking the role of You or the World. The conversation sends a single message from You to World by an operation called 'greet' with a value of type Message [Hon+11].

SYNTAX

Besides the simple interaction between two participants of the previous example, Scribble supports more advanced features. Interaction can be *sequenced* or sent in *parallel*. Message flow can be branched through a *choice* form. Scribble also supports recursion and nesting of protocol definitions. The following section will shortly describe these concepts. All following code snippets come from [Hon+11].

SEQUENCING

Sequencing allows messages to be sent to specific participants taking place in the specified order. It enforces messages to be sent from one participant to another. The following example shows a sequenced protocol.

Listing 2.25: Scribble sequence

```
order(Goods) from Buyer to Seller;
deliver(Shipment) from Seller to Supplier;
confirm(Invoice) from Seller to Buyer;
```

In this example a *Buyer* sends an order to the *Seller*, the *Seller* will subsequently send a shipment order to the *Supplier* and an invoice to the *Buyer*.

PARALLELISM

Parallel or 'unordered' interactions can be defined when there is no particular order for messages to be received. Consider the following example:

Listing 2.26: Scribble parallelism

```
messageType from Me to Participant1,Participant2,.., ParticipantN;
```

Messages are sent from *Me* to *Participant1*, *Participant2* up to *ParticipantN* in no particular order.

BRANCHING

Branching allows a protocol to prescribe some choice to continue between different protocol continuations following a certain scenario. Consider the following example where a *Buyer* sends an order to *Seller*, *Seller* returns an invoice and waits for *Buyer* to either accept or decline.

Listing 2.27: Scribble branching

```
order(Goods) from Buyer to Seller;
choice from Seller to Buyer {
        accept(Invoice):
                payment(Details) from Buyer to Seller;
        decline():
                 end;
}
```

RECURSION

Scribble allows recursion in protocols definitions to implement repetitions. The following example shows a scenario where the *Seller* continuously accepts orders from *Buyer* until the *Seller* receives a decline. Recursion is marked by a named block in the form of #Name.

Listing 2.28: Scribble recursion

```
rec X {
        order(Goods) from Buyer to Seller;
        choice from Seller to Buyer {
                accept():
                        ..
                        #X;
                decline():
                        end;
        }
}
```

2.3.2. PABBLE

Scribble can be ‘extended’ to allow parametrisation to describe a global interaction topology through an expressive notation designed for a *variable* number of participants. This Scribble extension, called Pabble, can automatically generate code for such parametrised protocols [NY14]. Parametrization provides the option to scale the communication scheme if required. Consider the following example of a Scribble protocol called ‘Ring’, it defines a simple loop between participants called ‘Workers’.

Listing 2.29: Scribble Ring protocol

```
global protocol Ring(role Worker1, role Worker2, role Worker3,
 role Worker4) {
        rec LOOP {
                Data(int) from Worker1 to Worker2;
                Data(int) from Worker2 to Worker3;
                Data(int) from Worker3 to Worker4;
                Data(int) from Worker4 to Worker1;
                continue LOOP;
        }
}
```

In this example, adding a fifth *Worker* to the ring would mean changing the protocol definition. Now consider another example which exploits the benefits of parametrization.

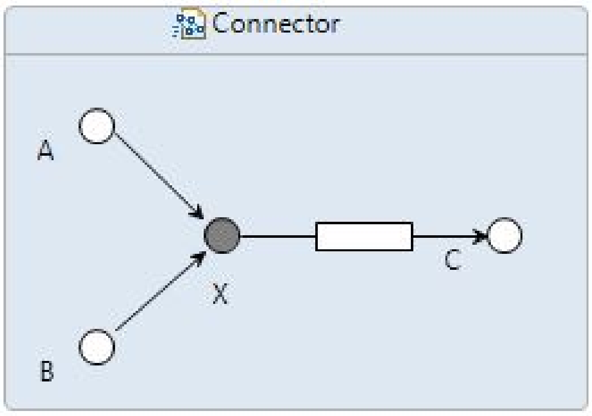
Listing 2.30: Pabble Ring protocol

```
global protocol Ring(role Worker[1..N]) {
        rec LOOP {
                Data(int) from Worker[i:1..N-1] to Worker[i+1];
                Data(int) from Worker[N] to Worker[1];
                continue LOOP;
        }
}
```

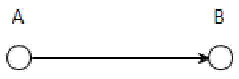
This Pabble protocol declares the *Ring* protocol by parametrizing the workers, allowing a more flexible approach for adding or removing them.

2.3.3. REO

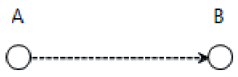
Reo [Arb04] takes a different approach on representation than Scribble, while still focusing on protocols among processes. Reo has a *graphical* syntax where protocols are described as labeled directed graphs called 'circuits'. These circuits represent the flow of messages between processes. The edges in these graphs represent communication channels, nodes represent logical places where channels begin or end. Reo supports several types of channels each of which provides a different kind of characteristic. Nodes either act as a source or sink of communication, or both. A 'mixed ' node nondeterministically takes offered data and replicates it to all outgoing channels. Consider the following example where node A and B represent source nodes, C is a sink node and X is an intermediate node.



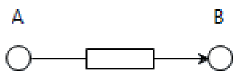
The edges within this graph pointing from A and B to X are default synchronous channels, the channel pointing from X to C is a channel implementing an asynchronous channel with a 1-capacity buffer. Reo supports different kinds of channels [Arb04]:



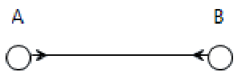
Sync: has a source and a sink. It accepts data synchronously and data is never lost.



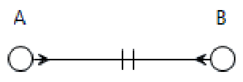
LossySync: has a source and sink. It always accepts all data; if there is no valid operation on the sink, the data are lost.



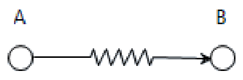
FIFO: has a source, sink, and an unbounded buffer. The source is guaranteed to accept all data, which will be kept in a FIFO buffer. Operations on the sink end of the channel take items from the buffer following FIFO.



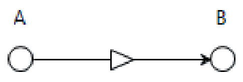
SyncDrain: has two source ends. It accepts data through these ends only synchronously. All data items written to this type of channel are lost.



AsyncDrain: has two source ends. It guarantees that two operations never succeed simultaneously. All data written to this channel is lost.

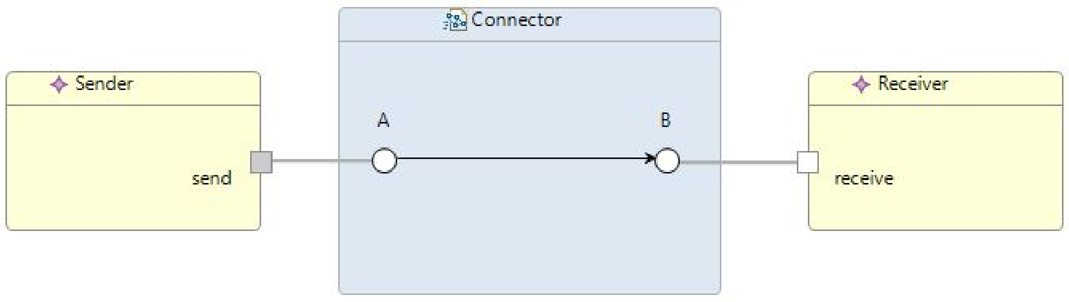


Filter: has a source and sink. It behaves the same as a *Sync channel*, except that if data does not match some predefined filter pattern, it is immediately lost.



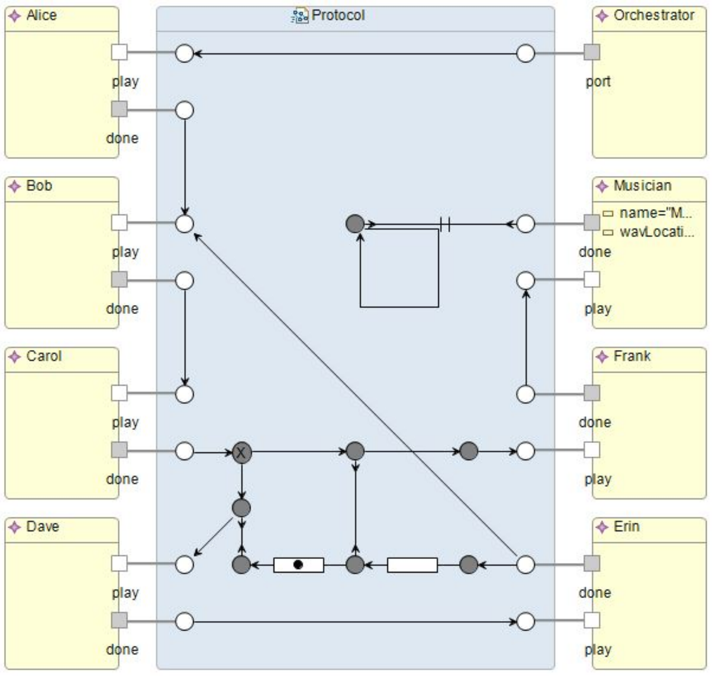
Transform: has a source and sink. It is able to transform data to some appropriate format during transport.

With these channels and nodes, a protocol can be specified. Reo allows programmers to link processes to source or sink nodes through ports. A process is defined as a static method in Java. Reo provides programmers with the option to perform operations on so called boundary nodes. Boundary nodes in Reo are defined as sources, without incoming data, and receivers, without outgoing data. This means, intermediate nodes cannot be operated on, but data will be replicated to all outgoing channels [JA13]. These boundary nodes present ports to attach operations to, but also an option to *wire* multiple Reo diagrams together. A simple diagram with a sync channel between node A and B interacting with processes called Sender and Receiver might look like this:



In this diagram both Sender and Receiver allow IO operation on ports which are represented by the white squares. Through these ports data is sent or received. Programmers are able to create arbitrarily complex diagrams. The following diagram implements a music orchestra and is taken from the PrDK demo.[4]

[4]http://www.open.ou.nl/ssj/prdk/



Reo can also be extended to support parametrization. Unfortunately, the graphical syntax does not support this. In order to support parametrisation, Reo relies on a textual representation called Pr [Van18]. Programmers can use a translator to parse the graphical circuits to a textual form, and modify them accordingly to support parametrization.

2.3.4. FUNDAMENTALS

This section will briefly describe fundamental aspects of protocol languages.

First, in any kind of conversation, there are senders and receivers. Scribble is built on the assumption that send actions are *non-blocking* and message order is preserved for correct communication. Reo, in contrast, offers data preservation as a choice and allows data to be lost when the protocol describes so, although data flowing through channels will not be lost if not instructed. These are important aspects of a protocol language, since it makes them easier to reason about: When messages are guaranteed to be sent and data is always preserved, comprehensibility of the protocol increases. The protocol also guards messages, so they cannot be tampered with.

A protocol provides a distinct interaction schema between sender(s) and receiver(s). This schema can include messages flowing in parallel or sequence. A sequence defines a control flow from the entry point of the protocol, through intermediate points, to end receivers. Parallel flows allow interleaving interactions which do not need to follow a specific order. Control paths can be directed by branching constructs which allow senders or receivers to choose a specific branch depending on certain conditions. Data flowing through channels can also be subjected to filters that accept data only when specific conditions are met and might include queuing mechanisms like *First In First Out* (FIFO) or *Last In First Out* (LIFO).

A protocol must also provide an API for a GPL to interface with. Programmers must be able to communicate through the protocols and react based on data received. A circuit in

Reo can be compiled to Java and interacted with. Scribble and Pabble generate source code in a preferred (but supported) language which can be used in an application.

More advanced protocol languages might offer parametrization options. Scribble offers parametrization through an extended version called Pabble [NY14], and Reo offers these capabilities through Pr after compiling the circuits to textual form.

3

Protocol Language Design

This chapter gives a brief overview of the Discourje protocol language and all its components. The chapter starts with a section that presents the running example used in the rest of this thesis, derived from the protocol language literature. The second section describes a programming model commonly used for concurrent programming. The third section describes a Clojure library that closely follows this programming model, and the chapter ends with a description of components required for the new protocol language.

3.1. Communication patterns

This section will explain two common protocols, the *one buyer protocol*[1] and *two buyer protocol*[HYC16].

One buyer protocol The one buyer protocol, visualized in Figure 3.1, represents the use-case of buying a product. It is a simple protocol where the *buyer* asks for the *quote* of a *product*. The *seller* is then offered a *choice*, illustrated with the dashed lines in the figure below, and able to respond with an *out of stock* message or with the *quote* of the product. The *buyer* can then continue ordering his product and the communication will end with an *order acknowledgement*.

[1]https://github.com/scribble/scribble-language-guide/tree/master/defineprotocol

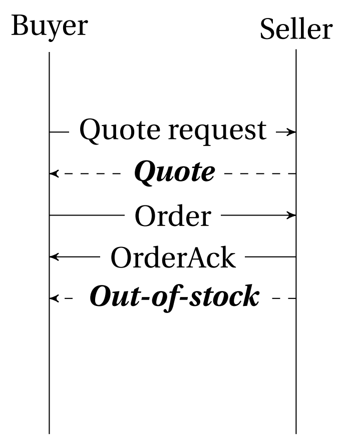


Figure 3.1: Sequence diagram for One Buyer Protocol

Two Buyer Protocol The two buyer protocol, visualized in Figure 3.2, represents the use-case of buying a product where the two buyers must consult with each other to determine how much each of them is willing to contribute to the total costs. It starts with an interaction from *buyer1* to *seller* asking the price of a product, identified by its title. The *seller* will respond by sending a quote to *both buyers*. Next, *buyer1* will ask *buyer2* if he can contribute half of the amount. *Buyer2* has to make a choice now, he can either accept or decline. If *buyer2* accepts he will confirm the order by responding to the *seller* with an ok message and also send his address to the *seller* in a separate message. The *seller* in this case responds by sending a message with a delivery date to both buyers, and then the protocol terminates. If *buyer2* does not accept, he will send the quit message to the *seller*, ending the protocol.

To demonstrate all features of Discourje the protocol is extended with recursion to be able to demonstrate a looping mechanism. The recursion is signified by not only sending the date from *seller* to *buyer2* but to both buyers. When *buyer1* receives the date, the protocol is repeated, and buyer1 will order another product.

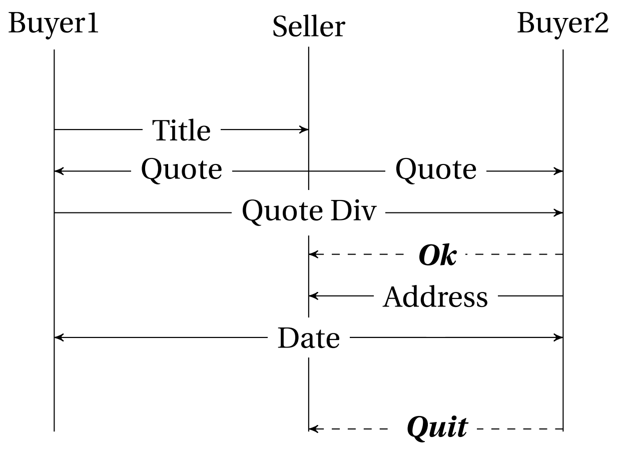


Figure 3.2: Sequence diagram for Two Buyer Protocol

3.2. PROGRAMMING MODEL

Parallel and concurrent programming introduce an additional level of complexity compared to sequential programming. Therefore, it is desirable to program through abstractions and modularity.

A common model for handling concurrent programming is a *Tasks & Channel* programming model [Fos95]. This model offers abstractions that can be composed in a modular manner to reduce the complexity of concurrent and parallel programming.

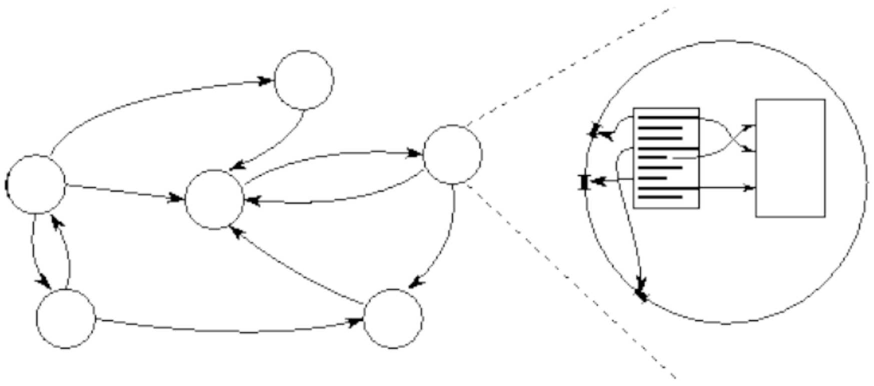


Figure 3.3: Tasks and channels programming model [Fos95].

In the simple representation of the tasks and channels programming model shown in Figure 3.3, the circles represent tasks and the arrows shown in Figure 3.3, the channels between them. The figure shows both the state of the computation as well as a detailed image of a single task. A computation consists of a set of tasks which need synchronization and communication to work together. A task encapsulates a local sub-program with its own local memory. A channel is a message queue where a sender can put messages and a receiver can take messages, blocking when no messages are available. This programming model allows for decoupling of computation and synchronization:, since a task is only communicating with in/output channels it does not need to coordinate with other tasks directly.

3.3. CLOJURE.CORE.ASYNC

Clojure.core.async is a library that provides facilities for asynchronous programming and communication using channels following Hoare's Communicating Sequential Processes (CSP) [BHR84]. It supports real threads or shared threads that run in a thread pool. A key property of these channels is that they are blocking, while buffers can be introduced to manage the work load. Blocking refers to the 'pausing' or 'stopping' a thread when a put or take operation on a channel is not able to proceed due to buffer restrictions. The fundamental operations for channels are sending and receiving values through put and take operations.

Clojure.core.async example The Clojure.core.async API important for the implementation of Discourje can be simplified to a subset of operations. Discourje core logic should support the essential operations to communicate through channels like blocking send and receives on channels and channel creation. A summary of supported functions is a follows:

- ***(>!! & chan, value)*** Put a value on a channel; will block if no buffer space is available.
- ***(<!! & chan)*** Take value from channel; will block if nothing is available.
- ***(chan & buf)*** Generate a channel with an optional buffer size argument.
- ***(thread & body)*** Execute the body in a new thread.
- ***(close! & chan)*** Close the given channel.

An example of the use of these functions is shown in Listing 3.1.

Listing 3.1: Clojure.Core.Async

```

(def alice-to-bob (chan 1))
(def bob-to-alice (chan 1))

(defn alice []
  (>!! alice-to-bob "Foo")
  (println "Alice received: " (<!! bob-to-alice)))

(defn bob []
  (println "Bob received: "(<!! alice-to-bob))
  (>!! bob-to-alice "Bar"))

(thread (alice))
(thread (bob))

=>Bob received:  Foo
=>Alice received:  Bar

(close! alice-to-bob)
(close! bob-to-alice)
```

The listing above shows the creation of two channels named *alice-to-bob* and *bob-to-alice*. These channels are used by the respective senders in the alice and bob functions. The *alice* function starts by putting (>!!) the *Foo* message on the alice-to-bob channel, followed by a take on the bob-to-alice channel to wait for ***any*** data to be put on the channel.
The *bob* function starts with listening on the *alice-to-bob* channel for ***any*** data. When the data is taken from the channel bob will put a *Bar* message on the *bob-to-alice* channel. Lastly, the channels are closed to free the memory.

One buyer protocol

Listing 3.2: One buyer Clojure.Core.Async

```
(def buyer-to-seller (chan))
(def seller-to-buyer (chan))

(defn buyer "Logic representing Buyer" []
  (>!! buyer-to-seller product)
  (let [quote (<!! seller-to-buyer)]
   (if (.isInStock quote)
    (do (>!! buyer-to-seller (doto (Order.) (.setProduct product) (.setQuote quote)))
       (println (<!! seller-to-buyer)))
    (println "Book is out of stock!"))))

(defn seller "Logic representing the Seller" []
  (if (in-stock? (<!! buyer-to-seller))
    (do (>!! seller-to-buyer (doto (Quote.) (.setInStock true) (.setPrice 40.00)
        (.setProduct product)))
      (let [order (<!! buyer-to-seller)]
        (>!! seller-to-buyer (format "order-acknowledgement:
           Order for  product: %s confirmed at price: $%s"
           (.getName (.getProduct order)) (.getPrice (.getQuote order))))))
        (>!! seller-to-buyer (doto (Quote.) (.setInStock false) (.setPrice 0)
          (.setProduct product)))))

(thread (buyer))
(thread (seller))

=>The Joy of Clojure is in stock: true
=>order-acknowledgement: Order for product: The Joy of
  Clojure confirmed at price: $40

=>The Joy of Clojure is in stock: false
=>Book is out of stock!

(close! buyer-to-seller)
(close! seller-to-buyer)
```

Listing 3.2 shows an implementation of the *one buyer protocol* in Clojure.core.async. It starts by the creation of two channels *buyer-to-seller* and *seller-to-buyer*. These channels are used to communicate between the two participants by the buyer and seller functions. The buyer function uses the blocking put (>!!) function to put the quote of the book *The Joy of Clojure* on the buyer-to-seller channel. Subsequently, buyer uses the blocking take (<!!) function to wait for ***any*** value to be put on the seller-to-buyer channel.
The seller function starts with a blocking take from the buyer-to-seller channel to await ***any*** data which is expected to be the Quote. The seller will send back the quote of $40.00 or an out of stock message when receiving ***any*** data, expected to be the quote request, on the seller-to-buyer channel.
Lastly, the buyer and seller functions are started on separate threads.

Protection An important point to make here is that the Clojure.core.async library offers no protection against communication mismatches whatsoever. The put and take operations have no control mechanism and are always allowed on the channels when buffer space is available. *This is also explicitly stated in 2013's announcement blog*[2]. Discourje will add a layer of protection and the next section will give a short summary of the required components.

3.4. COMPONENTS

The core functionality of Discourje consists of three components. The first component is the Discourje DSL that is used to create protocol specifications. A protocol specification is required for Discourje as the monitored pattern for communication and synchronization at run-time. See Chapter 4 for detailed information about the DSL.
The second component is an API that offers specialized send and receive abstractions that are subjected to validation against the pre-specified protocol specifications of the DSL. These abstractions are built as a layer on top of Clojure.core.async communication functions and are designed to mimic Clojure.core.async to minimize the migration effort. See Chapter 5 for more information about the Discourje API.

[2]http://clojure.com/blog/2013/06/28/clojure-core-async-channels.html

The third component is a run-time validation engine that is used by the specialized send and receive abstractions for validating any communication and synchronization. The monitor requires a protocol specification, specified with the Discourje DSL, to validate communication through the Discourje API. Chapter 6 presents a detailed description of the validation engine.

4 Domain Specific Language

This chapter gives a description of the Discourje protocol language.
This chapter starts by defining DSL constructs for each supported feature of Discourje. The second section shows how the one and two buyer protocols can be defined in the Discourje DSL.

4.1. Domain Specific Language

The purpose of the domain specific language (DSL) is to express the protocol clearly. At the core of the DSL is an *atomic interaction*, which represents a communication from A-to-B. Then, there are special forms to compose more complex configurations of these atomic-interactions.

The following paragraphs will present examples for each interaction type. For the purpose of explanation in this paper, state machines are added to show how each interaction type behaves. These state machines are *not* used at run-time in the implementation of Discourje.

Atomic Interaction An atomic interaction is the simplest definable interaction. Such an interaction represents a communication between two participants, i.e., sender and receiver. All interactions defined in Discourje must be identifiable for the *validation engine* in the run-time system to work properly (see Chapter 6 for information about the validation engine). This means an atomic interaction is made up from *three* elements.

- *Label:* Label to identify an interaction with.
- *Sender:* Sender acts as the source of the communication.
- *Receiver:* Receiver acts as the sink of the communication.

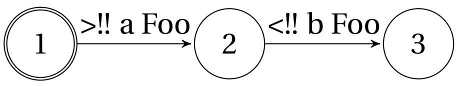
An atomic interaction defined in Discourje and Clojure syntax looks like the following:

Listing 4.1: Atomic Interaction

```
(-->> "Foo" "Alice" "Bob")
```

The arrow notation represents an atomic interaction. It requires a Label, sender and receiver as input arguments. The example above shows an interaction that demonstrates *Alice* sending a message labeled *Foo* to *Bob*.

The state machine respesenting this particular flow would be the following:



In this, and all following state machines in this section Alice, Bob and Carol will be abbreviated to a, b and c respectively

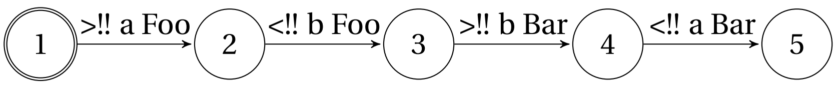
The state machine illustrates three states: a start, intermediate and end state labeled 1, 2 and 3 respectively. State 1 shows an outgoing edge to state 2 which represents the send of the Foo message by *Alice*. State 2 shows an outgoing edge to state 3 representing the receive of the Foo message by *Bob*. State 3 is the end of the state machine. Note that an atomic interaction is complete when communication identified with the label is *both* send as well as received.

Sequencing Discourje allows sequencing of interactions by writing multiple atomic interactions in order. An example of sequencing would be the following:

Listing 4.2: Sequencing

```
(-->> "Foo" "Alice" "Bob")
(-->> "Bar" "Bob" "Alice")
```

The protocol above represents *Alice* sending the *Foo* message to *Bob*, and *Bob* responding by sending the *Bar* message to *Alice*. The state machine representing this protocol is:



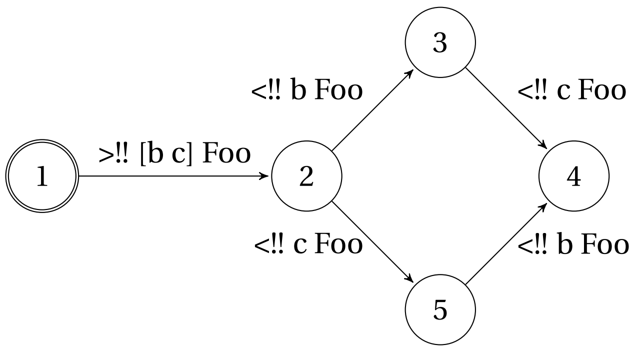
The figure above shows the Foo message first being sent and received, followed by the Bar message.

multicast Discourje also offers multicasting where a message is sent to multiple receivers, and the order in which the messages are received is not important. Notice the atomic interaction accepts a vector/array as receiver arguments. This means there is no special construct for multicast support, but it is integrated in the atomic interaction.

Listing 4.3: Parallelism

```
(-->> "Foo" "Alice" ["Bob", "Carol"])
```

The example above shows that two *Foo* interactions are merged into one interaction with multiple receivers. Note that the receivers might not receive the messages in the order in which they are defined and thus communication is *non deterministic*. The state machine representing this communication is the following:



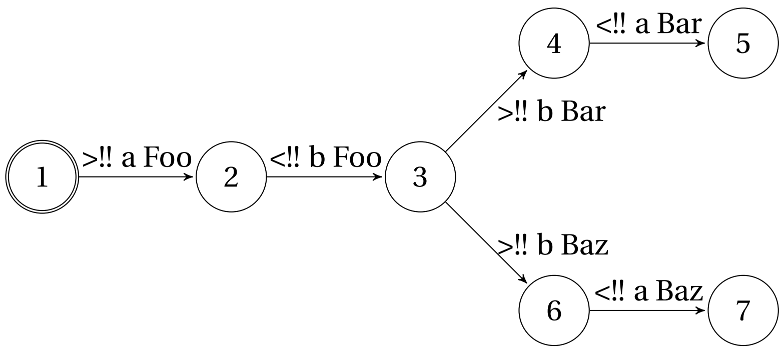
The figure above illustrates the non determinism of multicast. The *Foo* message is either first send and received by *Bob* and then *Carol*, or received by *Carol* first and then *Bob*. Sending is always done in the respective order of the defined communication pattern, but receiving might differ.

Branching Sometimes it is required to make a choice when specific conditions are met. Discourje offers branching through a choice construct identified by *choice, branch0, branch1, ..., branchN*

Listing 4.4: Branching

```
(-->> "Foo" "Alice" "Bob")
  (choice
    [(-->> "Bar" "Bob" "Alice")]
    [(-->> "Baz" "Bob" "Alice")])
```

The example above shows a choice where *Bob* can either send the *Bar* or *Baz* message to *Alice*. In order to allow Discourje to identify the correct branch, the label must be unique, i.e., there cannot be non-determinism. This means there cannot be duplicate labels in the first interaction of the branches.
The state machine representing the branching flow would be the following.



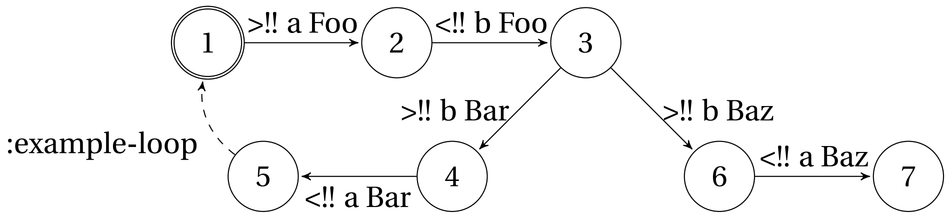
Note that this state machine is deterministic!

Recursion Discourje offers recursion as a looping mechanism that is *always executed at least once*. A recursion block is identified with a *rec, name* statement and a repetition is made when a *continue, name* statement is encountered (multiple continue statements are supported).

Listing 4.5: Recursion

```
(rec :example-loop
  (-->> "Foo" "Alice" "Bob")
  (choice
   [(-->> "Bar" "Bob" "Alice")
    (continue :example-loop)]
   [(-->> "Baz" "Bob" "Alice")]))
```

Listing 4.5 demonstrates a recursive protocol. With this protocol, *Alice* will send the *Foo* message to *Bob* and then *Bob* needs to make a choice. When *Bob* responds with a *Bar* message, the protocol will recur and *Alice* can again send the *Foo* message followed by *Bob's* choice. When *Bob* responds with a *Baz* message, the protocol will end.



Notice the loop from state 5 back to state 1 labeled :example-loop. The dashed arrow in this example is used to emphasize the loop. When a continue construct is encountered during execution of the protocol, the outgoing edges of the state where the continue points to are used for validation. This will be explained in detail in Chapter 6.

4.2. MODELING PROTOCOLS

The following section will describe the one and two buyer protocols using the Discourje DSL. The section will start off by illustrating the implementation of the one buyer protocol, followed by the two buyer protocol. Both examples will include the DSL definition and a state machine representing this definition.

Message exchange pattern: All defined interactions must be added to a protocol construct for Discourje to operate on (this construct will offer an interface, to for example query interactions, and will act as an abstraction). Since Clojure already has a macro called *defprotocol* and the notion of *protocols* as 'replacement' for interfaces, the Discourje DSL uses the term 'message exchange pattern' as a name for a communication pattern. A message exchange pattern can be created by the *mep* macro, which accepts at least 1 interaction with optional variadic interactions. The macro definition is as follows:

Listing 4.6: mep

```
(defmacro mep
  "Generate message exchange pattern aka protocol"
  [interactions \& more]
  `(->protocol [~interactions ~@more]))
```

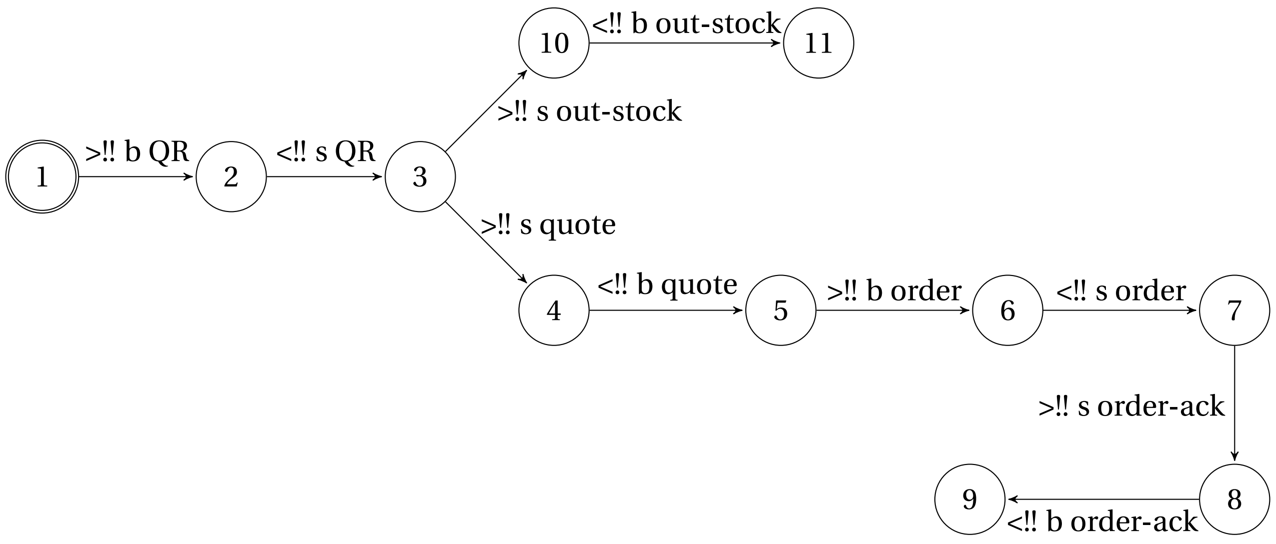
One buyer protocol Listing 4.7 defines the one buyer protocol in the Discourje DSL.

Listing 4.7: One buyer protocol

```
(def one-buyer-protocol
  (mep
    (-->> QuoteRequest "buyer" "seller")
    (choice
      [(-->> Quote "seller" "buyer")
       (-->> Order "buyer" "seller")
       (-->> OrderAcknowledgement "seller" "buyer")]
      [(-->> OutOfStock "seller" "buyer")])))
```

The protocol starts with an atomic interaction requesting the quote of the product. What follows is a choice by the seller. There are two branches in the choice: a branch for continuing the order and a branch for telling the product is out of stock. The branch for continuing the order starts with a quote message to send the price of the product back to the buyer. The buyer will respond with a message telling he wants to order the product. The seller will respond by sending an order acknowledgment. If the product is out of stock, the respective branch is chosen and the out of stock message is send to the buyer.

The state machine for the one buyer protocol looks as follows. Buyer and Seller are abbreviated to b and s respectively.



The state machine starts by sending and receiving the QuoteRequest (QR) message. Subsequently, there is a deterministic choice representing both branches. The branches model the atomic interactions defined in the protocol.

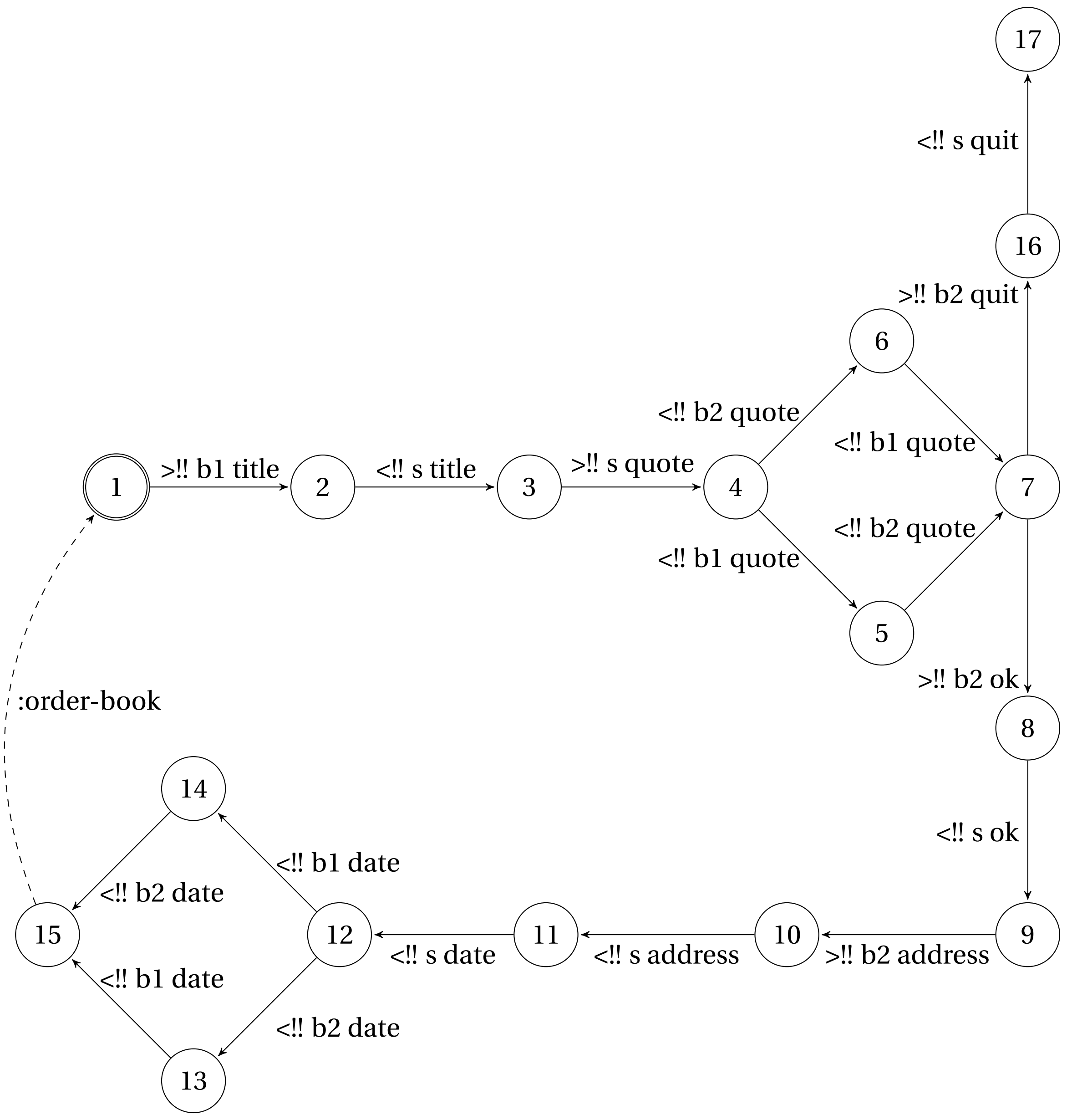
Two buyer protocol Listing 4.8 defines the two buyer protocol in the Discourje DSL.

Listing 4.8: Two buyer protocol

```
(def two-buyer-protocol
  (mep
    (rec :order-book
      (-->> "title" "buyer1" "seller")
      (-->> "quote" "seller" ["buyer1", "buyer2"])
      (-->> "quote-div" "buyer1" "buyer2")
      (choice
        [(-->> "ok" "buyer2" "seller")
         (-->> "address" "buyer2" "seller")
         (-->> "date" "seller" ["buyer2", "buyer1"])
         (continue :order-book)]
        [(-->> "quit" "buyer2" "seller")]))))
```

The protocol starts with a recursive construct named *:order-book* to allow for ordering more books when buyer2 agrees to contribute. The first interaction shows the Title messages being exchanged between buyer1 and seller. Then, the seller will send back the quote to buyer1 and buyer2, and buyer1 will send the quote-div message to buyer2. Now buyer2 has to make a choice to either send the quit message to the seller, ending the protocol, or sending the ok message and contribute to the book. When buyer2 chooses to contribute and sends his address, the seller will respond with the delivery date of the product to both Buyer1 and Buyer2. The last construct in the branch is a continue named :order-book, which recurs back to the definition of the recursive construct.

The state machine for the two buyer protocol looks as follows. Buyer1, Buyer2, and Seller are abbreviated to b1, b2 and s respectively.



5 API

This chapter describes the API for Discourje. The purpose of Discourje is to enable validation of communication and synchronization between threads by comparing any communication to a protocol specification. Chapter 4 explained how to create such a protocol specification, and the API must be able to use the specification at run-time.
To allow the run-time system to do this, a programmer must use special send and receive functions that incorporate validation against the protocol specification.
The first section explains three components that form the API. The second section shows a number of examples to demonstrate the API and how the API relates to Clojure.core.async. The chapter ends with a section on how to implement the one and two buyer protocols.

5.1. API COMPONENTS

The following section explains the purpose of the API components for Discourje. First the purpose of the validation component called the monitor will described. Second, infrastructure for communication will be explained. The section will end with the explanation of the communication abstractions. Full details of the monitor component and how validation works at run-time appear in Chapter 6.

Monitor The component responsible for the validation of all communication through Discourje is called the monitor. This component uses the defined protocol specification for validation. In this monitor there is only ***one*** *active* / *permitted* communication since a Discourje communication pattern (protocol) employs a clear sequential structure. When a communication is deemed valid, it is allowed to be performed, and after that, the monitor will activate the next communication as defined in the protocol. When it is invalid, the monitor will reject the communication.

Infrastructure Discourje requires some form of infrastructure for communication to occur. This infrastructure is made up pairwise unique channels between the participants defined by the DSL. This means that communication between A and B cannot interfere with the channel between A and C. Before using any of the communication abstractions, this infrastructure must be generated.

Communication abstractions The communication abstractions are functions to put and take values on/from channels. While communicating through these abstractions, every put/take is subjected to the monitor and its validation mechanism (which will be explained in full detail in Chapter 6).

5.2. DISCOURJE.CORE.ASYNC

Discourje's API has been designed to mimic the *Clojure.core.async* API to minimize the amount of changes required to migrate from Clojure.core.async to Discourje. The first paragraph will explain Discourje's features for channel creation and operations for putting and taking data from them. The second paragraph will cover configuration functions. This section ends with explaining message types, wildcards, and how to customize channel creation.

Discourje.core.async API The Discourje API consists of two groups of functions: process functions to interact with the channels (put or take values) and configuration functions to set up the infrastructure. Discourje offers replacements for a selected subset of operations from Clojure.core.async (see Chapter 3). The function names are identical, yet the input arguments types and return types differ.

Process macros

- ***(>!! & chan, value)*** Put a value on a *Discourje channel*(which encapsulates a core.async channel); will block if no buffer space is available.
- ***(<!! & chan label)*** Take value from a *Discourje channel* matching the label, will block if nothing is available. The label is used as a type identifier for the receive function.
- ***(thread & body)*** Execute the body on a different thread. This simply invokes the Clojure.core.async thread function and is added to the Discourje API for convenience.

Configuration macros

- ***(mep & interaction, & more)*** Generate a 'message exchange pattern' based on the interactions given. Interactions can be any combination of Discourje DSL macros.
- ***(add-infrastructure & message-exchange-pattern)*** Adds infrastructure to the defined message-exchange-pattern. This macro will generate the minimum number of channels required to execute the protocol properly, by performing static analysis on the message exchange pattern.
- ***(chan & sender, receiver, buf)*** Generate a *Discourje channel* between sender and receiver(s) with an optional buffer argument.
- ***(get-channel & sender, receiver, infrastructure)*** Obtain the channel from the specified sender to the specified receiver from the given infrastructure.
- ***(msg & label, content)*** Generate a message with the given label and data content.
- ***(get-content & msg)*** Get the content from a received message through Discourje's take function (<!!).

Listing 5.1: Discourje.core.async

```
 1 (def foo-bar-protocol
 2   (mep
 3     (-->> "Foo" "Alice" "Bob")
 4     (-->> "Bar" "Bob" "Alice")))
 5
 6 (def infra (add-infrastructure foo-bar-protocol))
 7 (def alice-to-bob (get-channel "Alice" "Bob" infra))
 8 (def bob-to-alice (get-channel "Bob" "Alice" infra))
 9
10 (defn alice []
11   (>!! alice-to-bob (msg "Foo" "Foo content"))
12   (println "Alice received: " (get-content (<!! bob-to-alice "Bar"))))
13
14 (defn bob []
15   (println "Bob received: " (get-content (<!! alice-to-bob "Foo")))
16   (>!! bob-to-alice (msg "Bar" "Bar content")))
17
18 (thread (alice))
19 (thread (bob))
20
21 =>Bob received:  Foo content
22 =>Alice received:  Bar content
```

Listing 5.1 shows an example. In the listing, lines 1 to 4 define a protocol to send 'Foo' and 'Bar' messages between Bob and Alice. This pattern will be used by the monitor to validate the run-time communications. Lines 6 to 8 generate the infrastructure. Notice that the 'foo-bar-protocol' is added to the add-infrastructure as input argument. When Discourje generates the infrastructure, it internally adds the monitor to each channel. Lines 7 and 8 define two constants through which the channels between Alice and Bob (automatically generated on line 6) can be accessed. These channels are then used in the alice and bob functions to communicate.

Message types Every piece of data communicated through Discourje is wrapped in an object called a *message*. A message has the following interface & data structure:

Listing 5.2: Message

```
(defprotocol sendable
  (get-label [this])
  (get-content [this]))

(defrecord message [label content]
  sendable
  (get-label [this] label)
  (get-content [this] content))
```

The message data structure has two fields: a label and the message content. All communication done through Discourje involves messages of this type to be able to identify and monitor the data being put or taken from channels.

Specifically, the *label* specified by the message must match the label of the active interaction in order for the monitor to permit the communication. The message *content* holds the actual data.
A new message can be created through the API by the *(msg & label, content)* macro, but Discourje is also able to generate messages automatically.

When 'raw' data (data that does not implement the *sendable* interface) is sent through

Discourje, a new message with the type of the raw data as label and the raw data as content will be generated.

Listing 5.3: Generating

```
(def protocol
  (mep
    (-->> java.lang.String "Alice" "Bob")))

Raw data input:
(>!! alice-to-bob "Foo")

Converted into:
(>!! alice-to-bob (msg java.lang.String "Foo"))
```

Listing 5.3 demonstrates how Discourje converts the raw data string *"Foo"* into a new message with label *java.lang.String* and content *"Foo"*. An important thing to note here is that the protocol used for validation must check for the label *java.lang.String* in the atomic interaction too.
The next section demonstrates how to use data types as message labels in the one buyer protocol and default messages in the two buyer protocol.

Wildcards Discourje offers the option to use a ‘nil’ label as a wildcard for communication in the take (<!!) function. The reasons behind this are the following: in order to allow a developer to gradually migrate from Clojure.core.async to Discourje, the labels for the take function are not required, since the Clojure.core.async syntax does not require this. Thus, The adaption effort for the source code to migrate to Discourje is initially as minimal as possible; then incrementally, wildcards can be refined to actual labels, by need. Another reason to use wildcards is that a protocol that only uses, for example, integers as message labels should not require explicitly stated labels since they are all identical. This means they do not add any value to the validation. The following example demonstrates this:

Listing 5.4: Wildcards config

```
(def integer-protocol
  (mep
    (-->> java.lang.Integer "Alice" "Bob")
    (-->> java.lang.Integer "Bob" "Alice")
    (-->> java.lang.Integer "Alice" "Carol")
    (-->> java.lang.Integer "Carol" "Alice")))

(def infra (add-infrastructure integer-protocol))
(def alice-to-bob (get-channel "Alice" "Bob" infra))
(def alice-to-carol (get-channel "Alice" "Carol" infra))
(def carol-to-alice (get-channel "Carol" "Alice" infra))
(def bob-to-alice (get-channel "Bob" "Alice" infra))
```

Listing 5.4 specifies a protocol containing only interactions with Java integers as their labels exchanged between Alice, Bob and Carol.

Listing 5.5: Before wildcards

```
(defn alice []
  (>!! alice-to-bob (msg java.lang.Integer 1))
  (println "Alice received: " (get-content (<!! bob-to-alice java.lang.Integer)))
  (>!! alice-to-carol (msg java.lang.Integer 3))
  (println "Alice received: " (get-content (<!! carol-to-alice java.lang.Integer))))

(defn bob []
  (println "Bob received: " (get-content (<!! alice-to-bob java.lang.Integer)))
  (>!! bob-to-alice (msg java.lang.Integer 2)))
```

```

(defn carol []
  (println "Carol received: " (get-content (<!! alice-to-carol java.lang.Integer)))
  (>!! carol-to-alice (msg java.lang.Integer 4)))
```

Listing 5.5 shows functions for Alice, Bob, and Carol to send the messages back and forth. Each send function must specify a message (msg) labeled java.lang.integer. The take operations must be labeled in the same manner. This explicit labeling of messages feels redundant since the protocol only communicates java.lang.integers. To remove this redundancy, wildcards can be enabled and the communication will be simplified.

Listing 5.6: After wildcards

```

(enable-wildcard)

(defn alice []
  (>!! alice-to-bob 1)
  (println "Alice received: " (get-content (<!! bob-to-alice)))
  (>!! alice-to-carol 3)
  (println "Alice received: " (get-content (<!! carol-to-alice))))

(defn bob []
  (println "Bob received: " (get-content (<!! alice-to-bob)))
  (>!! bob-to-alice 2))

(defn carol []
  (println "Carol received: " (get-content (<!! alice-to-carol)))
  (>!! carol-to-alice 4))

(disable-wildcard)
```

Listing 5.6 shows wildcards being enabled with the *(enable-wildcard)* function. The functions for Alice, Bob and Carol are then able to communicate without explicitly stating the label. The implementation in Listing 5.6 also makes use of *typed messages*, explained in the previous paragraph.

Note that wildcards can be enabled at any time, and thus if required any protocol is able to communicate without explicit labels.

Custom channels Discourje uses Clojure.core.async channels as channel abstractions. These Clojure.core.async channels allow for configuration of their buffer size. When Discourje generates the infrastructure, it generates Clojure.core.async channels with a buffer size of 1, but that may not always be desired. Sometimes a programmer may require custom buffer sizes for one or more channels.

Discourje supports custom channels to be added to the infrastructure. The programmer is then able to configure the buffer size for each channel individually.

5.3. EXAMPLES

This section shows the one and two buyer examples implemented in the Discourje.core.async API. The one buyer example uses Java objects as message types to demonstrate Java interopability and usage of raw data. The two buyer example uses Discourje messages labeled by text.

One buyer protocol The first step to implement the one buyer protocol is to define its communication pattern as presented in Chapter 4. Next, Discourje needs infrastructure for communication.

Listing 5.7: One buyer Infrastructure

```
(def infra (add-infrastructure buy-goods))

(def buyer-to-seller (get-channel "buyer" "seller" infra))
(def seller-to-buyer (get-channel "seller" "buyer" infra))
```

Listing 5.7 shows Discourje generating the infrastructure (no custom channel configurations for buffer size), followed by two simple queries to get the two created channels based on sender and receiver pair. These channels allow for identification of the correct sender and receivers at run-time. This differs from Clojure.core.async where a programmer can use any channel to communicate.

Listing 5.8: Generating

```
(def product (doto (Book.) (.setName "The Joy of Clojure")))

(defn in-stock? "return a 50% change true in stock" [book]
  (let [in-stock (== 1 (rand-int 2))]
    (println (format "%s is in stock: %s"
    (.getName (.getProduct book)) in-stock))
  in-stock))
```

Then a product is defined, in this case, a book called *The Joy of Clojure*. Also an in-stock function is defined, which simply returns true or false based on a 50% chance.

Listing 5.9: One buyer - Buyer logic

```
(defn buyer "Logic representing Buyer" []
  (>!! buyer-to-seller (doto (QuoteRequest.) (.setProduct product)))
    (let [quote (get-content (<!! seller-to-buyer Quote))]
          (if (.isInStock quote)
          (do
            (>!! buyer-to-seller (doto (Order.) (.setProduct product)
            (.setQuote quote)))
            (println (.getMessage (get-content (<!! seller-to-buyer
            OrderAcknowledgement)))))
          (println "Book is out of stock!"))))
```

Next the buyer function is defined, which starts with a Discourje-put on the buyer-to-seller channel which allows the monitor to check conformance with the active interaction. Then, immediately, a Discourje-take on the seller-to-buyer channel is performed. If the book is in stock, the buyer calls a Discourje-put on the buyer-to-seller channel to order the product, and a Discourje-take to wait for the order acknowledgement.

Differences with Clojure.core.async again are the concretely specified put and take communications. Clojure.core.async simply allows any data to be put or taken from any channel. Discourje does not allow this, and requires all communication to comply with the message exchange pattern for validation.

Listing 5.10: One buyer - Seller logic

```
(defn seller "Logic representing the Seller" []
  (if (in-stock? (get-content (<!! buyer-to-seller QuoteRequest)))
  (do (>!! seller-to-buyer (doto (Quote.) (.setInStock true)
  (.setPrice 40.00) (.setProduct product)))
          (let [order (get-content (<!! buyer-to-seller Order))]
          (>!! seller-to-buyer (doto (OrderAcknowledgement.)
          (.setOrder order)
          (.setMessage
          (format "order-acknowledgement: Order for product: %s
          confirmed at price: $%s"
          (.getName (.getProduct order))
          (.getPrice (.getQuote order)))))))
  (>!! seller-to-buyer (doto (Quote.) (.setInStock false) (.setPrice 0)
  (.setProduct product)))))
```

The seller function starts with a Discourje-take for the message labeled QuoteRequest from the buyer-to-seller channel. If the product is in stock the seller will Discourje-put the quote of the product on the seller-to-buyer channel and subsequently starts a Discourje-take for an Order labeled message, and ends with a Discourje-put of the OrderAcknowledgement. If the product is not in stock, the seller will Discourje-put the OutOfStock message on the seller-to-buyer channel.

Again the differences here are the specific Discourje put and take operations to specify which message is being communicated between which participants.

Listing 5.11: One buyer start

```
(thread (buyer))
(thread (seller))
```

Lastly, both the buyer and seller functions are started on different threads, which is exactly the same for both Discourje as Clojure.core.async.

Two buyer protocol Just as the with the one buyer protocol, the implementation starts by defining the communication pattern which is defined in Chapter 4.

Listing 5.12: Two buyer Infrastructure

```
(def infrastructure (add-infrastructure (two-buyer-protocol)))
```

Next, the infrastructure is generated. Note that the required channels are not queried globally upfront as in the one buyer protocol. In this case, all participants query their needed channels locally, demonstrated in the next three listings.

Listing 5.13: Two buyer - Buyer1 logic

```
(def books ["The Joy of Clojure"
            "Mastering Clojure Macros"
            "Programming Clojure"])

(defn generate-book "generate book title" []
  (first (shuffle books)))

(defn quote-div "Generate random number with max, quote value" [quote]
  (log-message (format "received quote: %s" quote))
  (let [randomN (+ (rand-int quote) 1)]
        (log-message "QD = " randomN)
  randomN))

(defn order-book "order a book from buyer1's perspective" [infra]
  (let [b1-s (get-channel "buyer1" "seller" infra)
        s-b1 (get-channel "seller" "buyer1" infra)
        b1-b2 (get-channel "buyer1" "buyer2" infra)]
    (>!! b1-s (msg "title" (generate-book)))
    (let [quote (<!!! s-b1 "quote")
          div (quote-div (get-content quote))]
      (do
        (>!! b1-b2 (msg "quote-div" div))
        (when (<!!! s-b1 "date")
          (order-book infra))))))
```

The order-book function in Listing 5.13 is the implementation of the protocol from buyer1's perspective. It starts by querying the channels needed for communication. Then it calls a Discourje-put on the b1-s (buyer1-to-seller) channel with a message labeled title and content the result of the generate book function. It then starts a Discourje-take function to await the quote from the seller. When the quote is received, buyer1 will Discourje-put the quote-div message on the b1-b2 (buyer1-to-buyer2) channel, and then start a Discourje-take for the repeat message.

Listing 5.14: Two buyer - Buyer2 logic

```
(defn contribute? [quote div]
        (log-message (format "received quote: %d and div: %d,
        contribute      = %s"
        quote div (>= (* 100 (float (/ div quote))) 50)))
  (>= (* 100 (float (/ div quote))) 50))

(defn generate-address "generates the address" []
"Open University, Valkenburgerweg 177, 6419 AT, Heerlen")

(defn order-book "Order a book from buyer2's perspective" [infra]
  (let [s-b2 (get-channel "seller" "buyer2" infra)
        b1-b2 (get-channel "buyer1" "buyer2" infra)
        b2-s (get-channel "buyer2" "seller" infra)
        quote (get-content (<!!! s-b2 "quote"))
        quote-div (get-content (<!! b1-b2 "quote-div"))]
    (if (contribute? quote quote-div)
      (do (>!! b2-s (msg "ok" "ok"))
          (>!! b2-s (msg "address" (generate-address)))
          (let [date (<!!! s-b2 "date")]
            (log-message (format "Thank you, I will put %s in my agenda!"
            (get-content date)))
            (order-book infra)))
      (>!! b2-s (msg "quit" "Price to high!")))))
```

The order-book function in Listing 5.14 starts by querying the required channels for buyer2 to operate on. It immediately starts with a Discourje-take for the quote-div message on be b1-b2 (buyer1-to-buyer2) channel, since this is the first communication where buyer2 is involved. Then buyer2 will determine if he wants to contribute to the order: if not buyer2 will Discourje-put the message labeled quit on the b2-s (buyer2-to-seller) channel, signifying the end of the communication. If he does decide to contribute to the order: buyer2 will Discourje-put a message labeled ok with his address as content on the b2-s channel. Buyer2 then starts a Discourje-take on the s-b2 (seller-to-buyer2) channel to await the date from seller. When buyer2 has received the date, he will Discourje-put the repeat message on the b2-s channel.

Listing 5.15: Two buyer - Seller logic

```
(defn quote-book [title]
  (log-message (format "received title: %s" title))
  (+ (rand-int 30) 1))

(defn get-date [days]
  (let [cal (Calendar/getInstance) d (new Date)]
        (doto cal
          (.setTime d)
          (.add Calendar/DATE days)
          (.getTime))))

```

```
(defn get-random-date [maxRange]
  (get-date (+ (rand-int maxRange) 1)))

(defn- end-reached [quit]
  (log-message (format "Protocol ended with: %s" quit)))

(defn order-book "Order book from seller's perspective" [infra]
  (let [b1-s (get-channel "buyer1" "seller" infra)
        s-b1 (get-channel "seller" "buyer1" infra)
        s-b2 (get-channel "seller" "buyer2" infra)
        b2-s (get-channel "buyer2" "seller" infra)
        title(<!! b1-s "title")]
    (>!! [s-b1 s-b2] (msg "quote" (quote-book (get-content title))))
    (let [choice-by-buyer2 (<!! b2-s ["ok" "quit"])]
      (cond
        (= "ok" (get-label choice-by-buyer2))
        (do
          (println (format "Order confirmed, will send to address: %s"
          (get-content (<!!! b2-s "address"))))
          (>!! [s-b2 s-b1] (msg "date" (get-random-date 5)))
          (order-book infra))
        (= "quit" (get-label choice-by-buyer2))
          (end-reached "Quit!")))))
```

The order-book function in Listing 5.15 starts by querying the channels and immediately starts a Discourje-take labeled title to await the message being sent by buyer1. The seller then responds to buyer1 with a message labeled quote on the s-b1 (seller-to-buyer1) channel. He then starts a Discourje-take on the b2-s (buyer2-to-seller) channel to wait for either the message labeled ok or quit (notice the seller awaits multiple communications through a vector). When he receives the ok message, the seller will Discourje-put the message labeled date to buyer2. Then the seller awaits the repeat message from buyer2, and subsequently Discourje-puts the repeat message on the s-b1 channel to signify buyer1 to order a new book.

Listing 5.16: Two buyer start

```
(thread (b1/order-book infrastructure))
(thread (b2/order-book infrastructure))
(thread (s/order-book infrastructure))
```

Lastly the buyer1, buyer2 and seller functions are started on different threads. Note that the infrastructure is given as input parameters in order to allow the participants to query their channels.

Differences with Clojure.core.async The main differences between Clojure.core.async and Discourje is the specification of channels, put and take operations. The put and take abstractions in Clojure.core.async simply put or take ***any*** data on or from channels. Discourje requires a programmer to specifically state which data is being put or taken, in terms of a label. The message exchange pattern that is validated by the monitor must also be explicitly coupled to the infrastructure (channels). The Discourje DSL adds identification on channels by sender and receiver pair that are required to be used for communication. Clojure.core.async channels do not specifically state who or what is allowed to interact with it. This adds more protection against communication mismatches. Another major difference with Clojure.core.async is the return value of the take function. Discourje returns a message object which can be used as an abstraction for making choices. Data returned by Clojure.core.async can be of any type which introduces additional complexity, for example: exception handling when taken data is not of the expected type.

6
Run-time system

This chapter describes the run-time system for Discourje. The first section will describe the purpose of the monitor in detail, and how it validates each type of communication construct of the Discourje DSL. The second section will describe protocol initialization and setup of the infrastructure (channels). The third section describes the monitor and how it behaves during synchronization. The fourth section describes logging levels and an example implementation of the two buyer protocol.

6.1. Validation engine

At the heart of the run-time system is the validation engine. This validation engine takes the protocol specification defined in the Discourje DSL and turns it into the monitor that is used for validation of communications.
There are several steps involved in this process, including generating the infrastructure, but the most important step related to validation is generating the state machine for communication. Specifically, the run-time system will parse the protocol specification defined in the Discourje DSL into graphs that look very similar to the state machines in Chapter 4. There are however some special cases for *choice, rec, and continue* which will be explained in the following section.

The monitor will validate all communication by simulating this state machine. It does this by *flagging a single state to be active*. The outgoing edge(s) of the active state will be the *only* permitted communication(s), which means there may only be *one* communication at one time. Note that a choice is deterministic, and thus there can be multiple edges/communications active, but only one edge is ever taken.

Let's take a detailed look at the recursion example in Chapter 4:

Listing 6.1: Recursion

```
(rec :example-loop
  (-->> "Foo" "Alice" "Bob")
  (choice
   [(-->> "Bar" "Bob" "Alice")
    (continue :example-loop)]
   [(-->> "Baz" "Bob" "Alice")]))
```

This example *specifically* defines a communication that starts by sending the Foo message from Alice to Bob. Bob *must* receive the message *before* making the choice to either send the Bar or Baz message back to Alice. When Bob decides to send the Bar message to Alice, she *must* receive it *before* the protocol is able to recur back to the beginning. The validation engine in the run-time will ensure that the communications indeed happen in this way, or else it reports a violation.

6.1.1. MONITOR REQUIREMENTS

While designing the monitor there are a number of requirements that were followed and implemented:

- **R1:** The monitor must be passive, meaning it may not alter messages or change the behavior of the underlying Clojure.core.async code.
- **R2:** Adding the monitor must invalidate an incorrect Clojure.core.async communication.
- **R3:** Adding the monitor must validate a correct Clojure.core.async communication.

The monitor must be a passive component which does not alter message content or change the behavior of underlying Clojure.core.async code. For example, forcing a delay on a currently forbidden put or take until it would be permitted (R1). Also, the monitor must invalidate an incorrect communication and show notifications or throw exceptions when it does not comply with the protocol (R2). The monitor must not invalidate a correct communication when it does comply with the protocol (R3).

6.2. PROTOCOL SETUP

Setting up a protocol requires a number of steps. The protocol data structure generated by the DSL will need to be coupled to the communication infrastructure. Discourje is able to generate the channels for the infrastructure or a programmer can supply his own set of channels. The following paragraphs will explain this in detail.

Generating infrastructure: To generate the infrastructure, Discourje will first parse the entire protocol to find all communication pairs, mapped by source to sink. This means Discourje only generates channels for the pairs that actually occur in the protocol definition. This greatly reduces the number of channels compared to simply generating channels for all different pairs, i.e. the cartesian product of protocol participants. Based on these pairs, channels will be generated and embedded in the following Discourje data structure:

Listing 6.2: Channel definition

```
(defprotocol transportable
  (get-provider [this])
  (get-consumer [this])
  (get-chan [this])
  (get-monitor [this])
  (get-buffer [this]))

(defrecord channel [provider consumers chan buffer monitor]
  transportable
  (get-provider [this] provider)
  (get-consumer [this] consumers)
  (get-chan [this] chan)
  (get-monitor [this] monitor)
  (get-buffer [this] buffer))
```

Discourje adds sender and receiver pairs to the channels to allow for identification of a specific channel by participants. There is also a buffer parameter of the channel record which is used to set the buffer size of the underlying Clojure.core.async channel it can also be used to check the buffer size of a channel since Clojure.core.async does not provide a way to check this. One last important note concerning these channel pairs is that they are not commutative, meaning a channel with Alice as sender and Bob as receiver is not the same as a channel with Bob as sender and Alice as receiver.

Custom Infrastructure: When there are special requirements for channels, a programmer can supply his own custom channels as infrastructure. The current version of Discourje allows for customization of the buffer size of channels, but may allow for more configuration options in the future. When a programmer supplies his own channels, they will be validated for correctness, meaning if they comply with the transportable interface (defprotocol) in Listing 6.2, Discourje will throw an exception when an incorrect channel configuration is given.

6.2.1. STATE MACHINE GENERATION

Not only does the run-time system of Discourje need access to the communication infrastructure, but also the interactions need to follow the flow specified in the protocol definition. Upon generating the infrastructure, Discourje will link the communications specified in the protocol definition to one another to create a state machine with some special cases for choice, rec and continue statements.
To do this, the following interfaces are used by the run-time system.

Listing 6.3: Interfaces

```
(defprotocol idable
(get-id [this]))

(defprotocol linkable
(get-next [this]))
```

These interfaces specify that each data structure must have an identifier, plus a get next operation to go to the next interaction. For all interactions, the ID generation is done through an UUID generator. However, the generation of the next is different in choice, rec and continue. The following paragraphs explain the generation process. This section will end by showing the data structure for the state machine for the one and two buyer protocols.

Atomic Interaction: Before discussing how the special cases,*choice, rec, and continue,* identify their next linked interaction, linking of atomic interactions must be explained. The atomic interaction data structure looks as follows:

Listing 6.4: Atomic interaction definition

```
(defrecord interaction [id action sender receivers next]
  idable
  (get-id [this] id)
  interactable
  (get-action [this] action)
  (get-sender [this] sender)
  (get-receivers [this] receivers)
  linkable
  (get-next [this] next)
  stringify
  (to-string [this] (format "Interaction - Action: %s, Sender: %s,
   Receivers: %s" action sender receivers)))
```

The listing above shows its parameters: The interaction Id, an action to label the interaction e.g. the sender and receiver(s), and a next interaction, which is the ID of another interaction. The atomic interaction also implements a number of interfaces, like the Idable interface,the linkable interface, the interactable interface (to be able to abstract the operations of getting the action, sender and receivers), and an additional stringify interface for logging purposes.

The linking process in regard to the atomic interaction is simple: *the next interaction defined in the protocol data structure will be set as 'next'.*

Listing 6.5: Atomic interaction linking

```
Before linking:

(def protocol
  (mep
    (-->> "Foo" "Alice" "Bob")
    (-->> "Bar" "Bob" "Alice")
    (-->> "Baz" "Alice" "Bob")))

After linking:

(def protocol
  (mep
    (-->> id: 1, "Foo" "Alice" "Bob" next: 2)
    (-->> id: 2, "Bar" "Bob" "Alice" next: 3)
    (-->> id: 3, "Baz" "Alice" "Bob" next: nil)))
```

The linking process couples the interaction through their ids ending with a *nil* in the last interaction, signifying the end of the protocol.

Choice: The choice construct is linked in a slightly different pattern since the run-time system will need to compare any possible interaction against all specified branches. The data structure for a choice looks as follows:

Listing 6.6: Branching definition

```
(defrecord branch [id branches next]
  idable
  (get-id [this] id)
  branchable
  (get-branches [this] branches)
  linkable
  (get-next [this] next)
  stringify
  (to-string [this] (format "Branching with first branches - %s"
   (apply str (for [b branches] (format "[ %s ]"
   (to-string (first b))))))))
```

The listing above shows the branch data structure requiring an id, branches (which are vectors of interactions) and a next identifier. The branch also implements the idable, linkable and stringify interfaces, just as the atomic interaction, plus an additional interface branchable to get the branches from the data structure. The linking process for a Branch looks as follows:

Listing 6.7: Branching linking

```
Before linking:

(def protocol
  (mep
    (-->> "Foo" "Alice" "Bob")
    (choice
      [(-->> "Bar" "Bob" "Alice")]
      [(-->> "Baz" "Bob" "Alice")])
    (-->> "Qux" "Alice" "Bob")))

After linking:

(def protocol
  (mep
    (-->> id: 1, "Foo" "Alice" "Seller" next: 2)
    (choice id: 2
      [(-->> id: 3, "Bar" "Bob" "Alice" next: 5)]
      [(-->> id: 4, "Baz" "Bob" "Alice" next: 5)])
    (-->> id: 5, "Qux" "Alice" "B" next: nil)))
```

Important here is that the interaction identified with the label 'Foo' links directly to the Choice, and the last interactions in each branch both link to the 'Qux' interaction. Note that nesting of branches is supported, meaning that the last interaction in a branch could potentially link to another branch, choice, or rec id.

Rec & continue: The linking process of a rec and its continue statement follow a similar pattern as choice where an interaction is directly linked to the rec and the continue next is also directly linked to the rec. The rec and continue data structures look as follows.

Listing 6.8: Rec & continue definition

```
(defrecord recursion [id name recursion next]
  idable
  (get-id [this] id)
  namable
  (get-name [this] name)
  recursable
  (get-recursion [this] recursion)
  linkable
  (get-next [this] next)
  stringify
  (to-string [this] (format "Recursion name: %s, with first
   in recursion- %s" name (to-string (first recursion)))))

(defrecord recur-identifier [id name option next]
  idable
  (get-id [this] id)
  namable
  (get-name [this] name)
  identifiable-recur
  (get-option [this] option)
  linkable
  (get-next [this] next)
  stringify
  (to-string [this] (format "Recur-identifier - name: %s,
   option: %s" name option)))
```

Again, both the *recursion* and *recur-identifier* implement the idable, linkable and stringify interface. They also both implement an interface called namable to get the name of a recursion statement to be able to match with a continue. After linking the interactions, the protocol looks as follows:

Listing 6.9: Rec & continue linking

```
Before linking:

(mep
  (rec :example-loop
    (-->> "Foo" "Alice" "Bob")
    (choice
      [(-->> "Bar" "Bob" "Alice")
       (continue :example-loop)]
      [(-->> "Baz" "Alice" "Bob")])))

After linking:

(mep
  (rec id: 1, :example-loop
    (-->> id: 2, "Foo" "Alice" "Bob" next: 3)
    (choice id: 3,
      [(-->> id: 4, "Bar" "Bob" "Alice" next: 5)
       (continue id: 5, :example-loop next: 1)]
      [(-->> id: 6, "Baz" "Bob" "Alice" next: nil)])
     next: nil))
```

Important in this listing is that the continue statement refers back to the rec with the same name, in this case *:example-loop* and could potentially be nested in choices or other rec statements.

6.2.2. ONE-AND-TWO BUYER PROTOCOLS

Now that all different types of constructs are described the one and two buyer protocol are given in Listings 6.10 and 6.11.

One buyer protocol

Listing 6.10: One buyer protocol linking

```
Before linking:

(def protocol
  (mep
    (-->> QuoteRequest "buyer" "seller")
    (choice
      [(-->> Quote "seller" "buyer")
       (-->> Order "buyer" "seller")
       (-->> OrderAcknowledgement "seller" "buyer")]
      [(-->> OutOfStock "seller" "buyer")])))

After linking:

(def protocol
  (mep
    (-->> id: 1, QuoteRequest "buyer" "seller" next: 2)
    (choice id: 2,
      [(-->> id: 3, Quote "seller" "buyer" next: 4)
       (-->> id: 4, Order "buyer" "seller" next: 5)
       (-->> id: 5, OrderAcknowledgement "seller" "buyer"]
      [(-->> id: 6, OutOfStock "seller" "buyer" next: nil)])
     next: nil))
```

Two buyer protocol

Listing 6.11: Two buyer protocol linking

```
Before linking:

(def two-buyer-protocol
  (mep
    (rec :order-book
      (-->> "title" "buyer1" "seller")
      (-->> "quote" "seller" ["buyer1" "buyer2"])
      (-->> "quote-div" "buyer1" "buyer2")
      (choice
        [(-->> "ok" "buyer2" "seller")
         (-->> "address" "buyer2" "seller")
         (-->> "date" "seller" ["buyer2" "buyer1"])
         (continue :order-book)]
        [(-->> "quit" "buyer2" "seller")]))))

After linking:

(def two-buyer-protocol
  (mep
    (rec id: 1, :order-book
      (-->> id: 2, "title" "buyer1" "seller" next: 3)
      (-->> id: 3, "quote" "seller" ["buyer1" "buyer2"] next: 4)
      (-->> id: 4, "quote-div" "buyer1" "buyer2" next: 5)
      (choice id: 5,
        [(-->> id: 6, "ok" "buyer2" "seller" next: 7)
         (-->> id: 7, "address" "buyer2" "seller" next: 8)
         (-->> id: 8, "date" "seller" ["buyer2" "buyer1"] next: 9)
         (continue id: 9, :order-book next: 1)]
        [(-->> id: 10, "quit" "buyer2" "seller" next: nil)]))
      :next: nil))
```

6.3. Protocol Validation

This section will explain the run-time validation logic. The section starts by explaining when validation takes place, followed by the specifics for each interaction type, starting with the atomic interaction.

Timing The timing of validation is similar for sending (put) and receiving (take) data through the Discourje API: both validations take place just *before* putting or taking the data on or from the underlying Clojure.core.async channel. In order to implement this behavior and replicate Clojure.core.async the put and take is extended with custom buffer checking.

Since Discourje is built as a layer on top of Clojure.core.async timing of validation must be added in such a way that it does not change Clojure.core.async's behavior (see Section 6.1.1 on the monitor requirements). Discourje will delay validation of a put operation until there is free space in the buffer. This means it will postpone the validation when the buffer is full, yet still supports the blocking behavior Clojure.core.async offers.
The take operation is also custom since Clojure.core.async does not allow for 'peeking' on channels. Peeking a channel is needed since Discourje needs to compare the the label of the current take operation with label of the next the message on the channel. When this validation fails, the message must not be taken from the channel but stay there. Additionally, there is a second variant of the take operation which delays/blocks the function from returning the taken value until *all* recipients of a multicast have successfully taken. This behavior is needed when the protocol has
communications where participants are unable to know when another receiver has taken a value. For example: In the two buyer protocol, the *Quote* message is sent by the seller to both buyer 1 and 2. When buyer1 receives this message, he will immediately send the *quote-div* message to buyer2. However, buyer2 might not yet have received the *quote* message

from the seller, resulting in an inadmissible communication.

Finally, only the **take** operation is responsible for switching the active interaction to the next active state. *The take operation signifies an interaction being complete.*

6.3.1. PUT VALIDATION

Before Discourje allows data to be put on an underlying Clojure.core.async channel, it verifies if the data coming in satisfies the *message (sendable)* datatype. If it does not, a new message will be generated with the type of the incoming data as label (see 'message types' in Chapter 5 for more information).
The second step in the validation of the put operation is to check whether the buffer is full. If so the put operation blocks until free buffer space is available.

Once this is so, the validation engine checks if data is being put in parallel as part of a multicast. If so, Discourje first checks if the supplied channels all share the same sender:

Listing 6.12: Share sender

```

(def alice-to-bob (get-channel "alice" "bob" infra))
(def bob-to-alice (get-channel "bob" "alice" infra))
(def alice-to-carol (get-channel "alice" "carol" infra))

faulty put:
(>!! [alice-to-bob bob-to-alice] (msg "faulty" "faulty"))

success put:
(>!! [alice-to-bob alice-to-carol] (msg "success" "success"))
```

Listing 6.12 shows two put operations. The first operation tries to put in parallel but the senders in the channels are not equal. Discourje will log or throw an exception when this occurs. The second multicast put is permitted since the channels share the same sender.
Following this, it is checked if all channels that are used in a multicast are monitored by the same object, to make sure channels between monitors cannot be shared. Sharing channels between different monitors would render the validation engine incorrect.
The last check for multicast is checking whether the individual puts are allowed by the active monitor. This check involves an intricate mechanism that differs from the validation approach for atomic interactions, choice, rec and continue; it is explained in detail in a subsequent section.

For non-multicast puts, the validation engine will immediately validate the channel and message against the current state of the monitor. If it passes the validation, the validation engine will allow the put on the channel.

6.3.2. TAKE VALIDATION

The take validation is less complex than the put validation. One of the reasons for this is that Discourje does not allow taking from multiple channels at once. This takes away a lot of complexity compared to the put operation.
As described earlier, the take on the channel will peek the next message on the channel before activating the validation mechanism check whether the take is valid. When there is no value in the channel buffer, the take operation will block until something is available

before peeking.
The take operation validates the input channel and supplied label against the current state of the monitor. When it succeeds, the active monitor will first be switched to the next, before returning the taken result data as output.

6.3.3. MONITOR

The monitor is the object used at run-time to validate all communication. The data structure looks as follows:

Listing 6.13: Monitor

```
(defrecord monitor [id interactions active-interaction]
  monitoring
  (get-monitor-id [this] id)

  (get-active-interaction [this] @active-interaction)

  (apply-interaction [this sender receivers label]
  (apply-interaction-to-mon sender receivers label active-interaction
   interactions))

  (valid-interaction? [this sender receivers label]
  (is-valid-communication? sender receivers label @active-interaction
   interactions)))
```

Importantly, the monitor has *mutable state*. The *active-interaction* is implemented as a Clojure Atom which is a mutable data type protected by *software transactional memory*. The active-interaction represents the interaction Discourje allows to be invoked next.

6.3.4. INTERACTION VALIDATION

The validation differs for each Discourje DSL construct, with the simplest being the atomic interaction. Again, special cases involve the choice, rec and continue. This section will first explain how an atomic interaction is validated followed by explanations for choice, rec and continue.

Validating: Atomic Interaction The verification of any interaction starts by asking the monitor to compare the type of the input data to the label in the active interaction. When the active interaction is an atomic interaction, the validation mechanism employs the following process: The *valid-interaction?* function of the monitor requires a *sender* and *receiver(s)* pair. These are extracted from the supplied channel(s) to the put and take operations. Next, the *label* is needed for identification of a put or take. Lastly, the *active interaction* is used for validation. The function signature is the following:

Listing 6.14: valid-interaction function signature

```

(valid-interaction? [this sender receivers label]
  (is-valid-communication? sender receivers label
        @active-interaction interactions))
```

The *is-valid-communication?* function enforces the following check to its input parameters and **all** checks must pass:

- **Sender:** Is the supplied sender equal to the sender in the active interaction?

- **Receiver:** Checks whether the receiver is a collection; If so, it means that sending is done for a multicast and thus the input receiver must exist in the active interaction receivers vector. When receiver is not a collection, the receiver must match the receiver in the active interaction.
- **Label:** First checks whether the input label is a collection; If so, it means listening on the same channel for multiple message labels (choice, see seller logic of the two buyer protocol in Chapter 5). It checks whether the active interaction complies with the input labels. When the input label is not a collection, the label must match with the active interaction.

When this check succeeds, the communication is valid and the operation is allowed to proceed. When the operation, is a take the next monitor is activated. For the atomic interaction, this is straightforward; since all interactions in the monitor are linked as a graph (see the previous section) the monitor is able to query the next interaction based on the active interaction *next:<id>*.

An **important** implementation detail here is that switching to a new active monitor is done through Clojure's *swap!* function[1]. Since the active interaction is mutable, Clojure only allows manipulating this data through transaction-like functions of the STM. The swap! function tries to 'update' the value in the atom object; in this case the active interaction. It will retry this function until it succeeds, so the logic executed inside the swap! must be free of side effects. For example, adding a println to the swap! logic will print for every retry.

In case of the an atomic interaction, the next interaction is queried and supplied to the swap! function. It then replaces the content of the active interaction with this data.

Another important point is that only atomic interactions are ever validated since the atomic interactions specify a communication between participants. The following paragraphs describe constructs that organize these interactions in specific ways to support branching and recursion logic, but their data structure itself is never truly validated since they do not specifically state a communication. The constructs simply offer a way to identify that another validation process must be used.
Also, since all interactions are linked as a graph, the selection process of the newly monitored interaction is fairly straightforward.

Validating: Choice Validating a choice involves a different process since all branches must be checked against the input of the put or take function. Another difference is the activation of the new active monitored interaction, considering the correct branch must be chosen.

When a choice is encountered in the validation process the monitor must check the incoming put or take operation against all branches. To simplify this, the monitor only needs to check the first operation in all of the branches. Note that Discourje supports *nesting* of choice and recursion, which means this has to be taken into account when checking these interactions.

[1]https://clojuredocs.org/clojure.core/swap!

After retrieving the first interaction of all branches, it simply calls the validate function again on each interaction. This means validation of the choice construct is done in a recursive manner, which allows to easily support nested validation. Validation succeeds if at least one atomic interaction passes the validation process.

When the put or take operation is deemed valid, the next operation must be activated. This process also differs from the atomic interaction. The swapping process of the active monitored interaction involves selecting the correct branch and taking the next interaction, based on the id of the currently authorized interaction since the choice construct itself does not represent an interaction. Again nesting must be taken into account.

Validating: Rec Validation of a recursion construct is slightly different from the choice construct. It is less complex since the recursion construct only contains one path for recursion, compared to the multiple branches in the choice. When a *rec* is encountered in the monitor, the validation process employs the following process.

The first interaction defined in the rec is selected and, recursively, passed into the validation function again to take nesting into account. If it succeeds the validation process, the put or take is permitted.

The activation of the new active interaction is similar to choice; the mechanism selects the next operation based on the authorized interaction.

Validating: Continue The continue statement involves a validation process similar to the rec statement. When the monitor encounters the continue statement, it will query the corresponding rec statement, and simply recursively invoke the validation mechanism on the result. This will start the validation of the recursion, which is described in the previous paragraph. The activation of the next monitor is also the same as with the rec construct.

This means recursion is implemented by simply looping back to the corresponding rec statement. The validation process will then continue as described. There is no ‘special sub protocol‘ or intricate looping mechanism. The linked graph ids solve the recursion mechanism in an easy way.

6.4. LOGGING AND ERROR-HANDLING

To notify a programmer or system of invalid communication, there must be logging and error-handling capabilities. These capabilities allow a system to cope with errors and in turn, avoid communication mismatches and work around them. Discourje offers different levels of logging which can be configured as global settings:

- **Logging:** This configuration will only log communication to the console. This level is ***non-blocking*** on invalid communication.
- **Exceptions:** This configuration will not log to the console, but throw exceptions when incorrect communication is encountered. This level is ***blocking*** on invalid communication.

- **Logging & exceptions:** This level combines both logging and exceptions. It will allow logging to the console, but also throw exceptions when communication is deemed invalid. This level is ***blocking*** on invalid communication.

Blocking in this sense means that a put or take operation is not allowed to proceed and an exception is thrown to be handled accordingly. When just the logging level is enabled, the invalid communication is logged but *the communication itself is allowed to proceed.*

This behavior can be used to truly avoid communication mismatches. Imagine a process where it is not entirely clear when communication is ready to be invoked. When the exceptions level is enabled, the programmer could implement his process logic to catch certain exceptions and retry a communication or steer the communication down another branch of a choice construct. The next subsection will demonstrate implementation details and how to take advantage of this.

6.4.1. CUSTOM TWO BUYER PROTOCOL

To demonstrate the usage for logging capabilities of Discourje imagine the following scenario: A programmer wants to implement the two buyer protocol but with custom channels, all having a buffer size of 2. Increasing the buffer size of the channels has implications for the implementation of the logic for buyer2 specifically. The reason for this is that Discourje will allow put and take communications to proceed when buffer space is available, yet a previous value might not yet have been taken or sent. In the two buyer protocol, there is a specific point in the communication where this problem reveals itself. After buyer2 chooses to contribute to the product, he will send the 'ok' and 'address' messages to the seller. Having a buffer size of 2 will allow both values to be put on the channel but the monitor will only swap the active interaction when the 'ok' message is actually received. As a result, the monitor will report an inadmissible communication.

The following snippets will show how to implement error handling for this kind of scenario. The communication pattern for the two buyer protocol, buyer1 and seller logic is unchanged.

The infrastructure requires the custom generated channels all with a buffer size of 2.

Listing 6.15: Two buyer Infrastructure

```
;custom channels
(def b1-s (chan "buyer1" "seller" 2))
(def s-b1 (chan "seller" "buyer1" 2))
(def b1-b2(chan  "buyer1" "buyer2" 2))
(def s-b2 (chan "seller" "buyer2" 2))
(def b2-s (chan "buyer2" "seller" 2))

;generate the infrastructure with custom channels for the protocol
(def infrastructure (add-infrastructure two-buyer-protocol [b1-s s-b1 b1-b2 s-b2 b2-s]))
```

Listing 6.16: Two buyer - Buyer2 logic

```
(defn contribute?  [quote div]
  (>= (* 100 (float (/ div quote))) 50))

(defn generate-address []
  "Open University, Valkenburgerweg 177, 6419 AT, Heerlen")

(defn order-book "Order a book from buyer2's perspective" [infra]
  (let [s-b2 (get-channel "seller" "buyer2" infra)
        b1-b2 (get-channel "buyer1" "buyer2" infra)
        b2-s (get-channel "buyer2" "seller" infra)
        ok-delivered? (atom false)
        quote (get-content (<!!! s-b2 "quote"))
        quote-div (get-content (<!! b1-b2 "quote-div"))]
    (if (contribute? quote quote-div)
      (do
        (>!! b2-s (msg "ok" "ok"))
        (while (false? @ok-delivered?)
          (try+
            (println "sending address")
            (>!! b2-s (msg "address" (generate-address)))
            (reset! ok-delivered? true)
            (catch [:type :incorrect-communication] {}
              (println "address not delivered, retrying..."))))
        (let [date (<!!! s-b2 "date")]
          (log-message (format "Thank you, I will put %s in my agenda!"
          (get-content date)))
          (order-book infra)))
      (>!! b2-s (msg "quit" "Price to high!")))))
```

The snippet above shows an implementation of buyer2 logic with a try-catch clause around the sending of the address message from buyer2 to seller. Again, buyer2 is unable to know when the ok message is delivered to seller so buyer2 retries to send the address message (in a simple while loop) until it succeeds.

7 BENCHMARKS

This chapter presents details of all benchmarks done for Discourje. The chapter starts with information on what types of benchmarks have been run. Next, the hardware setup will be explained in detail, followed by the results of the benchmarks on this setup. Subsequently, implications of these results are discussed including possible improvements and preliminary benchmarks.

7.1. BENCHMARK TYPES

To study the performance of Discourje, four types of benchmarks are performed. The benchmarks include the *one and two buyer protocols* to demonstrate realistic scenarios. Also, two additional benchmarks have been selected; *pipeline* and *scatter & gather*. The pipeline and scatter & gather benchmarks were chosen to investigate how Discourje performs and scales. Both benchmarks are easy to scale in amount of threads and iterations. The following sections explain all benchmarks in detail.

Importantly, each Discourje benchmark has an equivalent Clojure.core.async counterpart. This Clojure.core.async benchmark executes the same logic, but without Discourje's monitor overhead. By benchmarking equivalent Discourje programs and Clojure.core.async programs, insight is gained on the difference in performance between them.

7.1.1. PIPELINE

The pipeline benchmark is selected to show how Discourje performs compared to Clojure.core.async in an easy to scale way. A pipeline structure is a simple protocol that looks as follows:

$$\begin{tikzcd} 0 \arrow[r] & 1 \arrow[r] & 2 \arrow[r] & \cdot\cdot \arrow[r] & N \end{tikzcd}$$

The network diagram above illustrates a pipeline of N worker threads. The nodes in the diagram represent threads and the edges between them represent channels through which communications transpire. A pipeline follows a sequential protocol where *thread 0* communicates with *thread 1*, and then *thread 1* with *thread 2*, and so on, until reaching N. This protocol is easy to scale in both repetitions and number of threads, where a repetition consists of a single chain of communications in which the whole pipeline participates.

The pipeline benchmark is implemented through a protocol generator. This generator requires only the number of threads and the number of repetitions. The generator creates a protocol that looks as follows:

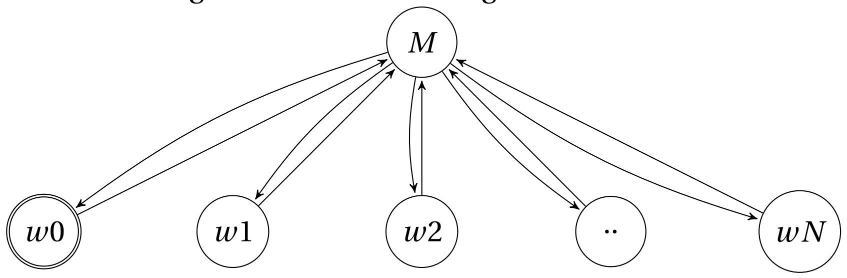
Listing 7.1: Pipeline protocol

```
(pipeline-prot
  (mep
    (-->> 1 p0 p1)
    (-->> 1 p1 p2)
    (-->> 1 .. ..)
    (-->> 1 .. pn)))
```

The listing above shows an example of the generated pipeline protocol. In this protocol, each communication sends a message labeled 1 to the next worker.

7.1.2. SCATTER & GATHER

The scatter & gather benchmark serves the same purpose as the pipeline benchmark.
The scatter & gather protocol employs a master/worker structure. The master will send a message (in parallel) to N workers. Each worker will then send a message back to the master, and when all workers have responded, one repetition of the protocol is complete.
A network diagram for the scatter & gather benchmark looks as follows.



In this network diagram, as before, each node represents a master or a worker thread, and edges represent communication channels between them. This benchmark is scalable in both the number of workers (threads) and repetitions.
This benchmark is also implemented in Discourje through a generator. The generator creates a protocol that looks like the following Listing:

Listing 7.2: Scatter & Gather protocol

```
(worker-prot
  (mep
    (-->> 1 master [worker0 worker1 .. workerN])
    (-->> 1 worker0 master)
    (-->> 1 worker1 master)
    (-->> 1 ....... master)
    (-->> 1 workerN master)))
```

7.1.3. ONE & TWO BUYER PROTOCOLS

The remaining two benchmark types are the one and two buyer protocols. These protocols represent a more realistic communication flow.
This has the following implications for the protocols: In the one and two buyer protocols, the product that the buyer is asking for, is always in stock (to always run the full protocol and to remove randomness). Buyer2 will always accept to contribute to the product buyer1 is asking for.
For testing benchmarks, the protocols can be run with a number of repetitions.

7.2. HARDWARE SETUP

The hardware setup for running the Discourje benchmarks was provided by SURFSara. Permission was granted to run the benchmarks on the Dutch national supercomputer Lisa[1]. The following hardware node was chosen for the benchmarks:

Processor	Intel Gold 6130
Clock	2.10 GHz
Scratch	1.7 TB
Memory	96 GB UPI 10.4 GT/s
Sockets	1
Cache	22 MB
Cores	16
GPUs	-
Interconnected	10 Gbit/s ethernet

Table 7.1: Hardware setup

7.3. RESULTS

For the benchmarks run for Discourje the following requirements and constraints hold:

- Time constraint of 5 minutes per benchmark, based on Discourje run-time.
- For pipeline and scatter & gather, the following numbers workers were chosen: *2, 3, 4, 6, 8, 12, 16, 24, 32.*
- Each benchmark was run 10 times. The average time is taken as result time.

One Buyer protocol Discourje was able to run the one buyer protocol at most *1400000* iterations within the *5 minute* time cap with an average of *04:55 minutes* and a standard deviation of *2.3 seconds.* Clojure.core.async was able to run the same number of repetitions in on average *41 seconds* and a standard deviation of *5 seconds.*

Discourje achieved an average of *0.2 milliseconds per iteration* of the one buyer protocol, while Clojure.core.async was able to run an average of *0.02 milliseconds per iteration.* These results show that Clojure is 7 times faster than Discourje in this benchmark, with an absolute difference between Clojure.core.async and Discourje of 181μs per iteration.

	Discourje	Clojure.core.async
Average	295s	41s
Standard deviation	2349ms	4945ms
Time/iteration	0.210ms	0.029ms
Slowdown	7 times	
Difference per iteration	181μs	

Table 7.2: One buyer benchmark, 1400000 iterations, time in milliseconds

[1]https://userinfo.surfsara.nl/systems/lisa/description

Two Buyer protocol Discourje was able to run the two buyer protocol at most *670000* iterations within the *5 minute* time cap with an average of *04:52 minutes* and a standard deviation of *2.3 seconds*. Clojure.core.async was able to run the same number of repetitions in on average *33 seconds* and a standard deviation of *3.2 seconds*.

Discourje achieved an average of *0.4 milliseconds per iteration* of the two buyer protocol, while Clojure.core.async was able to run an average of *0.05 milliseconds per iteration*. These results show that Clojure is 9 times faster than Discourje in this benchmark with an absolute difference between Clojure.core.async and Discourje of 386μs per iteration.

	Discourje	Clojure.core.async
Average	292s	33s
Standard deviation	2851ms	3185ms
Time/iteration	0.436ms	0.049ms
Slowdown	9 times	
Difference per iteration	386μs	

Table 7.3: Two buyer benchmark, 760000 iterations, time in milliseconds

Scatter & gather

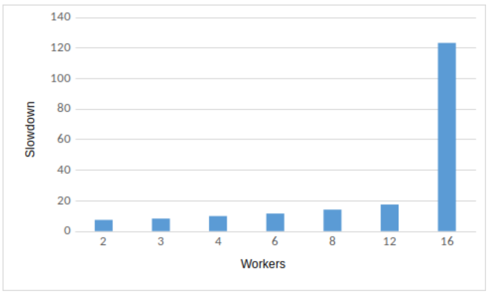


Figure 7.1: Slowdown of Discourje compared to Clojure.core.async.

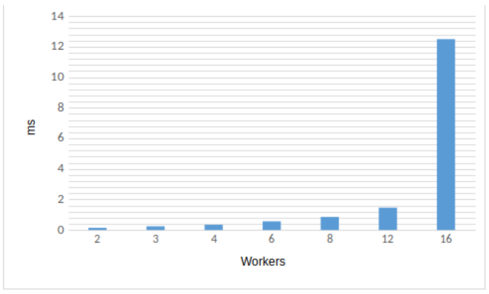


Figure 7.2: Difference between 1 iteration of Discourje and Clojure.core.async.

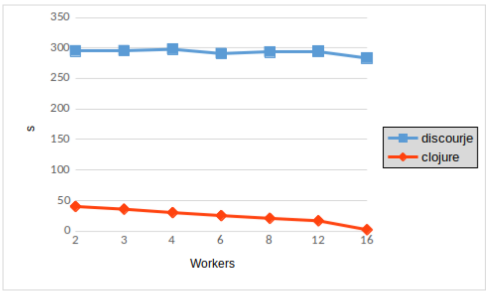


Figure 7.3: Average time for each of the benchmarks.

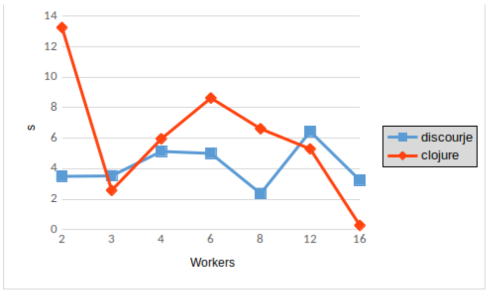


Figure 7.4: Standard deviation for each of the benchmarks.

Figure 7.1 shows the slowdown of Discourje (i.e., validation overhead) relative to Clojure.core.async. The benchmarks for *24* and *32* workers are omitted from the figure intentionally, because for those numbers of worker threads, Discourje's slowdown increases to *175 times* and *278 times* respectively. Showing these results in the figure would make the other results hard to read.
Figure 7.2 shows the absolute difference in milliseconds for one iteration for each number of worker threads between Discourje and Clojure.core.async. Results for *24* workers (*30ms*) and *32* workers (*65ms*) have been omitted for the same reason as before;
Figure 7.5 shows the average (10 runs per benchmark) for each number of workers; Figure 7.6 shows the corresponding standard deviations.

Pipeline

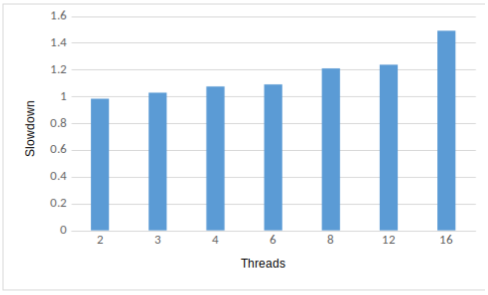


Figure 7.5: Slowdown of Discourje compared to Clojure.core.async.

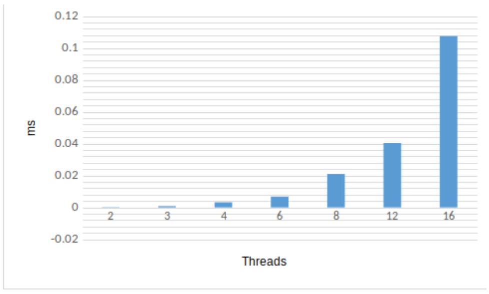


Figure 7.6: Difference between 1 iteration of Discourje and Clojure.core.async.

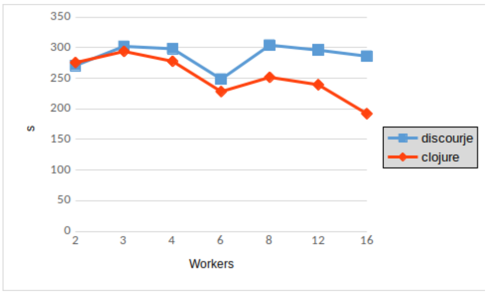


Figure 7.7: Average time for each of the benchmarks.

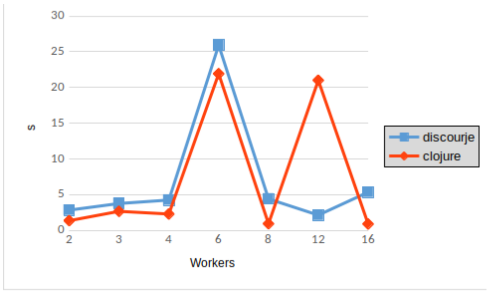


Figure 7.8: Standard deviation for each of the benchmarks.

Figure 7.5 shows the slowdown of Discourje (i.e., validation overhead) relative to Clojure.core.async. The benchmarks for *24* and *32* workers are omitted from the figure intentionally, because for those numbers of worker threads, Discourje's slowdown increases to *19 times slower* and *179 times slower* respectively. Showing these results in the figure would make the other results hard to read.
Figure 7.6 shows the absolute difference in milliseconds for one iteration for each number of worker threads between Discourje and Clojure.core.async. Results for *24* workers (*7ms*) and *32* workers (*92ms*) have been omitted for the same reason as before;
Figure 7.7 shows the average (10 runs per benchmark) for each number of workers; Figure 7.8 shows the corresponding standard deviations.

7.4. IMPLICATIONS

The results of the benchmarks show that Clojure.core.async is considerably faster than Discourje. Due to the nature of how Discourje is built on top of Clojure.core.async, a slowdown was expected, but not to the extent for 24 and 32 worker threads in the results of the pipeline and scatter & gather protocols. For the one and two buyer protocols Clojure.core.async is roughly 10 times faster.

Interestingly, the slowdown of Discourje in both the pipeline and the scatter & gather benchmark increases quickly as the number of threads increases beyond 16. This number is the same as the number of cores of the benchmark machine.

Pipeline After analysis of the pipeline benchmark results it became clear that the monitor had impact on the slowdown, and specifically, the query to activate the next permitted interaction (see Chapter 6 for details). The speed of the query correlates with the 'length' of the protocol, so a protocol with two monitored interactions is queried faster than a protocol with twenty interactions. This also negatively affects the scatter & gather benchmark, and the one and two buyer benchmarks.
The reason why Discourje slows down this much as the size of a protocol increases, is that the run-time query for the next interaction to monitor, after the current interaction is validated, *always* starts at the beginning of the protocol. For example: in the one buyer protocol, when the *quote request* interaction is validated and the choice construct needs to be activated, the query will find the *next* immediately since the choice is set as next of the *quote request*. However, when the *order-ack* interaction needs to be queried, Discourje will again start at index 0, and from there search for an interaction with an ID that matches the *order-ack*, thereby traversing the entire protocol specification in a depth-first manner. This means that the longer the protocol is, the slower the query becomes.

Scatter & gather The results of the scatter & gather benchmark show that Discourje slows down significantly when more workers are added to the protocol. For the last benchmark of 32 workers, Discourje ran an average of 65 ms per iteration whereas Clojure.core.async ran an average of 0.2 ms per iteration. Analysis of the implementation of the scatter & gather protocol in Discourje found the same issue as with the pipeline. The query, at run-time, for the next permitted interaction seems to contribute to Discourje's degradation. However, in the scatter & gather protocol, there seems to be another important implementation detail that causes a performance hit.

In the scatter & gather protocol the master sends a message to each worker (thread) in parallel. Each worker must respond to the master, and when they have all responded, the current iteration ends. An attentive reader might noticed that Discourje does not support this sort of behavior. In Discourje, there can be *only 1 interaction* monitored and permitted at one point in time. Also notice, in Listing 7.2, the protocol specification defines that each worker must respond in order.
In order to work around this problem, Discourje's *logging and error handling* capabilities were used (see Section 6.4 for more information). In the scatter & gather implementation, each worker will continuously try to send back to the master. Each time the worker does not succeed, an exception will be caught and the communication is retried immediately.

The throwing and catching of these exceptions seem to yield a significant performance hit, to the point where Discourje is 278 times slower than Clojure.core.async with 32 workers.

One and two buyer protocols Considering the query overhead of the monitor, first note that the two buyer protocol is 1.5 times larger than the one buyer protocol. Specifically, in the two buyer protocol, 6 interactions (title, quote, quote-div, ok, address, date) are monitored each iteration, while in the one buyer protocol only 4 interactions (quote-request, quote, order, order-ack) are monitored per iteration. The results show that using Discourje, an iteration of the one buyer protocol takes only half the time of an iteration of the two buyer protocol.
These results indicate again that the number of interactions in a protocol specification correlate to slowdown using Discourje. However, the absolute difference between Discourje and Clojure.core.async for the one buyer protocol is only 181μs. For the two buyer protocol the absolute difference per iteration is roughly double at 386μs.

Concluding thoughts In each benchmark, the run-time query of the new permitted interaction seems to contribute significantly to the slowdown of Discourje when the number of interactions in the protocol specification grows. Also, throwing many exceptions introduces a performance hit. The next section presents improvements to solve these issues.

7.5. IMPROVEMENTS

This section will describe improvements which could be implemented to improve the performance of Discourje. Observing and analyzing the results of the benchmarks led to the following insights:

- **I1:** An 'unordened' or 'parallel' interaction type should be added to be able to activate multiple interactions at once. This will solve the performance degradation in the scatter & gather protocol.
- **I2:** Throwing exceptions is slow. There should be an alternative, practical way for signaling a developer a communication is invalid without using exceptions.
- **I3:** There is a correlation between the 'size' of a protocol and performance degradation. The reason for this is that the state machine generated for the monitor has IDs for the next interactions. These should be replaced by direct references at compile-time, to avoid queries at run-time.

The next three subsections explain the suggested improvements in more detail.

7.5.1. UNORDENED COMMUNICATION TYPE

Implementing and analyzing the results for the scatter & gather benchmark made it clear Discourje lacks the option to receive different messages concurrently. Currently, Discourje does support multicasting of the same message to multiple receivers.

The current scatter & gather protocol (see Listing 7.2) uses a multicast from the master to all workers. However, all workers must respond in order. This leads to a communication flow where the master sends a message to all workers at the same time. However, each worker must wait sending a message to the master until the previous worker in the list has sent a message *and* the master has actually taken it from a channel. This is directly demonstrated in the generated *workers-prot* shown in Listing 7.2. According to this protocol, worker 0 must have communicated with the master before worker 1 is able to communicate with the master.

This behavior feels contradictory to the nature of the typical scatter & gather flow since all workers should be able to respond to the master without having to wait for each other. Workers must be independent.
The current implementation of the scatter & gather benchmark does implement the workers independently. This results in an implementation where each worker tries to communicate with the master, which will likely throw an exception (slowing down the benchmark) since the worker is not allowed to communicate yet. The worker will then retry, as many times as needed, until the communication succeeds, again generating exceptions in the process.

Discourje should add the option to *parallelize* different interactions, independent of message label, sender and receiver. This will remove all exception throwing and handling from, for example, the scatter & gather implementation. The protocol implementing this type of behavior might look as follows:

Listing 7.3: Parallelism example

```

(def scatter-gather-parallel
  (mep
    (-->> 1 "master" ["worker0","worker1", "...", "workerN"])
    (parallel [(-->> 1 "worker0" "master")
                (-->> 1 "worker1" "master")
                (-->> 1 ".."      "master")
                (-->> 1 "workerN" "master")])))
```

The listing above should allow for each worker to respond independent of other workers.

The main challenge of this extension is that Discourje currently only permits **one** interaction to be active so, after a parallel communication, only one branch of the parallel construct can continue. Imagine a scenario where, in the scatter & gather protocol, the master must send a confirmation message to each worker. It feels intuitive to directly respond to each worker, but this would mean that, depending on a parallelized communication, a new communication path must be enabled, ending up in state machine where multiple paths are traversed at the same time. Also, note that branching and recursion logic could be nested. The following example illustrates this problem:

Listing 7.4: Nested parallel example

```

(def scatter-gather-parallel-nested
  (mep
    (-->> 1 "master" ["worker0","worker1", "...", "workerN"])
    (parallel [[(-->> 1 "worker0" "master") (-->> "confirm" "master" "worker0")]
               [(-->> 1 "worker1" "master") (-->> "confirm" "master" "worker1")]
               [(-->> 1 ".."      "master") (-->> "confirm" "master" "...")]
               [(-->> 1 "workerN" "master") (-->> "confirm" "master" "workerN")]])))
```

The listing above shows how the master should respond to each worker with a message labeled ‘confirm’. Since all interactions in the parallel are happening concurrently, the response messages should also be permitted.

7.5.2. EXCEPTIONS

Throwing exceptions on the JVM is slow. For each exception, a full stack-trace must be generated, which costs time. Additionally, in the implementation of the Discourje run-time system a third party library, called SlingShot[2], was used to generate and throw exceptions. SlingShot allows all objects to be thrown as an exception (instead of only objects of Java type Throwable). SlingShot was added to Discourje as a convenience library, but eventually has a significant impact on performance.

A possible solution to fix this issue is to use special return values for the put and take functions in case validation fails, instead of throwing exceptions. Clojure.core.async has a rather elegant solution for this pattern; both the put and take operations return nil when they are not successful. When they are successful, the put operation returns the channel on which the data was put, and the take returns the value from the channel as expected. A developer is able to check the return value from the put and take functions and act accordingly when they return nil. This achieves the same result as throwing an exception, yet without expensive stack-trace generation.

7.5.3. IDS TO REFERENCES

Discourje's monitor is implemented as a state machine, where each interaction represents an edge. When the interactions are 'linked' and this state machine is generated (described in Chapters 4 and 6), an ID string is set as 'next' for each of the interactions. As a result, each time the active interaction is updated, a query must search for the next interaction by this id. The monitor always starts this query at index 0 of the supplied protocol, meaning query time directly correlates with the amount of interactions specified in a protocol.

A possible solution to solve this problem is to not save the next id, but to save a reference to the entire next interaction object. This would completely remove the run-time query, since the next interaction can be activated directly.
Currently, a linked protocol ends up looking as follows:

Listing 7.5: Atomic interaction linking

```
Before linking:

(def protocol
  (mep
    (-->> "Foo" "Alice" "Bob")
    (-->> "Bar" "Bob" "Alice")
    (-->> "Baz" "Alice" "Bob")))

After linking:

(def protocol
  (mep
    (-->> id: 1, "Foo" "Alice" "Bob" next: 2)
    (-->> id: 2, "Bar" "Bob" "Alice" next: 3)
    (-->> id: 3, "Baz" "Alice" "Bob" next: nil)))
```

[2]https://github.com/scgilardi/slingshot

To implement *next* as a reference instead of an id, the protocol must end up looking as follows:

Listing 7.6: New linking

```
New linking:

(def protocol
  (mep
    (-->> id: 1, "Foo" "Alice" "Bob" next:
      (-->> id: 2, "Bar" "Bob" "Alice" next:
        (-->> id: 3, "Baz" "Alice" "Bob" next: nil)))))
```

In Listing 7.6, the value of next is changed from an id to a direct reference to the next interaction, resulting in a (deeply) nested data structure (depending on the length of the protocol). The example above only demonstrates how an atomic interaction could be linked; this process must be generalized for choice, rec and continue.

7.6. PRELIMINARY BENCHMARKS

This section reports on the results of *preliminary* benchmarks performed for the pipeline and scatter & gather protocols after the run-time query for the next interaction was removed and proper return values (no exceptions) were implemented.

To save time, only the linking of the atomic-interaction is updated, meaning choice, rec and continue are *not* implemented. This means the preliminary benchmarks can only be run for the pipeline and scatter & gather benchmarks.
The workaround for the scatter & gather benchmark is still implemented since Discourje does not provide a parallel communication type. However, post-update Discourje will no longer throw exceptions but will return proper values from the send and receive abstractions when invalid communication is detected.
The results will show how post-update Discourje compares to pre-update Discourje and Clojure.core.async.

7.6.1. PIPELINE

Pre-update & post-update Discourje

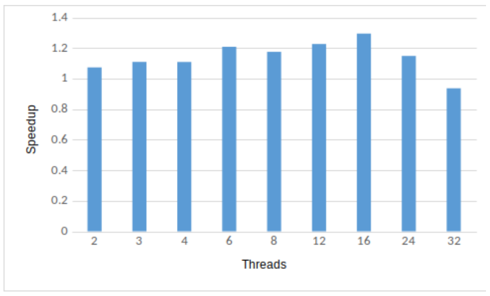


Figure 7.9: Slowdown of pre-update Discourje compared to post-update Discourje.

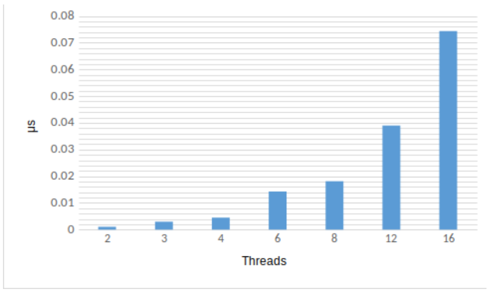


Figure 7.10: Difference between 1 iteration of pre-update Discourje and post-update Discourje.

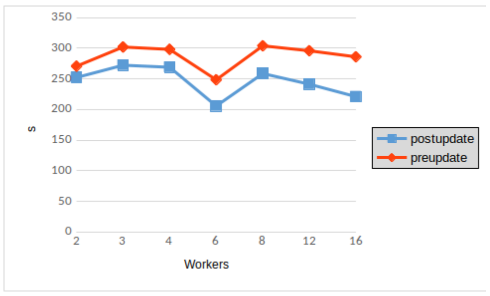


Figure 7.11: Average time for each of the benchmarks.

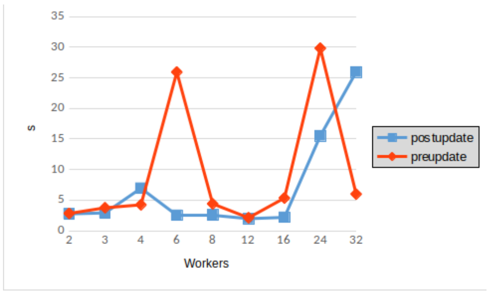


Figure 7.12: Standard deviation for each of the benchmarks.

Figure 7.9 shows the speedup of post-update Discourje relative to pre-update Discourje (>1 means post-update Discourje is faster). Interestingly, with 32 threads, post-update Discourje is slower than pre-update Discourje.
Figure 7.10 shows the absolute difference in milliseconds for one iteration between pre and post update Discourje.
Figure 7.1 shows the average (10 runs per benchmark) for each number of workers; Figure 7.12 shows the corresponding standard deviations.

Implications The results of the benchmarks show that removing the run-time query made post-update Discourje 1.1 times faster than pre-update Discourje. Interestingly, pre-update Discourje is 1.06 times faster than post-update Discourje with 32 threads in the pipeline. Overall the results show that the removal of the run-time query for the next permitted interaction made Discourje faster, this improvement should be generalized from atomic interactions to all language constructs.

Clojure & post-update Discourje

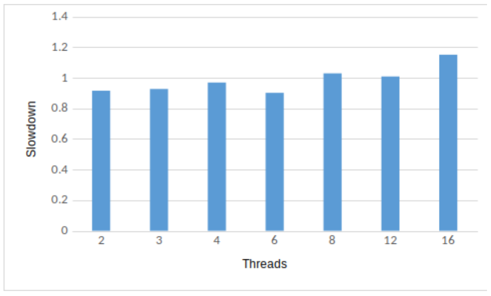


Figure 7.13: Slowdown of Discourje compared to Clojure.core.async.

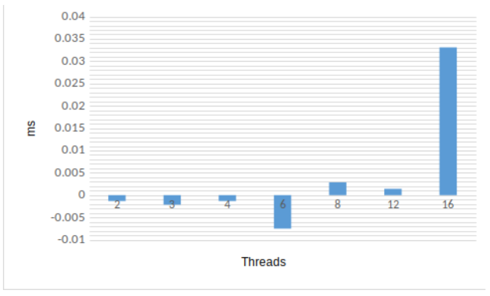


Figure 7.14: Difference between 1 iteration of Discourje and Clojure.core.async.

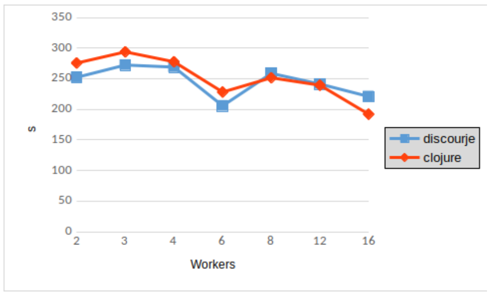


Figure 7.15: Average time for each of the benchmarks.

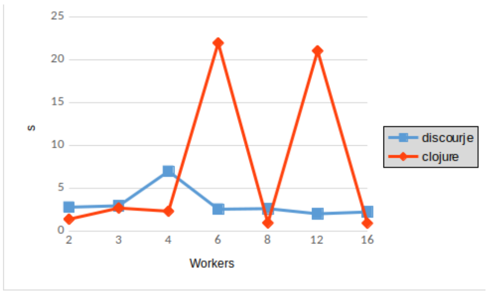


Figure 7.16: Standard deviation for each of the benchmarks.

Figure 7.13 shows the slowdown of Discourje (i.e., validation overhead) relative to Clojure.core.async. The benchmarks for *24* and *32* workers are omitted from the figure intentionally, because for those numbers of worker threads, Discourje's slowdown increases to *16 times slower* and *192 times slower* respectively. Showing these results in the figure would make the other results hard to read.
Figure 7.14 shows the absolute difference in milliseconds for one iteration for each number of worker threads between Discourje and Clojure.core.async. Results for *24* workers (*6.7ms*) and *32* workers (*98ms*) have been omitted for the same reason as before;
Figure 7.15 shows the average (10 runs per benchmark) for each number of workers; Figure 7.16 shows the corresponding standard deviations.

Implications Interestingly, the results show that Discourje is faster than Clojure.core.async up to 6 threads in the pipeline, but then slows down rapidly beyond 12 threads. Post-update Discourje's performance seems comparable to Clojure.core.async's performance so long as the number of threads does not exceed the number of cores. Perhaps, when multiple threads run on the same core, they must compete with each other resulting in a performance hit. Another possible cause could be inefficient use of the Clojure STM when multiple threads run on the same core, and they have to wait for each other, since the STM enforces 'swapping' the active interaction in transaction-like operations.[3]

[3]http://java.ociweb.com/mark/stm/article.htmlClojureSTMHigh

7.6.2. SCATTER & GATHER

Pre-update & post-update Discourje

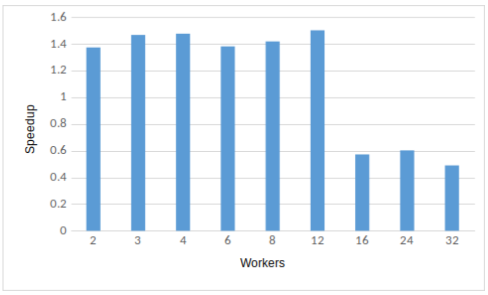


Figure 7.17: Slowdown of pre-update Discourje compared to post-update Discourje.

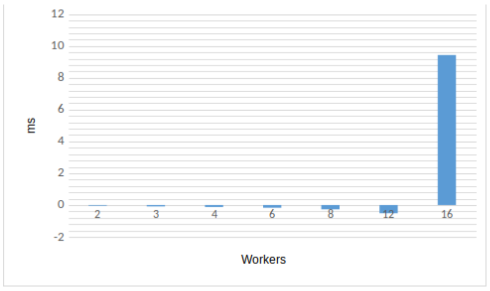


Figure 7.18: Difference between 1 iteration of pre-update Discourje and post-update Discourje.

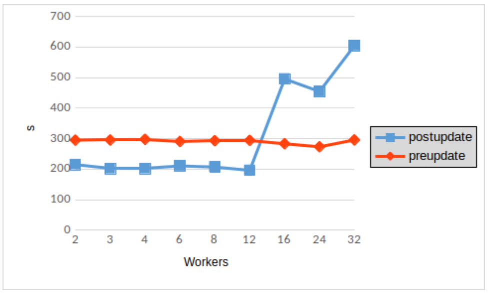


Figure 7.19: Average time for each of the benchmarks.

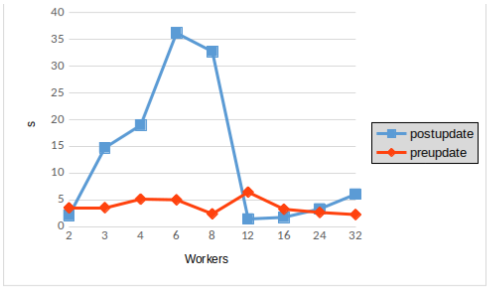


Figure 7.20: Standard deviation for each of the benchmarks.

Figure 7.17 shows the speedup of post-update Discourje relative to pre-update Discoruje. Interestingly, post-update Discourje is roughly 1.3 times faster than pre-update Discourje, but slows down at 16 workers
Figure 7.18 shows the absolute difference in milliseconds for one iteration for each number of worker threads between pre-update Discourje and post-update Discourje.
Figure 7.19 shows the average (10 runs per benchmark) for each number of workers;
Figure 7.20 shows the corresponding standard deviations.

Implications The results of the benchmarks show that removing the run-time query and exceptions made Discourje 1.3 times faster than pre-update Discourje. Interestingly, pre-update Discourje is almost twice as fast with 16, 24 and 32 workers. Overall the results show that the removal of the run-time query for the next permitted interaction made Discourje faster. The removal of exceptions does not seem to make a significant contribution to the speedup of post-update Discourje. In fact, it seems to slow post-update Discourje down with 16, 24 and 32 workers. A possible cause could be that, without exceptions and the maximum number of cores on the hardware in use, post-update Discourje allows workers to retry sending to the master faster, resulting in more overhead of the monitor.

Clojure & post-update Discourje

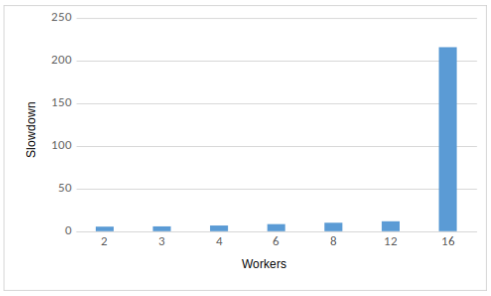


Figure 7.21: Slowdown of Discourje compared to Clojure.core.async.

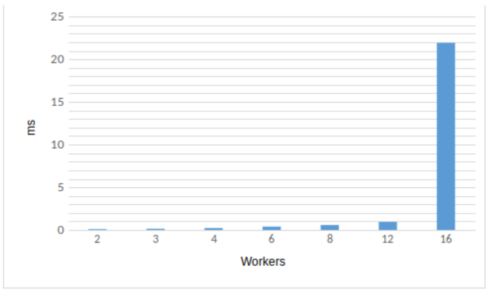


Figure 7.22: Difference between 1 iteration of Discourje and Clojure.core.async.

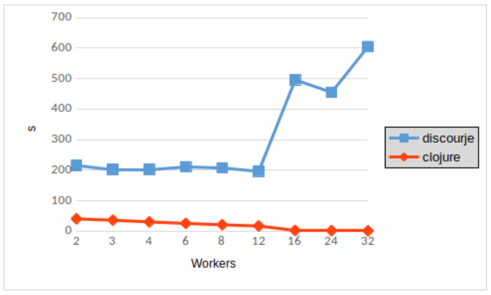


Figure 7.23: Average time for each of the benchmarks.

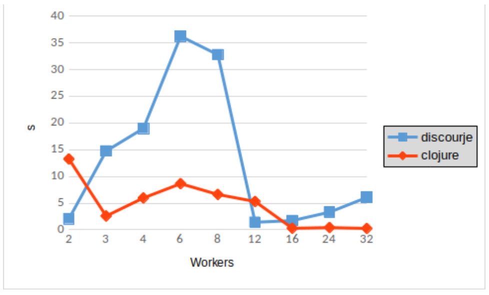


Figure 7.24: Standard deviation for each of the benchmarks.

Figure 7.21 shows the slowdown of post-update Discourje (i.e., validation overhead) relative to Clojure.core.async. The benchmarks for *24* and *32* workers are omitted from the figure intentionally, because for those numbers of worker threads, Discourje's slowdown increases to *291 times slower* and *568 times slower* respectively. Showing these results in the figure would make the other results hard to read.
Figure 7.22 shows the absolute difference in milliseconds for one iteration for each number of worker threads between post-update Discourje and Clojure.core.async. Results for *24* workers (*50ms*) and *32* workers (*134ms*) have been omitted for the same reason as before; Figure 7.23 shows the average (10 runs per benchmark) for each number of workers; Figure 7.24 shows the corresponding standard deviations.

Implications The results show that post-update Discourje is faster than pre-update Discourje. Results of post-update Discourje of the scatter & gather benchmarks show a similar speedup as with the preliminary pipeline benchmarks. This implies that the removal of exceptions did not make a serious impact on performance. However, to further bolster usability, proper return values indicating a miscommunication should nevertheless be implemented to mimic Clojure.core.async's API.

8 Conclusion

This chapter presents the conclusion of the research project. The first section summarizes the answers to the research questions. The second section describes future work on Discourje.

8.1. Answers to research questions

Is it possible to create Clojure macros with embedded coordination logic while taking advantage of immutability characteristics? The main contribution of this thesis is a collection of embedded protocol language macros to provide a seamless syntax and IDE experience. Discourje offers these macros as an abstraction layer on top of Clojure.core.async. The macros form the Discourje DSL, described in Chapter 6, the run-time API to invoke communication primitives in threads, and the run-time system that implements this API. As a result, Discourje alleviates three usability issues that existing protocol languages suffer from: (1) they require special syntax and semantics;(2) they may be bound to a specific IDE;(3) they require additional compilation stages which can be disruptive.

Specifically, to use the Discourje DSL, programmers require no additional knowledge of syntax beyond standard Clojure; of course, programmers do need to know the semantics of the different kinds of interaction types (atomic interaction, choice, rec, continue). Discourje also completely eliminates the need for a dedicated protocol language compiler, as existing protocol languages have: the Discourje DSL, the API, and the run-time system are all written as Clojure data structures, which are compiled to Java byte-code just as any other Clojure code. Finally, Discourje has no dependency on any IDE or text editor, unlike other existing protocol languages.

The key design aim for Discourje's API was to mimic the Clojure.core.async's API, to minimize the migration effort from Clojure.core.async to Discourje. When Discourje is used with wildcards and typed messages, the API matches completely with Clojure.core.async. This allows programmers to effortlessly migrate from Clojure.core.async to Discourje and take advantage of basic validation functionality; programmers can then gradually start using more advanced monitoring capabilities as they need.

Discourje is also able to take full advantage of Clojures immutability characteristics when all communication is done through messages which use *native Clojure data types.When* *using native Clojure data types, freedom of data races is guaranteed.* However, Discourje fundamentally cannot offer the same guarantee when using Java objects as messages, since these objects might be mutable.

How do these macros perform? As Discourje is implemented as a layer on top of Clojure.core.async, it is reasonable to expect it to introduce a performance hit; the aim of the benchmarks was to investigate how much overhead Discourje imposes.

Generally speaking, the benchmarks reveal that Discourje is roughly 10 times slower than Clojure.core.async. The one and two buyer protocol benchmarks show consistently that Discourje is about 10 times slower than Clojure.core.async. The pipeline and scatter & gather benchmark show similar results when the number of threads does not exceed the number of physical processor cores (16 in the benchmark machine); when the number of threads increases beyond the number of cores, Discourje's performance quickly deteriorates.

The benchmarks gave insight into two issues in Discourje's implementation. First, the monitor appears to slow down communication when a protocol specification gets larger. Second, the scatter & gather protocol implementation crucially relied on Discourje's logging and error handling capabilities; this causes Discourje to throw many exceptions, which results in a performance hit.

Preliminary benchmarks show that improvements are promising. After the updates, Discourje is roughly 1.1 times faster than before the updates.

The benchmarks are run with threads that do not perform any computation or processing when data is sent or received; the reported overheads are, thus, only related to communication. This means in a realistic program environment, Discourje's overhead can often be insignificant since actual expensive computational tasks in such cases dominate total program run-time. For example: Imagine a scenario where an implementation requires reading and writing data from a disk, or downloading data from the internet. In such a scenario, the performance of the program is highly dependent on disk read and write time, or internet connection speed. This means that the performance overhead Discourje introduces could be negligible relative to the computational logic in between communications.

Is it possible to increase performance by moving macro execution from run-time to compile-time and what are trade offs? The benchmarks exposed several improvements to speed up Discourje. One of these improvements involves refactoring the run-time queries for the next monitor to a direct reference. This essentially removes an impact-full task from run-time to compile-time since the generation of the monitor will do this type of work when constructed through a macro. Another potential disadvantage is that the resulting monitor will end up as a (deeply) nested data structure (depending on the length and constructs of the protocol), which might make debugging protocol violations difficult. Future work might involve investigating the impact of this possible issue.

8.2. FUTURE WORK

Improvements found through benchmarks: After analyzing the results of the benchmarks, three improvements are suggested. The first improvement of giving a developer the choice to throw exceptions or use simple return values from put or take abstractions to handle incompliant communications can be implemented fairly easily.
The second improvement; restructuring state machine next ids to direct references, is likely to require more effort since all monitor logic must be changed, and re-tested.
The third improvement (adding 'parallel communications') will require some more research on what specific semantics is desired. The implementation of parallelism where multiple state machines are traversed at the same time has a significant impact on Discourje since it currently only supports one active interaction at one time. Also, adding this type of parallel logic could make reasoning about a protocol difficult.

Measuring usability: Measuring the actual usability of Discourje is an interesting next step. Future work could aim at setting up (at least) two test groups, where one group uses Discourje and the other group Clojure.core.async for implementing one, or multiple, concurrent programs. Usability could be measured through code quality metrics, time it takes to complete the implementation of a concurrent program, benchmarks for each program, and surveys asking developers how they experienced writing the programs. Additionally, the migration effort from a simple Clojure.core.async application to Discourje could be measured also.

ClojureScript: Clojure can be compiled to Javascript to run, for example, in a browser. Additionally, Clojure.core.async is also compatible with ClojureScript. Future work can explore the behavior of Discourje when compiled to ClojureScript, to allow the same usability advantages for concurrent Javascript programs.

Parametrization: The concept of parametrized coordination languages has been explored in previous research [Van18] (as described in Chapter 4). Some advanced protocol languages, like PrDK and Pabble support ths feature. Discourje currently has no support for parametrization. An interesting line of future work is to explore how to implement such functionality in Discourje, while still promoting usability.

A

Compile-time vs. run-time

Since Clojure lives on the JVM there are comparisons with Java compilation. *The following images are taken from a presentation given by Rich Hickey called: Clojure - An Introduction for Java Programmers.*[1]

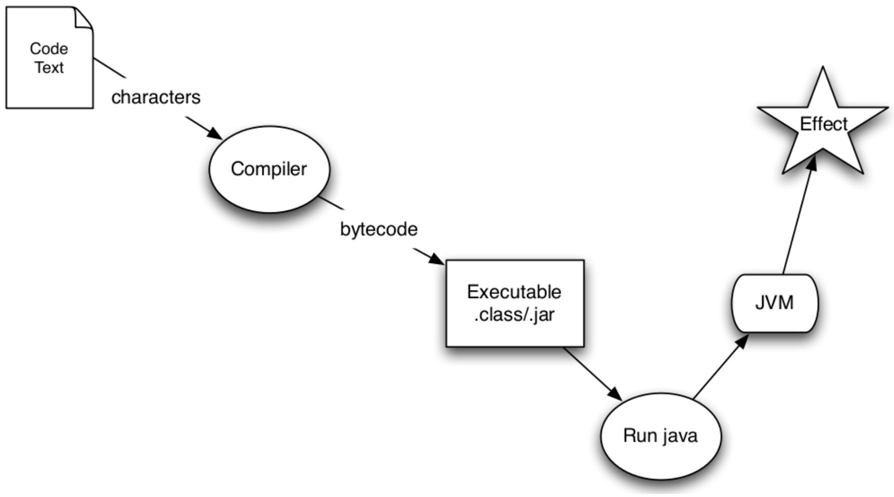


Figure A.1: Traditional Java evaluation.

Traditional Java compilation looks like the figure above. It shows source code is compiled into byte-code as *.class* or *.jar* files. These files of byte-code can be executed by the JVM and form a program.

In this scenario, a programmer has two ways of interacting with the JVM. The programmer can either write source code and compile it into byte-code, or he can use external (third-party) .class and .jar files. The programmer cannot influence the compilation process in any way since this will violate the Java sandbox.

In this model there is a clear separation of compile-and-run-time systems since the Java compiler is in full control of the compilation process. This, however, is different in Clojure and is shown in the figure below.

[1]https://www.slideshare.net/adorepump/clojure-an-introduction-for-java-programmers

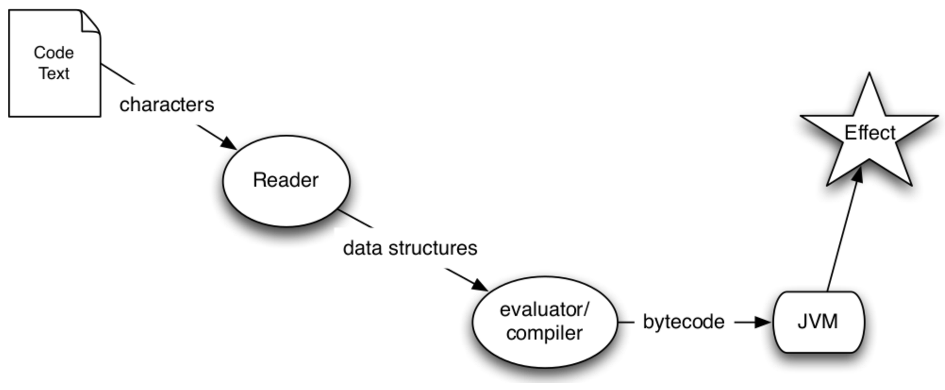


Figure A.2: Clojure evaluation.

In Clojure, there is a component called the *Reader* which is responsible for turning text into Clojure data structures. The reader is also the reason why Clojure is homoiconic.

Homoiconic means that Clojure programs are represented as Clojure data structures. This means that Clojure is defined in terms of evaluation of data structures instead of syntax of characters in files. This is quite different than Java, since Java requires a lot a of syntax for the compiler (if, else, while, for, etc.).
The job of the reader is to parse the Clojure text (files) and produce the data structures the compiler will use. In turn, the compiler will compile the data structures into byte-code.
Important here is the separation of the reader and the compiler. The compiler will only ever 'see' data structures since the reader produces these; This is what is unique to LISP and allows for homiconicity. Having this separation of concerns has several advantages:

First, since the reader is responsible for turning text into data structures, Clojure is able to offer a REPL which forms an interactive environment.

Second, the compiler could be fed pure data structures instead of text, stripping out the reader. This essentially means that programs can generate data structures that the compiler can use and turn into another program (program-generating programs).

Third on this list is the power of all LISPs: the macro system. The macro system allows programmers to participate in the process of generating the data structures for the compiler. This means that a programmer can extend the compiler, and thus the language, by writing macros which directly represent data structures. The image below shows the entire Clojure syntactic abstraction.

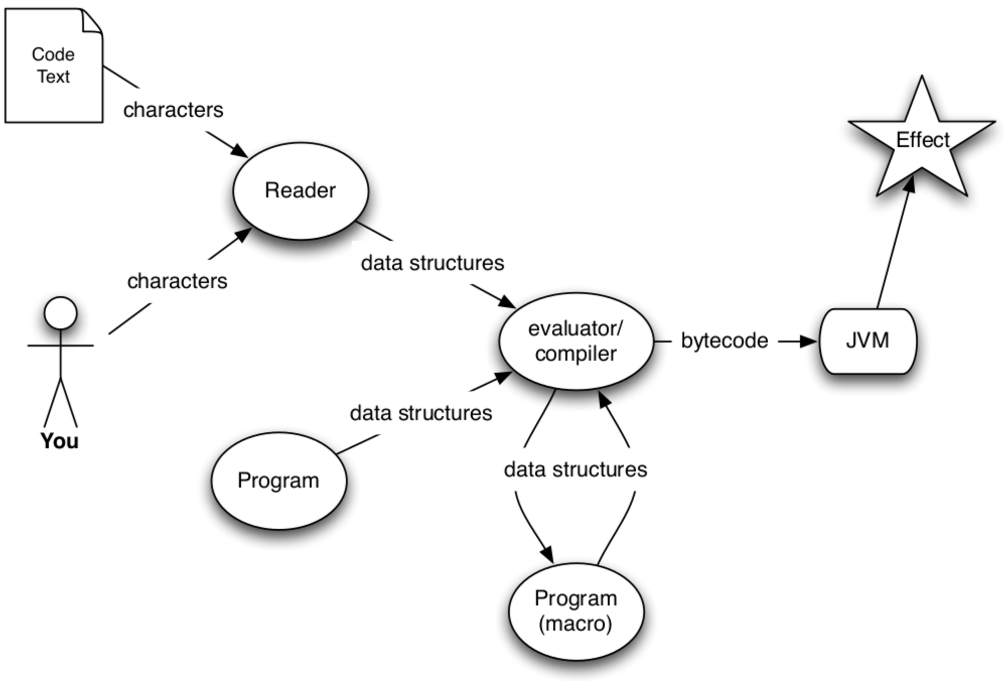


Figure A.3: Total Clojure evaluation.

Important in this figure is that the programmer can write macros which represent tiny programs. The programmer is then able to use these macros in Clojure text (files) and send them to the reader and in turn to the compiler. When the compiler encounters a macro keyword, it will not try to compile or evaluate it, but lookup the definition of this data structure which is represented as the custom written macro and evaluate that.

Conclusion The difference between compile-time and run-time in Clojure is somewhat intertwined. The programmer could either write text to feed to the reader, which is then parsed into data structures for the compiler. Or a program could be generating new data structures directly. The programmer can also use macros to extend the compiler, thereby manipulating the compilation process and adding special language features. In this regard, the compile-time and run-time environments are in control of the programmer.

Discourje compile-time system The compile-time system of Discourje consists of all DSL macros for defining a protocol specification. These macros represent custom data structures which can be combined to construct more complex data structures. These macros, when evaluated, are just normal Clojure data structures.

Discourje run-time system The run-time system of Discourje handles communication and validation. It also requires some setup and configuration for the validation to work properly. This functionality is offered through the Discourje API.

Bibliography

[Arb04] Farhad Arbab. “Reo: a channel-based coordination model for component composition”. In: *Mathematical Structures in Computer Science* 14.3 (2004), pp. 329–366. DOI: 10.1017/S0960129504004153.

[BHR84] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. “A Theory of Communicating Sequential Processes”. In: *J. ACM* 31.3 (June 1984), pp. 560–599. ISSN: 0004-5411. DOI: 10.1145/828.833. URL: http://doi.acm.org/10.1145/828.833.

[Cop+15] Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. “A Gentle Introduction to Multiparty Asynchronous Session Types”. In: *Formal Methods for Multicore Programming: 15th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2015, Bertinoro, Italy, June 15-19, 2015, Advanced Lectures*. Ed. by Marco Bernardo and Einar Broch Johnsen. Cham: Springer International Publishing, 2015, pp. 146–178. ISBN: 978-3-319-18941-3. DOI: 10.1007/978-3-319-18941-3_4. URL: https://doi.org/10.1007/978-3-319-18941-3_4.

[FH11] Michael Fogus and Chris Houser. *The Joy of Clojure: Thinking the Clojure Way*. 1st. Greenwich, CT, USA: Manning Publications Co., 2011. ISBN: 1935182641, 9781935182641.

[Fos95] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0201575949.

[Hon+11] Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen, and Nobuko Yoshida. “Scribbling Interactions with a Formal Foundation”. In: *Distributed Computing and Internet Technology*. Ed. by Raja Natarajan and Adegboyega Ojo. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 55–75. ISBN: 978-3-642-19056-8.

[HVK98] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. “Language primitives and type discipline for structured communication-based programming”. In: *Programming Languages and Systems*. Ed. by Chris Hankin. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 122–138. ISBN: 978-3-540-69722-0.

[HYC16] Kohei Honda, Nobuko Yoshida, and Marco Carbone. “Multiparty Asynchronous Session Types”. In: *J. ACM* 63.1 (Mar. 2016), 9:1–9:67. ISSN: 0004-5411. DOI: 10.1145/2827695. URL: http://doi.acm.org/10.1145/2827695.

[JA13] Sung-Shik T. Q. Jongmans and Farhad Arbab. “Global Consensus through Local Synchronization”. In: *Advances in Service-Oriented and Cloud Computing*. Ed. by Carlos Canal and Massimo Villari. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 174–188. ISBN: 978-3-642-45364-9.

[JA18] S.-S.T.Q. Jongmans and F. Arbab. “Centralized coordination vs. partially-distributed coordination with Reo and constraint automata”. In: *Science of Computer Programming* 160 (2018). Fundamentals of Software Engineering (selected papers of FSEN 2015), pp. 48–77. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2017.06.004. URL: http://www.sciencedirect.com/science/article/pii/S0167642317301259.

[JR14] C. Jones and F.Y. Rashid. *Mastering Clojure Macros: Write Cleaner, Faster, Smarter Code*. Pragmatic Programmers. Pragmatic Bookshelf, 2014. ISBN: 9781941222225. URL: https://books.google.nl/books?id=vxXLoQEACAAJ.

[MHB18] Alex Miller, Stuart Halloway, and Aaron Bedra. *Programming Clojure (The Pragmatic Programmers)*. Pragmatic Bookshelf, 2018. ISBN: 1680502468. URL: https://www.amazon.com/Programming-Clojure-Pragmatic-Programmers-Miller/dp/1680502468?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=1680502468.

[NY14] Nicholas Ng and Nobuko Yoshida. “Pabble: Parameterised Scribble for Parallel Programming”. In: *22nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing*. IEEE Computer Society, 2014, pp. 707–714. DOI: 10.1109/PDP.2014.20.

[SL15] J. M. Shalf and R. Leland. “Computing beyond Moore’s Law”. In: *Computer* 48.12 (Dec. 2015), pp. 14–23. ISSN: 0018-9162. DOI: 10.1109/MC.2015.374. URL: doi.ieeecomputersociety.org/10.1109/MC.2015.374.

[Van18] Bernie Van Veen. “Modular Programming of Synchronization and Communication among Tasks in Parallel Programs”. In: ed. by Sung-Shik T. Q. Jongmans. IEEE Computer Society, 2018, pp. 425–435.

[Zai+14] Pavel Zaichenkov, Bert Gijsbers, Clemens Grelck, Olga Tveretina, and A Shafarenko. *A Case Study in Coordination Programming: Performance Evaluation of S-Net vs Intel’s Concurrent Collections*. May 2014.