Universidade da Beira Interior
Informatics Department

Departamento de Informática

MP4 - 2023: *Semantic Analyzer for DSL (transformations)*

Authored By:

Leonardo Mendes dos Santos, Nº 48708

Mentors:

Professor Dra. Manuela Pereira de Sousa
Professor Dr. Simão Melo de Sousa

June 11, 2024

Acknowledgements

A special thanks to everyone who has seen me grow and develop over these 3 years of university. It was a pleasure to be able to develop this project with the incredible support of everyone.

To my supervisors, Dra. Manuela Sousa, Dr. António Navara, and Dr. Marco Giunti for all the support and encouragement they have offered me over the last few months, without which I would never have achieved the desired result.

I would like to thank Professor Simão Melo de Sousa, not only for guiding me through this project, which far exceeded my expectations, but also for all the other academic experiences he has given me, like SWERC and MIUP, and for always being an inspiration for his dedication as a teacher and his great taste in music!

To my colleagues and friends, because academic training is not only about studying and working, it is also about the personal relationships you create here and take with you for life. In particular, to António Cruz and Francisco Santos, for the countless hours we spent working, for the conversations that helped in the most difficult moments, and for revising this very document. I learned from you that the best work is done together, may we have many more projects together for the rest of our lives.

To my family for their interest in my work, which they could hardly understand, and for their patience during the most anxious times throughout my training. I know it wasn't always easy, but it would have been impossible without your support.

And lastly, but definitely not least, to my amazing and caring girlfriend, Constança, who is the most incredible support someone could ever have. I know for a fact that my future would have been very different from what it is today without her, be that the Master's position I got because of the great drive she gives me to work, or the happiness she injects at every second of my days with her.

Contents

List of Figures

List of Tables

List of Code Excerpts

Acronyms

AST Abstract Syntax Tree

CCS Calculus of Communicating Systems

COPES *COnsensus Protocols Environments and Specifications*

DSL Domain Specific Language

INRIA *Institut national de recherche en sciences et technologies du numérique*

UBI Universidade da Beira Interior

PBFT Practical Byzantine Fault Tolerance

MOBS MOdular Blockchain Simulator

PoW Proof of Work

PoS Proof of Stake

LTS Labeled Transition System

VM Virtual Machine

Chapter 1

Introduction

For as long as consensus algorithms have existed, there has not been an easy way to define them programmatically, such as with a Domain Specific Language (DSL). This project aims to address that gap. In this document, we present a strong case for the semantic model of such a solution and explore its inner workings.

1.1 Context

This report was conducted as part of Universidade da Beira Interior (UBI)'s Project unit course.
This project is part of the *COnsensus Protocols Environments and Specifications* (COPES) research project, which involves international companies and academic partners and will be carried out with their collaboration/feedback. The COPES research project focuses on the robust design of new consensus algorithms for blockchains.

1.2 Motivation

Consensus algorithms are essential in Distributed Systems to achieve agreement between peers or nodes. Having a clear, simple, and efficient solution for writing and testing them is invaluable for the efficiency at which they can be written.
Writing these algorithms by hand is also very prone to errors, this project aims to provide a way to quickly test if the algorithm is well designed.

1.3 Objectives

The project had the following objectives:

1. Mastery of project-essential concepts:
    a) Understand and research various consensus protocols;
    b) Understand and research Labeled Transition System (LTS), OCaml's modularity and functorization;
    c) Understand the Coq Proof Assistant and the previous work done with it.
2. Test existing OCaml multi-threading libraries for non-determinism, which is essential for these protocols:
    a) Implement mock examples of small versions of existing consensus protocols in various OCaml libraries designed for multi-threading;
    b) Select the library that provides both non-determinism and a good developer experience.
3. Implement a modular Virtual Machine (VM) for the consensus protocols, with various network types available;
4. Implement various examples of protocols in that definition.

1.4 Contributions

The contributions were the following:

1. Successful non-determinism tests for the aforementioned OCaml libraries;
2. Implemented the modular and generalized implementation of the LTS in OCaml;
3. Made it easy to run any kind of consensus protocol from its LTS representation;
4. Created and added the capability to have multiple types of networks;
5. Started specifying various consensus protocols with our definition.

1.5 Document Structure

In order to reflect the work that has been done, this document is structured as follows:

1. The first chapter – **Introduction** – presents the project, the motivation for choosing it, its framework, its objectives, and the organization of the document.
2. The second chapter – **State of the Art and Concepts** – describes the most important concepts in this project, as well as presents previous work and related projects.
3. The third chapter – **Technologies and Tools Used** – presents the technologies that helped us develop this project, specifically important features of OCaml that we could not have done without.
4. The fourth chapter – **Implementation and Testing** – focuses on the practical work done on this project, the tests performed on various libraries, the early ideas, and the system's current implementation.
5. The fourth chapter – **Conclusions and Future Work** – reflects on the work done and what could (and will) be done to extend and complete this project.

Chapter

2

State of the Art and Concepts

2.1 Introduction

This project completely refactors Lupin's previous efforts to create a language that, while conceptually sound, resulted in more of a notation than an actual language. The original approach was innovative but it fell short in execution. In this chapter, we will illustrate the concepts, other similar tools, and the work done to create a great semantic analyzer for Lupin.

2.2 Consensus and Consensus Protocols

Consensus is reaching a majority opinion by everyone involved in a particular distributed network. An example is blockchain mechanisms: an agreement must be made on which blocks to produce, which chain to adopt and to determine the single state of the network. The consensus protocol determines how individual nodes interact with the distributed network to reach a consensus.
Here are a few examples of very renowned protocols:

- **Practical Byzantine Fault Tolerance (PBFT)** [1] - PBFT is a **Byzantine Fault Tolerant** protocol, which means it is resilient against faults like crashes, missing information, and wrong information, purposefully or not, with low algorithmic complexity. Fig. 2.1 describes how PBFT works.
- **Proof of Work (PoW)** [1] - famously adopted by Bitcoin and Ethereum. In PoW a node is selected to create a new block by the computational power it produces on the network in each round of consensus, like a competition. The participating nodes need to solve a cryptographic

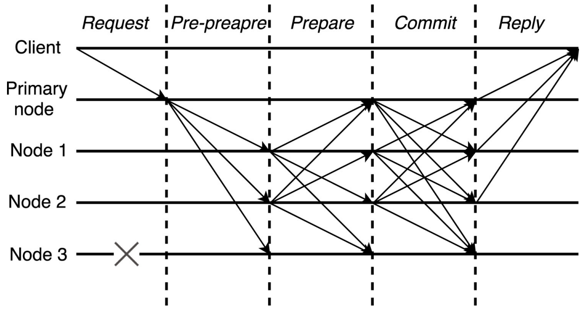

Figure 2.1: PBFT Flow Chart

puzzle. The node that solves the puzzle the quickest creates a new block, earning a reward in the process.

- **Proof of Stake (PoS)** [1] - selecting the node that creates a new block depends on the amount of stake a node holds, rather than its computational power. The stake is normally quantified as the number of coins a node has currently invested.

- **Chandra-Toueg** [2] - a rotating coordinator, round-oriented consensus protocol that features a failure detection system based on an abstract version of timeouts.

2.3 Formal Semantics

A fascinating thing about the formal semantics of a language is that, if done correctly, it defines an interpreter of that language for you free of charge! This is because formal semantics define a precise and unambiguous description of the meaning of the language's constructs, precisely what an Abstract Syntax Tree (AST) must be. In this case, the type of formal semantics we will be using is of the **operational** kind, and it is called LTS.

Operational semantics describe what our program should do step by step, and focuses on describing the transitions between those states in contrast with denotational and axiomatic semantics, which map the language constructs to mathematical objects or logical formulas, respectively, that represent their meanings. These latter two remove the focus on the execution steps.

2.3.1 LTS

A Labeled Transition System (LTS) [3] is a mathematical model used to describe the behavior of systems in terms of states and transitions between those states, where transitions are labeled with actions. Formally, an LTS is defined as a triple $(S, \Lambda, \rightarrow)$ where:

- S is a set of states.
- Λ is a set of labels (representing actions or transitions).
- $\rightarrow \subseteq S \times \Lambda \times S$ is a transition relation.

We can see that it follows exactly our definition of operational semantics:

- **State Transitions:** The transitions between states are directly described, which is a core aspect of operational semantics.
- **Labels as Actions:** The labels in a LTS often represent the execution of specific language constructs or actions.
- **Execution Steps:** By defining states and transitions, a LTS models the operational steps a program takes, aligning with the idea of operational semantics where the focus is on how the program executes.

Furthermore, we will take advantage of the fact that LTS models non-determinism, which is essential for distributed systems and concurrency. The non-determinism happens when two or more rules can be applied at once.

2.3.2 Calculus of Communicating Systems (CCS)

A classical example of a LTS in action is CCS [4]. CCS is a formal language developed around 1980 by Robin Milner, that describes how two processes interact and evolve their executions through communication.

The basic rules can be described as follows:

$$\frac{}{a.P \xrightarrow{a} P} \quad (Act)$$

$$\frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \quad (Sum1)$$

$$\frac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'} \quad (Sum2)$$

$$\frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \quad (Par1)$$

$$\frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'} \quad (Par2)$$

$$\frac{P \xrightarrow{a} P', Q \xrightarrow{\bar{a}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \quad (Com)$$

$$\frac{P \xrightarrow{\alpha} P'}{P \setminus L \xrightarrow{\alpha} P' \setminus L} \quad (\alpha \notin L) \quad (Res)$$

$$\frac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{f(\alpha)} P'[f]} \quad (Rel)$$

In these rules:

- P and Q are processes.
- a and $\bar{a}$ are complementary actions.
- τ represents an internal action.
- α represents an action which can be a, $\bar{a}$, or τ.
- $\bar{L}$ denotes the complement of the set L.
- $P \xrightarrow{\alpha} P'$ means that action α was applied to P and the process transitioned to P'.
- $P_1 + P_2$ is a process that can proceed its execution either as P_1 or P_2.
- $P \mid Q$ means that processes P and Q exist simultaneously.
- $P \setminus L$ is a process that does not contain the actions defined in set L.
- $P[f]$ is the process P with the relabelling function f applied to its actions.

Our semantics take a lot of inspiration from the system above, as we will now discuss.

2.3.3 Our Semantics

The semantic model we will use was developed by Dr. Marco Giunti, from Oxford University. Dr. Marco is also part of the COPES project. In order to make Lupin a concrete language with a proven type system and its own virtual machine, he implemented a modified version of CCS for our purposes in the Coq Proof Assistant.

Code Excerpts 2.1: Small excerpt of the LTS semantics of Lupin

```
(** Lts semantics *)
Inductive lts : configuration -> action ->
                configuration -> Prop :=
| par_left_lts
    M F1 M' F' a F2:
  lts (conf M F1) a (conf M' F') ->
  lts (conf M (parallel_peer F1 F2)) a
      (conf M' (parallel_peer F'  F2))
| par_right_lts
    M F1 M' F' a F2:
  lts (conf M F1) a (conf M' F') ->
  lts (conf M (parallel_peer F2 F1)) a
      (conf M' (parallel_peer F2  F'))
| send_lts
    M G E R i lc m k args E' G' c R':
  G = single_peer E (R, i) ->
  E.(identity) = id_value i ->
  G' = single_peer E' (R', i) ->
  E'.(identity) = id_value i ->
  m = make_msg lc k args ->
  constructor_precondition E lc k output_direction
      = Ret true ->
  constructor_msgcondition E  lc k output_direction m
      = Ret true ->
  constructor_postcondition E' lc k output_direction
      = Ret true ->
  find_value (constructor_channel E lc k output_direction)
      = c ->
  lts (conf M G) (output_action c m) (conf (M @ c&m) G')
| receive_lts
    M G E R i lc m k args E' G' c R':
```

```
    G = single_peer E (R, i) ->
    E.(identity) = id_value i ->
    G' = single_peer E' (R', i) ->
    E'.(identity) = id_value i ->
    m = make_msg lc k args ->
    constructor_precondition E lc k input_direction
        = Ret true ->
    constructor_msgcondition E  lc k input_direction m
        = Ret true ->
    constructor_postcondition E' lc k input_direction
        = Ret true ->
    find_value (constructor_channel E lc k input_direction)
        = c ->
    m  M c ->
    lts (conf M G) (input_action c m) (conf M G')
  | tau_lts
      G M E R i G' lc R' E':
    G = single_peer E (R, i) ->
    E.(identity) = id_value i ->
    G' = single_peer E' (R', i) ->
    E'.(identity) = id_value i ->
    constructor_precondition E lc empty_key tau_direction
        = Ret true ->
    constructor_postcondition E' lc empty_key tau_direction
        = Ret true ->
    lts (conf M G) tau_action (conf M G').
```

This inductive type defines the rules our nodes can apply at every step, along with the pre and post-conditions needed for the rule itself to be valid.
The types and functions needed for this inductive type are available in Appendix A.
An example of a rule that follows this inductive type:

$$\frac{\begin{array}{l} E = \{\,\mathrm{id} = i, \mathrm{phase} = 3, \mathrm{ack} = v, \ldots\,\} \\ m = \mathrm{ABLF}\,\{\,\mathrm{ack} = v, \ldots\,\} \\ E' = E\{\,\mathrm{phase} = 4, \ldots\,\} \end{array}}{M \,@\, E > N_i \;--\; \mathrm{net}\,!\,\mathrm{m} \longrightarrow M + m \,@\, E' > N_i} \quad [MSG\_ACK]$$

This rule in particular is Chandra-Toueg's acknowledge message. The conclusion of this rule can be read as follows:

- The current configuration is mailbox M and entity E, node N_i. Sending m through the network net (net ! m) will result in the message m being appended to the mailbox and a transition of the entity to state E' on node N_i.

Every protocol can be defined using such rules. Appendix B contains three examples, Chandra-Toueg, Bitcoin's PoW, and Algorand.
Our virtual machine and semantic analyzer, then, must be able to convert these types of rules into operations to be performed by all of the nodes on our network.

2.4 Similar Tools

2.4.1 Babel

Babel [5] is a Java framework, created by fellow Portuguese researchers Pedro Fouto, Pedro Ákos Costa, Nuno Preguiça, and João Leitão from NOVA FCT, designed to simplify the development, implementation, and execution of distributed protocols and systems, just like Lupin. Babel uses an event-driven programming model that allows developers to focus on the logic of distributed algorithms while abstracting low-level aspects such as communication management, timeouts, and concurrency issues. In their paper, three main components of the framework are introduced:

- **Protocols:** Protocols work as state machines that evolve based on external events. Each protocol has an event queue containing timers, channel notifications, network messages, and intra-process events. Protocols are executed in dedicated threads, nodes cooperate via message passing.

- **Core:** The central component of Babel coordinates the execution of all protocols within a process. It ensures that interactions between protocols are managed efficiently.

- **Channels:** communication is abstracted through channels, which can be extended or modified by developers. Channels enable different capabilities such as P2P, Client/Server, and failure detection.

Even though Babel and Lupin have some overlapping ideas, like the main objectives, message passing and the channels abstraction, Babel is, at its core, a way to write consensus protocols on real hardware, and does not provide a way to simulate the protocol or check, at compile time, for any errors with the protocol itself, like impossible scenarios or deadlocks. Lupin aims to do that in the future. Babel isn't a testing library, so it doesn't allow developers to introduce faulty or Byzantine nodes, this is also a future feature of Lupin.

2.4.2 Netrix

Netrix [6] is a DSL tool designed to test consensus protocol implementations in distributed systems. Programmers can manipulate the network communication system and introduce faulty nodes. Netrix provides, as one of its standout features, a way to define high-level, scenario-based unit tests.
The primary aim of Netrix is to improve coverage, facilitate robust bug reproduction, and aid in regression testing across different implementation versions of existing protocols, rather than allowing users to create new protocols.

Netrix is extremely good at what it does, and we will definitely take some inspiration from this tool for Lupin's future work, mainly:

- **Network Filters**: These filters allow developers to specify conditions for message delivery or faults (e.g., dropping, delaying, or modifying messages), analogous to how the Linux's `IPTables` tool works. This feature in particular allows programmers to simulate Byzantine faults.

- **Randomized Exploration**: Netrix uses algorithms like PCTCP, which have been proven to be effective at exploring non-deterministic execution paths and finding bugs by random sampling of message and fault orderings.

This tool primarily focuses on testing existing distributed systems' consensus protocols rather than allowing users to create new protocols like Lupin aspires to do. That being said, we can learn from Netrix's excellent testing capabilities for when the time comes to add those features to Lupin.

2.4.3 Tezos' TezTale

TezTale [7] is an open-source monitoring tool for Tenderbake, Tezos' consensus algorithm. Developed by Nomadic Labs, it allows users to track consensus processes, visualize data from Tezos nodes, and inspect past and current network states.
Teztale was originally built for incident response and provides real-time insights and detailed analytics on block proposals, voting phases, and delegate performance. It can graph all of this information and much, much more.

It is a truly amazing tool, and we will take inspiration from its node analysis at a later stage of the project.

2.5 MOdular Blockchain Simulator (MOBS)

MOBS [8] is a blockchain simulator developed by Miguel Alves as his Master Thesis.
It features a web interface that makes it easy to visualize, change, and analyze the progress of a blockchain.
In the future, it is expected that this project will be able to generate code that can be used as input into this visualizer for the user to observe and control the consensus protocol.

2.6 Conclusions

In this chapter, we explored the evolution of Lupin, focusing on the transition from a lackluster semantic analyzer to one fully-fledged language.
We examined various consensus protocols and detailed the creation of a robust semantic analyzer with a strong case for operational semantics and their benefits, as well as LTS and CCS's best characteristics and why they work well for our use case.
We also explored other similar tools and provided reasoning for why our DSL is different, or what we should take as inspiration to make Lupin better.

Chapter

3

Technologies and Tools Used

3.1 Introduction

This chapter provides an overview of the technologies and tools utilized in our project. It covers the OCaml programming language, its specific tools and libraries used, version control practices, and the design principles that guided our approach.

3.2 OCaml

OCaml is a functional programming language developed at the *Institut national de recherche en sciences et technologies du numérique* (INRIA), it first appeared in 1996.
OCaml provides developers with incredible features, a lot of which we will be using in this paper, such as:

1. **Static, strong inferred typing**;
2. **Parametric polymorphism and algebraic data types**;
3. **Functors and Modules**;
4. **Execution safety**;
5. **Garbage collection**;
6. **Amazing standard library**, with, importantly, a good implementation of randomness;
7. **Great libraries and features for multicore**, new in OCaml 5;

8. **Vast and useful ecosystem of packages and tools**;

The OCaml version used in the context of this project was `5.1.1`, chosen specifically to take advantage of the new multicore features mentioned previously.

3.3 OCaml Specific Tools

3.3.1 *Dune* and *Opam*

Dune is the defacto build system for OCaml projects. It is fast and parallel, and has no system dependencies, meaning you only need OCaml itself to build *Dune* and new packages. *Dune* is composable, meaning you can compose *Dune* projects together, which will be useful when Francisco Santos, who is building the parser for Lupin, and I combine efforts.

Opam is the package manager for OCaml, it allows easy installation of packages, and other OCaml versions through what they call *switches*.

3.3.2 OCaml Libraries

The recent efforts to extend multicore functionality in OCaml resulted in multiple approaches to the problem, of which, we tested four:

- ***Domains*:** Low-level parallel programming primitives provided by OCaml's standard library.
- ***Domainslib*:** Provides data and control structures for parallel programming, such as an abstraction for Channels much like *Go*'s. Extends *Domains* in meaningful ways, such as adding task pools and async-await.
- ***eio*:** Implements effects-based direct-style IO. Instead of parallelism, *eio* focuses on concurrency, and implements green threads that are scheduled by OCaml's runtime.
- ***Saturn*:** Implements high performance lock-free and parallelism-safe data structures to be used alongside the aformentioned libraries.

As we will see in section 4.2.1, after extensive testing we eventually decided on using *Domainslib*, mostly because of the Channels implementation and ease of use.

3.3.3 *Git* and GitLab

Git is an infamous version control system that sees extensive use in the computer science field.
GitLab is a *git* server and DevOps platform that the *RELEASE* lab uses for its projects, this one is no different.

3.4 Design Principles

3.4.1 Modules and Functors in OCaml

Modules in OCaml are the main way to organize code. They are collections of definitions grouped together. Functors are functions at the module level.

All OCaml files are modules by themselves, but we can create modules, along with their type signature, with this syntax:

Code Excerpt 3.1: Module example

```
module type NameT = sig
    type t
    val x : t
    val f : unit -> t
end

module Name : NameT = struct
    type t = string
    let x = "Lupin"
    let f ( ) = x
end
```

In the context of Category Theory, we can think of modules as categories and functors as functions that map categories to other categories.

To put it simply, we can use functors to transform modules into other modules.

Here is a simple example:

Code Excerpt 3.2: Functor example

```
module type X = sig
  val x : int
end

module IncX (M : X) = struct
  let y = M.x + 1
```

```
end
```

3.4.2 Our Modular Approach

We need to be able to have multiple network topologies and different rules for each protocol. This is a perfect scenario for modules, as they allow for interchangeability.
Suppose we create multiple *Net* modules, all with the same signatures. In that case, we can abstract the implementations from the actual nodes of the protocol and interchangeably use one or the other. The same goes for a set of *Rule* modules.
If we want to test some other set of rules, we can just input another *Rule* module into the *Node* functor. Figure 3.1 illustrates an example of what was just described.

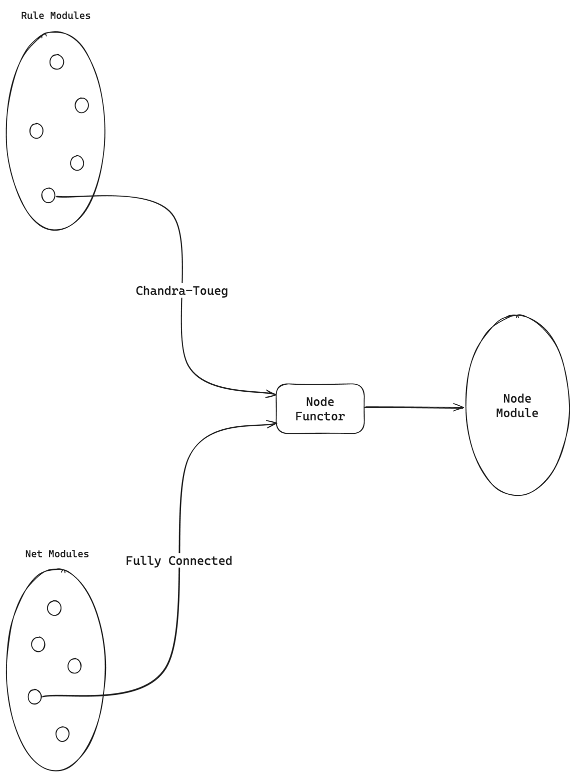

Figure 3.1: *Node* Module Functor

In the context of Category Theory, we can think of *Rule Modules* and *Net Modules* as a category of categories, where we choose one category according to our needs.

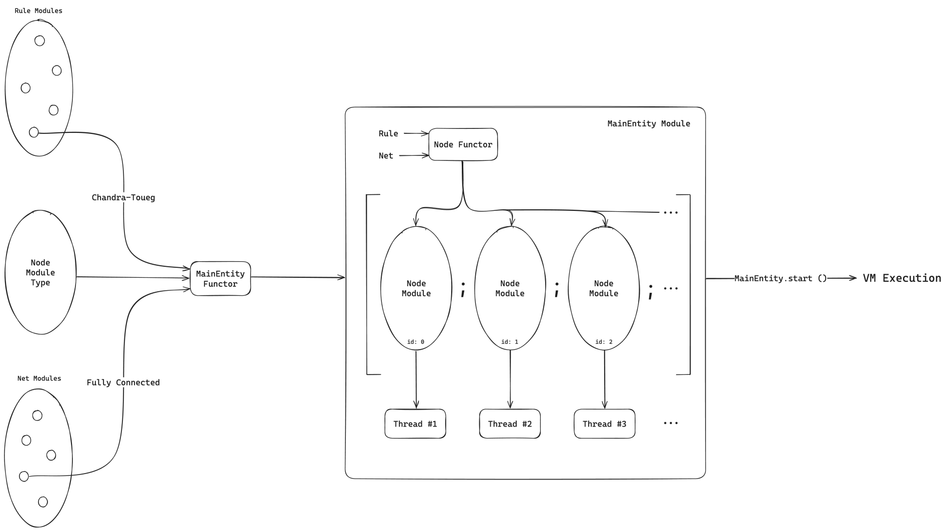

Figure 3.2: *MainEntity* Module Functor

In practice, the system is a little more complicated. We need to have a way to create multiple nodes, create threads for each, and assign them their respective peers, which come from the network topology. We call the module that creates the network and the nodes the *MainEntity*. Figure 3.2 illustrates an example of the system in a simplified way.

3.4.3 Advantages and Limitations

This approach has very clear advantages:

1. **Abstraction**: As discussed previously, modules allow us to abstract implementations and only provide the necessary interfaces for our purposes.

2. **Interchangeability**: Functors allow us to switch between modules based on requirements without modifying the dependent code.

3. **Type Safety**: OCaml's type system ensures that module interfaces are correctly adhered to, catching many errors at compile time.

4. **Separation of Concerns** and **Scalability**: Separate modules are easier to maintain, extend, and scale.

But also has limitations:

1. **Abstraction is Hard**: It is hard to know when and what to abstract and when and what not to.

2. **Debugging**: This can be challenging because the source of an issue might be hidden behind module interfaces.

3.4.4 But why a VM?

Part of the semantic analyzer's job is to check if the code written by the user makes sense logically and mathematically, and adheres to the rules of the language itself. Type-checking in this case is easy, and although the topic isn't covered here, a uni-directional one would work perfectly, it is the verification that the protocol written by the user is logically sound that is more difficult.

We believe a good way to verify if a protocol doesn't have deadlocks, livelocks or impossible rules is to actually **run** that protocol on real hardware. This approach allows us to catch early errors, provide running protocol logs to the user, and simulate different network conditions and Byzantine nodes if the user asks us to!

3.5 Conclusions

At this point, we now know the necessary concepts, technologies, and tools used for the duration of this project. This chapter was dedicated to introducing OCaml and its new features which made this project possible, and the libraries and version control system used. We also talked about modules and functors, an essential part of the next chapter. Let us now see how it all fits together!

Chapter

4

Implementation and Testing

4.1 Introduction

Chapter 4 contains the most practical aspects of the project. Let's focus on the implementation and testing of the aforementioned libraries, the proposed solution, provide a comprehensive overview of the adopted methodologies and the step-by-step processes of implementing a theoretical scenario simply by writing LTS rules like the ones we saw in 2.3.3!

4.2 Implementation Details

4.2.1 Testing for Non-Determinism

The first step in our efforts was to figure out which OCaml 5 library was the best in terms of developer experience and performance, and if we could recreate non-determinism. We decided to implement some small scenarios to test them out.

4.2.1.1 Scenario 1: Sending a list of numbers

This scenario is simple: we have a single producer single consumer queue that we populate on the first thread with the contents of a list, the other thread then receives those values, adds 5 to each, and sends them back.

Code Excerpt 4.1: Scenario 1: *eio*

```
open Eio.Std
module C =
   Saturn.Single_prod_single_cons_queue

let q1 = C.create ~size_exponent:5
let q2 = C.create ~size_exponent:5
let lst = [ 1; 2; 3; 4; ...; 10 ]

let main _c =
  Switch.run ~name:"main" @@
  fun sw −> Fiber.fork ~sw
  (fun () −>
      List.iter
      (fun el −>
         C.push q1 el) lst;
      traceln "t1 finished sending";
      Fiber.yield ();
      let rec h () =
        match C.pop q2 with
        | Some n −>
            traceln "t1: read %d" n;
            h ()
        | _ −> ()
      in
      h ());
  Fiber.fork ~sw (fun () −>
      let rec h () =
        match C.pop q1 with
        | Some n −>
            C.push q2 (n + 5);
            traceln "t2: sending %d" n;
            h ()
        | _ −> traceln
               "t2 finished sending"
      in
      h ())

let () = Eio_main.run main
```

Code Excerpt 4.2: Scenario 1: *domains*

```
open Eio.Std
module C =
   Saturn.Single_prod_single_cons_queue

let q1 = C.create ~size_exponent:5
let q2 = C.create ~size_exponent:5
let lst = [ 1; 2; 3; 4; ...; 10 ]

let () =
  let t1 =
    Domain.spawn (fun () −>
        List.iter
            (fun el −> C.push q1 el) lst;
        traceln "t1 finished sending";
        let rec h () =
          match C.pop q2 with
          | Some n −>
              traceln "t1: read %d" n;
              h ()
          | _ −> h ()
        in
        h ())
  in
  let t2 =
    Domain.spawn (fun () −>
        let rec h () =
          match C.pop q1 with
          | Some n −>
              C.push q2 (n + 5);
              traceln "t2: sending %d" n;
              h ()
          | _ −> traceln
                 "t2 finished sending"
        in
        h ())
  in
  Domain.join t1;
  Domain.join t2
```

We expect the *eio* version to be somewhat deterministic, because we can control when the OCaml scheduler hands control to another green thread (fiber) using `Fiber.yield`. This means that the execution will run on one thread until `Fiber.yield` gets called or the OCaml scheduler decides to change fibers (which doesn't happen very often, normally happens if the current thread isn't doing anything meaningful), resulting in the first thread adding all numbers to the queue before calling `Fiber.yield` and the second thread popping numbers from it and adding 5 to them, then adding them back and finishing its execution. The first thread picks up after the second thread finishes, reads all numbers from the queue, and prints them. This almost always happens and the output is the same as the one in table 4.1.

The *domains* version is different because the threads aren't concurrent and their execution is handled by the operating system. Non-determinism is exhibited, and the results always change compared to table 4.1.

eio	***domains***
+t1 finished sending	+t1 finished sending
+t2: sending 1	+t1: read 6
+t2: sending 2	+t2: sending 1
+t2: sending 3	+t1: read 7
+t2: sending 4	+t2: sending 2
+t2: sending 5	+t2: sending 3
+t2: sending 6	+t1: read 8
+t2: sending 7	+t2: sending 4
+t2: sending 8	+t2: sending 5
+t2: sending 9	+t1: read 9
+t2: sending 10	+t1: read 10
+t2 finished sending	+t1: read 11
+t1: read 6	+t2: sending 6
+t1: read 7	+t2: sending 7
+t1: read 8	+t1: read 12
+t1: read 9	+t2: sending 8
+t1: read 10	+t2: sending 9
+t1: read 11	+t2: sending 10
+t1: read 12	+t2 finished sending
+t1: read 13	+t1: read 13
+t1: read 14	+t1: read 14
+t1: read 15	+t1: read 15

Table 4.1: Results of Scenario 1

4.2.1.2 Scenario 2: Task pools

As mentioned previously, *Domainslib* has primitives for running tasks in a pool of threads. *eio* also allows this using `Switch`, which groups fibers and other resources together.

This scenario is also quite simple. Two arrays with 1000 random numbers are created, and two threads will each add up numbers with the same index and create a new, shared array with those sums.

The goal of this scenario is to test the performance of both implementations.

Table 4.2 contains the average times taken for running the main function 1000 times (without I/O operations) for each implementation.

eio	***domainslib***
≈ 0.000038s	≈ 0.000182s

Table 4.2: Time taken in Scenario 2

These results were achieved on a Macbook M1 Air.

As we can clearly see, *eio* is 4 to 5 times faster than *Domainslib*, possibly because *eio*'s threads are lighter and take less time to create and manage. The *eio* code in 4.3 is also a tiny bit shorter than the *Domainslib* version (4.4)

The performance gain with *eio* is, in this case, significant but not noticeable. In the next section, we will discuss why *Domainslib* was preferred.

Code Excerpt 4.3: Scenario 2: *eio*

```
open Eio.Std

let () = Random.init
    (Unix.time () |> int_of_float)

let sum p u arr1 arr2 res =
  for i = p to u − 1 do
    res.(i) <− arr1.(i) + arr2.(i)
  done;
  ()

let main () =
  let size = 1000 in
  let arr1 = Array.init size
      (fun _ −> Random.int 1000) in
  let arr2 = Array.init size
      (fun _ −> Random.int 1000) in
  let res = Array.make size 0 in
  ( Eio_main.run @@ fun _env −>
    Switch.run @@ fun sw −>
    Fiber.fork ~sw (fun () −>
        sum 0 (size / 2) arr1 arr2 res);
    Fiber.fork ~sw (fun () −>
        sum (size / 2) size arr1 arr2 res)
  );

  for i = 0 to size − 1 do
    Format.printf "%d = %d + %d@."
      res.(i) arr1.(i) arr2.(i)
  done
```

Code Excerpt 4.4: Scenario 2: *Domainslib*

```
let () = Random.init (Unix.time ()
                    |> int_of_float)

let sum p u arr1 arr2 res =
  for i = p to u − 1 do
    res.(i) <− arr1.(i) + arr2.(i)
  done;
  ()

let () =
  let size = 1000 in
  let arr1 = Array.make size 0
      |> Array.map (fun _ −>
         Random.int 1000) in
  let arr2 = Array.make size 0
      |> Array.map (fun _ −>
         Random.int 1000) in
  let res = Array.make size 0 in
  let pool = Domainslib.Task.setup_pool
      ~num_domains:2 () in
  let main () =
    Domainslib.Task.run pool (fun () −>
        let a =
          Domainslib.Task.async pool
            (fun () −>
             sum 0 (size / 2) arr1 arr2 res)
        in
        let b =
          Domainslib.Task.async pool
            (fun () −>
             sum (size / 2) size arr1 arr2 res)
        in
        Domainslib.Task.await pool a;
        Domainslib.Task.await pool b)
  in
  Domainslib.Task.teardown_pool pool;
  for i = 0 to size − 1 do
    Format.printf "%d = %d + %d@."
      res.(i) arr1.(i) arr2.(i)
  done
```

4.2.1.3 Experiments: *Domainslib*'s Channels

At this point we still don't know if interactions between the threads themselves are non-deterministic. For example, if we have 4 threads, 2 of them receivers and the other 2 senders, we aren't supposed to know which thread will receive the messages first and what messages it receives. This behavior is essential for modeling the internet like we aim to do with our *Net* modules.

We decided to check if *Domainslib*'s Channels exhibit this behavior because they resemble *Go*'s Channels, which do.

Appendix C mentions three different experiments:

1. **Unbounded Channels** (C.1): 4 threads each send 4 values through one single Channel, and another thread reads values from that Channel;

2. **Bounded Channels** (C.2): the same as the previous experiment but with bounded Channels.

3. N **Senders and Receivers** (C.3): N threads send values to a single Channel, and N threads receive values from it. If N equals 1, the execution should be deterministic, else, the receivers should each receive any of the numbers in the channel with $\frac{1}{N}$ probability.

Table 4.3 demonstrates the results from C.1 on 3 different runs, we can see that non-determinism happens between runs. C.2 exhibited the same behaviour.

Table 4.4 demonstrates the result for 4 senders/receivers and 1 sender/receiver, and shows that the expected behavior is confirmed! If N equals 1, the numbers get sent and received in order, if else, the numbers will get completely mixed.

These results are satisfactory, and the developer experience working with *Domainslib* is great! Because of this, we chose *Domainslib* for our VM's threads and the communication between them.

Run #1	***Run #2***	***Run #3***
Sending from id: 3 value: 6	Sending from id: 1 value: 0	Sending from id: 1 value: 0
Sending from id: 1 value: 0	Sending from id: 3 value: 6	Sending from id: 2 value: 3
Sending from id: 1 value: 1	Sending from id: 3 value: 7	Sending from id: 3 value: 6
Sending from id: 1 value: 2	Sending from id: 1 value: 1	Received: 0 | Times: 0
Sending from id: 2 value: 3	Sending from id: 3 value: 8	Received: 3 | Times: 1
Sending from id: 4 value: 9	Sending from id: 2 value: 3	Received: 6 | Times: 2
Sending from id: 3 value: 7	Sending from id: 2 value: 4	Sending from id: 2 value: 4
Sending from id: 4 value: 10	Sending from id: 2 value: 5	Sending from id: 1 value: 1
Sending from id: 3 value: 8	Sending from id: 1 value: 2	Sending from id: 2 value: 5
Sending from id: 2 value: 4	Received: 0 | Times: 0	Sending from id: 1 value: 2
Sending from id: 4 value: 11	Sending from id: 4 value: 9	Received: 4 | Times: 3
Sending from id: 2 value: 5	Sending from id: 4 value: 10	Received: 1 | Times: 4
Received: 6 | Times: 0	Sending from id: 4 value: 11	Sending from id: 4 value: 9
Received: 0 | Times: 1	Received: 6 | Times: 1	Received: 2 | Times: 5
Received: 1 | Times: 2	Received: 7 | Times: 2	Sending from id: 4 value: 10
Received: 2 | Times: 3	Received: 8 | Times: 3	Sending from id: 3 value: 7
Received: 9 | Times: 4	Received: 3 | Times: 4	Sending from id: 4 value: 11
Received: 10 | Times: 5	Received: 4 | Times: 5	Sending from id: 3 value: 8
Received: 7 | Times: 6	Received: 1 | Times: 6	Received: 9 | Times: 6
Received: 8 | Times: 7	Received: 5 | Times: 7	Received: 10 | Times: 7
Received: 3 | Times: 8	Received: 2 | Times: 8	Received: 5 | Times: 8
Received: 4 | Times: 9	Received: 9 | Times: 9	Received: 7 | Times: 9
Received: 11 | Times: 10	Received: 10 | Times: 10	Received: 11 | Times: 10
Received: 5 | Times: 11	Received: 11 | Times: 11	Received: 8 | Times: 11

Table 4.3: Results for Unbounded Channel Tests

$N = 4$	$N = 1$
Initial Array: 0 1 2 3 4 5 6 7 8 9 10 11	Initial Array: 0 1 2 3 4 5 6 7 8 9 10 11
Sending from id:0 value:0	Sending from id:1 value:0
Sending from id:1 value:3	Sending from id:1 value:1
Sending from id:1 value:4	Sending from id:1 value:2
Sending from id:1 value:5	Sending from id:1 value:3
Sending from id:0 value:1	Sending from id:1 value:4
Received on id:1 value:3	Sending from id:1 value:5
Received on id:0 value:4	Sending from id:1 value:6
Sending from id:3 value:9	Sending from id:1 value:7
Sending from id:3 value:10	Sending from id:1 value:8
Sending from id:3 value:11	Sending from id:1 value:9
Received on id:2 value:5	Sending from id:1 value:10
Received on id:2 value:9	Sending from id:1 value:11
Received on id:2 value:10	Received: 0 | Times: 0
Sending from id:2 value:6	Received: 1 | Times: 1
Received on id:0 value:11	Received: 2 | Times: 2
Received on id:0 value:6	Received: 3 | Times: 3
Received on id:3 value:0	Received: 4 | Times: 4
Sending from id:2 value:7	Received: 5 | Times: 5
Received on id:1 value:1	Received: 6 | Times: 6
Sending from id:2 value:8	Received: 7 | Times: 7
Sending from id:0 value:2	Received: 8 | Times: 8
Received on id:3 value:7	Received: 9 | Times: 9
Received on id:3 value:2	Received: 10 | Times: 10
Received on id:1 value:8	Received: 11 | Times: 11
Lock-free Stack: 0 3 4 5 1 9 10 11 6 7 8 2	Lock-free Stack: 0 1 2 3 4 5 6 7 8 9 10 11
Channel: 4 11 6 3 1 8 5 9 10 0 7 2	Channel: 0 1 2 3 4 5 6 7 8 9 10 11

Table 4.4: Results for N Senders and Receivers

4.2.1.4 Scenario 3: Small Blockchain with *Domainslib*

Now that we know that, with *Domainslib*:

a) the thread executions are non-deterministic;

b) and the Channel interactions are non-deterministic,

we can start to experiment with scenarios that are closer to what we expect from Lupin.

For this scenario, implemented in Appendix D, we assumed a simple blockchain without verification of the blocks to be added. In this example, we can send a block to other nodes and they will always accept and broadcast it to other nodes, except if the block is already present in their blockchain.

The LTS rules for this scenario are only two:

$$\frac{\begin{array}{l} E = \{\,\text{id} = i,\ \text{blockchain} = \text{bn} + \text{Block}\,\{\,\_, h\,\},\ \ldots\,\} \\ m = \text{Block}\,\{\,i, h\,\} \\ c = \text{neighbours}\ i \end{array}}{M @ E \ > \ N_i \ -- \ \text{c}\,!\,\text{m} \longrightarrow M + m * c @ E \ > \ N_i} \quad [MSG\_SEND]$$

$$\frac{\begin{array}{l} E = \{\,\text{id} = i,\ \text{blockchain} = \text{bn}\ \ldots\,\} \\ c = \text{neighbours}\ i \\ m = \text{Block}\,\{\,j, h\,\} \\ m * c \in M \\ m \notin bn \\ E' = E\,\{\,\text{blockchain} \leftarrow \text{bn} + m,\ \ldots\,\} \end{array}}{M @ E \ > \ N_i \ -- \ \text{c}\,?\,\text{m} \longrightarrow M @ E' \ > \ N_i} \quad [MSG\_RCV]$$

And the network topology is illustrated by Figure 4.1.
This scenario served as a warm-up and as a way to consolidate the concepts learned and tested until this point.

4.2.1.5 Conclusions and Insight Gained

With the libraries tested for non-determinism and *Domainslib* chosen among them, we are now ready to begin the second stage of the project: testing the modular ideas we presented in 3.4.2.

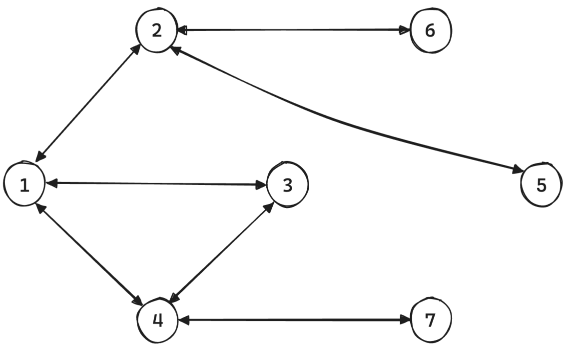

Figure 4.1: Scenario 3 Network Topology

4.2.2 Preliminary Modular Approach

Our first approach was not as fleshed out as the one presented at 3.4.2. We hadn't yet thought of the *Rule* module, and instead embedded the execution steps into the *Node* module itself like Figure 4.2 illustrates. We also hadn't yet properly thought of a way to use the LTS definition to use rules like the ones in Appendix B to write the protocol.

Next, we will see why this approach didn't work for our purposes.

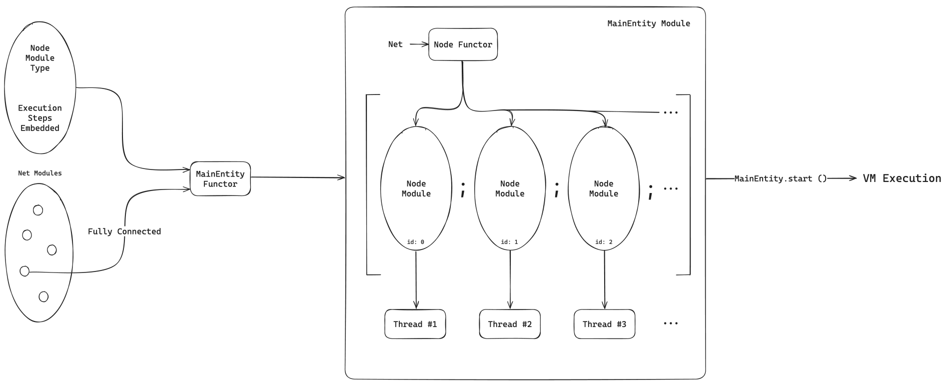

Figure 4.2: Preliminar Modular Structure

4.2.2.1 Implementation Details

To test our initial ideas, a version of Chandra-Toueg was implemented with this modular approach, available in Appendix E.

If we run the protocol and, inside the `MainEntity` module, send spontaneous messages to all the nodes, they will agree on the value with the majority vote.

At this point, we were relying too much on the idea of code generation. What had to happen for this approach to be possible was that all of the code inside the `Ct` module (the *Node* definition) needed to be generated by an intermediate layer between the parser and the interpreter. Ideally, the *Node* definition should be generalized and not generated, because that would create too many moving parts and a lot of development overhead.

Another problem is that the spontaneous messages are being sent from the `MainEntity` module, the main thread. This is an issue, it isn't correct to assume spontaneous messages need to be sent over a channel to a node, instead, they need to be created by the node itself depending on its internal state.

4.2.2.2 Conclusions and Insight Gained

Although this approach had its fair share of problems, we learned from them to create the structure defined in 3.4.2. Something this approach did well was the *Net* module, which was something we kept and will see in action in the next chapter.

4.2.3 LTS + Modular Implementation

The newest version of the modular structure solves the previously mentioned issues. We will see that code generation by the middle layer is no longer necessary to such an extent, as the required code now matches the AST of our language, spontaneous messages function correctly, and support for an alarm (local timeouts) system was added.

4.2.3.1 Types

The conversion of the types from the Coq code in Appendix A to OCaml resulted in the following definitions:

Code Excerpt 4.5: Type definitions

```
type lupin_constr = Constructor of string
type role = Role of string

type message = Make of lupin_constr * string * arguments
and arguments = value list
(* and channel = *)

and value =
  | Id of int
```

```
  | Nat of int
  | Number of float
  | String of string
  | Bool of bool
  | BoolOp of operator * value * value
  | BoolPredicate of predicate * value
  | Messages of message list

and operator =
  | NatOp : (int -> int -> bool) -> operator
  | BoolOp : (bool -> bool -> bool) -> operator

and predicate = BoolP of bool * bool

type entity = {
  identity : value;
  mutable f1 : (string * value) option;
  mutable f2 : (string * value) option;
  mutable f3 : (string * value) option;
  mutable f4 : (string * value) option;
  mutable f5 : (string * value) option;
  mutable f6 : (string * value) option;
  mutable f7 : (string * value) option;
  mutable f8 : (string * value) option;
  mutable f9 : (string * value) option;
  mutable f10 : (string * value) option;
}

type configuration = Conf of entity * role * peer
and peer = Single of entity * (role * int) | Parallel of peer * peer

type action = Tau | Input of message | Output of message
type direction = Input | Output | Tau
type lts = ParLeft | ParRight | Send | Receive | Tau
type event = Lts of lts | Spontaneous | Alarm of float
```

These types will be used to construct the rules for each protocol. We also have a new type, `event`, which can model spontaneous and alarm (through timeouts) events inside of those rules.

The `entity` type is a generic type for entities, it holds 10 fields of pairs of strings (the name of the field) and values (the mapping of that field). For example, if we want an entity to have a list of messages that it has received, and to hold the value of the current round, we can add the following fields:

```
entity.f1 <– Some ("msg_list", Messages [ a; b; c; ... ])
entity.f2 <– Some ("round", Nat 3)
```

4.2.3.2 *Node* and *MainEntity* Modules

This is where the magic happens. Contrary to the other modules that follow, these two will not have generated code. Their implementation can be found in Appendix F.1.

The signatures for both are as follows:

Code Excerpt 4.6: Type signatures for *Node* and *MainEntity* modules

```
open Types

module type Node_f = functor
  (Net : module type of Net)
  (Rules : module type of Rules)
  –> sig
  type 'a net = 'a Net.t

  val self : entity ref
  val role : role ref
  val in_buffer : message option UnorderedSet.t
  val out_buffer : message option UnorderedSet.t
  val rules : Rules.rules option ref
  val init : int –> unit
  val set_peer : entity –> role –> int –> unit
  val make_rules : unit –> unit
  val run : unit –> unit
end

module type MainEntity = functor
  (Net : module type of Net)
  (Rules : module type of Rules)
  (Node : Node_f)
  –> sig

  val start : unit –> unit
end
```

The nodes have two buffers, one for incoming and another for outgoing messages. The messages are stored in an *UnorderedSet* (implementation in Appendix F.2), which offers a primitive to remove a random element. This is how we simulate the non-deterministic behavior of message ordering in real network systems.

The `run` function calls `send_rcv`, which is a recursive loop whose execution can be described as follows:

1. Calculate the difference between the current time and the time that the alarm was started, if it is active;
2. Try removing a random message from the buffers;
3. Refresh the rule list with the new information;
4. Filter the rule list for:
   a) Sending rules, if there are outgoing messages;
   b) Receiving rules, if there are incoming messages;
   c) Spontaneous rules, which are always considered;
   d) Alarm rules, if the alarm has rung (if the difference is bigger than the maximum specified in the rule).
5. Filter the rules that are possible given the current configuration;
6. Check if the remaining rule list is empty:
   a) If so, reintroduce any messages that were removed from the buffers into them again;
   b) If not, choose one at random and execute it.
7. Go back to step 1.

Listing 4.7 is an excerpt of the implementation of this system. The *Rules* module functions will be explained further in 4.2.3.5.

Code Excerpt 4.7: Excerpt of `send_rcv` loop function

```
let rec send_rcv () =
  let t =
    match !time with
    | true, tr ->
        let tmp = Unix.gettimeofday () -. tr in
        Format.printf "%d: alarm time: %f@." (!self.identity |> get_id) tmp;
        (true, tmp)
    | false, _ -> !time
  in
```

```
try_receive !!net;
let in_msg_opt = UnorderedSet.remove_random in_buffer in
let out_msg_opt = UnorderedSet.remove_random out_buffer in
match (in_msg_opt, out_msg_opt) with
| Some in_msg, Some out_msg ->
    (* Create rules with the current configuration *)
    rules :=
      Some (Rules.make_rules !self role !!peer time !!net in_msg out_msg
            out_buffer);

    (* Because we have incoming and outgoing messages
       let's filter with both Lts Send and Lts Receive
       and concatenate the results *)
    let filtered =
      Rules.filter_possible (Lts Send) !self !role !!peer t out_msg
        !!rules
      @ Rules.filter_possible (Lts Receive) !self !role !!peer t in_msg
          !!rules
    in
    let len = List.length filtered in

    if len = 0 then (
      (* There aren't any applicable rules. Let's
         add back the messages we removed from the buffers *)
      UnorderedSet.add in_buffer in_msg;
      UnorderedSet.add out_buffer out_msg;
      send_rcv ());
    (* We have applicable rules, let's choose one and run it *)
    (match List.nth filtered (Random.int len) with _, _, _, f -> f ());
    send_rcv ()
| Some in_msg, None ->
  ...
```

The variable `time` holds information about the alarm. It is a tuple of a boolean, true if the alarm is on, and a float, which holds the time at which the alarm was started or ended.

The function `try_receive` is a non-blocking function that tries to read the node's channel. If there is a message to be read it places it inside the incoming buffer, if not it returns.

The *MainEntity* module is responsible for creating the threads that run each node, their peer assignment, and the network that connects them, as we saw in 3.4.2.

Code Excerpt 4.8: *MainEntity* implementation

```
let n = 5
let net : message Net.t option ref = ref None

...

module MainEntity
    (Net : module type of Net)
    (Rules : module type of Rules)
    (Node : Node_f) =
struct
  let start () =
    let arr = Array.make n 0 in
    let arr =
      Array.map (fun _ -> (module Node (Net) (Rules) (Common) : Node_t)) arr
    in
    net := Some (Net.create n);
    let peers = Net.peers n in

    List.iter (fun (a, b) -> Format.printf "%d -> %d@." a b) peers;

    (* Node initialization *)
    Array.iteri (fun i (module M : Node_t) -> M.init i) arr;

    (* Node initialization part two, peers and rules *)
    Array.iteri
      (fun i (module M : Node_t) ->
        let _, peer_id = List.nth peers i in
        let (module Peer : Node_t) = arr.(peer_id) in
        M.set_peer !Peer.self !Peer.role peer_id;
        M.make_rules ())
      arr;
    let pool = Domainslib.Task.setup_pool ~num_domains:5 () in
    Domainslib.Task.run pool (fun () ->
        let async = Domainslib.Task.async in
        let await = Domainslib.Task.await in
        let thread_arr =
          Array.map (fun (module M : Node_t) -> async pool M.run) arr
        in
        Array.iter (await pool) thread_arr)
end
```

The `start` function creates the network and the peer list, then creates an array populated with nodes. We loop through that array once to initialize them, and a second time to set their peers and create the base rules. After that, we create a task pool where we iterate through the array once more and run each node in a separate thread.

4.2.3.3 *Common* Module

This is where we define helper functions for our protocols. It is imported everywhere to be used in any of the other modules. It can be generated automatically. Here is an example of this module for Chandra-Toueg:

Code Excerpt 4.9: Helper functions for Chandra-Toueg

```
open Types

let ( !! ) s = !s |> Option.get

let new_entity () =
  {
    identity = Id (−1);
    f1 = Some ("phase", Nat 1);
    f2 = Some ("round", Nat 0);
    f3 = Some ("belief", Nat 0);
    f4 = Some ("votes", Messages []);
    f5 = Some ("granted_votes", Messages []);
    f6 = Some ("ack", Bool false);
    f7 = Some ("current_leader", Id (−1));
    f8 = None;
    f9 = None;
    f10 = None;
  }

let new_role () = Role "NON_COORDINATOR"
let get_id = function Id id −> id | _ −> assert false
let get_constr = function Make (l, _, _) −> l
let get_key = function Make (_, k, _) −> k
let get_args = function Make (_, _, a) −> a
let get_peer_entity = function Single (e, _) −> e | _ −> assert false

let get_field entity k =
  let lst = [
      entity.f1;
      entity.f2;
      entity.f3;
```

```
      entity.f4;
      entity.f5;
      entity.f6;
      entity.f7;
      entity.f8;
      entity.f9;
      entity.f10;
    ] in
  List.find_opt
    (fun a ->
      match a with Some (ident, _) when ident = k -> true | _ -> false)
    lst
  |> fun a -> match a with Some (Some (_, f)) -> Some f | _ -> None
```

The only functions that need to change compared to other protocols are the `new_entity` and `new_role` functions, the rest can stay the same and are extremely useful in other parts of our program. User-defined functions inside Lupin can also live here, and be used anywhere.

4.2.3.4 *Net* Modules

Net modules need to follow this signature:

Code Excerpt 4.10: *Net* modules signature

```
module type Net = sig
  (* Channels *)
  module C : sig
    type 'a t
    val make_unbounded : unit -> 'a t
    val send : 'a t -> 'a -> unit
    val recv : 'a t -> 'a
    val recv_poll : 'a t -> 'a option
  end

  type 'a t = (int, 'a C.t) Hashtbl.t

  val create : int -> 'a t
  val add : 'a t -> int -> unit
  val size : 'a t -> int

  val send_all : 'a t -> int -> 'a -> unit
  val send : 'a t -> int -> int -> 'a -> unit
  val recv : 'a t -> int -> 'a
```

```
  val recv_opt : 'a t -> int -> 'a option
  val recv_poll : 'a t -> int -> 'a option

  val peers : int -> (int * int) list
end
```

Appendix F.5 and F.6 are two examples of these modules, the first is a fully connected undirected graph and the second is a partially connected directed graph.

The `peers` function generates peer pairs following the network size and topology. In the future, Lupin users should be able to define their own network topology and peers or select one from these implementations. By default, the fully connected module would be used. Currently, the topology is chosen manually.

4.2.3.5 *Rule* Modules

This module defines the rules of our protocols. It is part of the AST that Francisco's parser will match.

Rule modules need to follow this signature:

Code Excerpt 4.11: *Rule* modules signature

```
module type Rules = sig
    type t = event * configuration * lupin_constr option * (unit -> unit)
    type rules = t list

    val make_rules : entity -> role ref -> peer ->
                    'a -> (int, message Net.C.t) Hashtbl.t ->
                    message option -> 'b -> rules

    val filter_possible : event -> entity -> role ->
                         peer -> bool * float ->
                         message option -> rules -> rules
end
```

Each different protocol will have different rules that are returned from `make_rules`.

Each rule (type `t`) must have an event it adheres to, a pre-configuration it must check, a message constructor (or not) to check against, and an action that gets executed if the rule is accepted (the function with type `unit` → `unit`).

Let's look at a few examples of this whole process.

Suppose we want to write the following rule, from Chandra-Toueg's LTS:

$$\frac{\begin{array}{l} E = \{\, \text{id} = i,\ \text{phase} = 1,\ \text{round} = r,\ \text{belief} = b,\ \ldots \} \\ m = \text{ABLF}\,\{\, i, r, 1, b \,\} \\ E' = E\,\{\, \text{phase} \leftarrow 2 \,\} \end{array}}{M \,@\, E \;>\; N_i \;--\; \text{net}\,!\,\text{m} \longrightarrow M + m \,@\, E' \;>\; N_i} \quad [MSG\_ABLF]$$

We can write it in our definition as:

Code Excerpt 4.12: MSG_ABLF as Rule type

```
let make_rules entity role peer time net in_msg out_msg out_buffer : t list = [
  ...
  (
    Lts Send,
    Conf ({ entity with f1 = Some ("phase", Nat 1) }, !role, peer),
    Some (Constructor "ABLF"),
    fun () ->
      let m =
        Make
          ( Constructor "ABLF",
            "ABFL",
            [
              entity.identity;
              get_field entity "round" |> Option.get;
              Nat 1;
              get_field entity "belief" |> Option.get;
            ] )
      in
      Net.send_all net (entity.identity |> get_id) m;
      entity.f1 <- Some ("phase", Nat 2)
  );
  ...
]
```

Another example, this time an LTS Receive:

$$\frac{\begin{array}{l} E = \{\, \mathrm{id} = i,\ \mathrm{phase} = 2,\ \mathrm{round} = r_e,\ \ldots \} \\ m = \mathrm{CBLF}\,\{\, j, b \,\} \\ m \in M \\ b \neq \perp \\ E' = E\,\{\, \mathrm{phase} \leftarrow 3,\ \mathrm{current\_leader} \leftarrow j, \\ \qquad\quad \mathrm{belief} \leftarrow b,\ \mathrm{ack} = \textbf{true} \,\} \end{array}}{M \,@\, E \ >\ N_i \ \text{-- --}\ \mathrm{net}\,?\,\mathrm{m} \longrightarrow M \,@\, E' \ >\ N_i} \quad [RCV\_CBLF\_OK]$$

Which translates to:

Code Excerpt 4.13: RCV_CBLF_OK as Rule type

```
let make_rules entity role peer time net in_msg out_msg out_buffer : t list = [
  ...
  (
    Lts Receive,
    Conf ({ entity with f1 = Some ("phase", Nat 2) }, !role, peer),
    Some (Constructor "CBLF"),
    fun () ->
      let j, b =
        match in_msg with
        | Some (Make (Constructor "CBLF", "CBLF", lst)) ->
            (List.hd lst, List.nth lst 3)
        | _ -> assert false
      in
      if b == Nat (-1) then ();
      entity.f1 <- Some ("phase", Nat 3);
      entity.f7 <- Some ("current_leader", j);
      entity.f3 <- Some ("belief", b);
      entity.f6 <- Some ("ack", Bool true)
  );
  ...
]
```

Another example, this time of a spontaneous message:

$$\frac{\begin{array}{l} E = \{\, \text{id} = i,\ \text{role} = \text{NON\_COORDINATOR},\ \text{round} = r,\ \ldots \,\} \\ n = \text{network\_size} \\ r \% n \equiv i \\ E' = E\,\{\, \text{role} \leftarrow \text{COORDINATOR} \,\} \end{array}}{M @ E \ > \ N_i \ -- \ \tau \longrightarrow M @ E' \ > \ N_i} \quad [COORD]$$

Which translates to:

Code Excerpt 4.14: COORD as Rule type

```
let make_rules entity role peer time net in_msg out_msg : t list = [
  ...
  (
    Spontaneous,
    Conf (entity, Role "NON_COORDINATOR", peer),
    None,
    fun () ->
      let r =
        get_field entity "round" |> Option.get |> fun a ->
        match a with Nat x -> x | _ -> assert false
      in
      let id = entity.identity |> get_id in
      let n = Net.size net in
      if r mod n = id then role := Role "COORDINATOR"
      else ()
  );
  ...
]
```

As we saw previously, at each point in the execution each node refreshes its list of rules with their current state. We then need to check what rules can be executed. For that, we make use of the `filter_possible` function:

Code Excerpt 4.15: `filter_possible` function

```
let filter_possible event entity role peer time msg rules =
  let alarm_switch, time = time in
  List.filter
    (fun ((e, conf, m, _) : t) ->
      (match e with
      | Alarm t when alarm_switch && t < time -> true
      | _ -> false)
```

```
    || (e = event || e = Spontaneous)
        && (match msg with
            | Some (Make (c, _, _)) -> Some c = m
            | None -> m = None)
        &&
        match conf with
        | Conf (e, r, p) when e = entity && r = role && p = peer -> true
        | _ -> false)
  rules
```

This function always considers spontaneous events, even if we only specify the event as `Lts Send`, for example. It also doesn't see the alarm as a priority event. These are the default behaviors, but, in the future, there should be options for them to be changed.
After this filter, each node can run one of its remaining rules, if any, and we can watch our running protocol!

4.2.3.6 Example with Scenario 3

Let's recreate Scenario 3 (4.2.1.4) with our implementation! The only thing that will be different will be the way we create blocks to be sent: in the last implementation, the user would input an *id* of the sender node and the message to be hashed and sent into the console. Now we can't do that, so let's use an alarm instead! Every two seconds the node with *id* equal to 1 should create a random number, hash it, and send it to all other nodes.

First, we need to choose our network topology. We will use the partially connected *Net* module because this is the one that represents our original scenario better. Wherever we use the *Net* module, we need to specify we want the file with the partially connected implementation: `module Net = Net_pcg`.

The second thing we should do is to change the `new_entity` and `new_role` functions in the *Common* module:

Code Excerpt 4.16: *Common* module for Scenario 3

```
...
let new_entity () =
  {
    identity = Id (-1);
    f1 = Some ("blockchain", Messages []);
    f2 = None;
    f3 = None;
    f4 = None;
    f5 = None;
    f6 = None;
```

```
    f7 = None;
    f8 = None;
    f9 = None;
    f10 = None;
  }

let new_role () = Role "MINER"
...
```

Next, we need to create rules:

Code Excerpt 4.17: Scenario 3 Rules

```
let make_rules entity role peer time net in_msg out_msg out_buffer : t list =
  [
    ( Lts Send,
      Conf (entity, !role, peer),
      Some (Constructor "Block"),
      fun () ->
        let m = match out_msg with Some msg -> msg | None -> assert false in
        Net.send_all net (entity.identity |> get_id) m );
    ( Lts Receive,
      Conf (entity, !role, peer),
      Some (Constructor "Block"),
      fun () ->
        let m = match in_msg with Some msg -> msg | None -> assert false in
        let content =
          match m with
          | Make (_, _, [ _; String content ]) -> content
          | _ -> assert false
        in
        let blockchain =
          get_field entity "blockchain" |> fun a ->
          match a with Some (Messages lst) -> lst | _ -> assert false
        in

        if List.mem m blockchain then ()
        else (
          Format.printf "%d: Got Block -> %s@."
            (entity.identity |> get_id)
            content;
          UnorderedSet.add out_buffer (Some m);
          entity.f1 <- Some ("blockchain", Messages (m :: blockchain))) );
    ( Alarm 2.0,
      Conf (entity, !role, peer),
```

```
      None,
      fun () ->
        time := (false, Unix.gettimeofday ());
        let m_str =
          Digestif.SHA256.feed_string ctx (Random.int 1000 |> string_of_int)
          |> Digestif.SHA256.get |> Digestif.SHA256.to_hex
        in
        let m =
          Make (Constructor "Block", "Block", [ entity.identity; String m_str ])
        in
        let blockchain =
          get_field entity "blockchain" |> fun a ->
          match a with Some (Messages lst) -> lst | _ -> assert false
        in
        entity.f1 <- Some ("blockchain", Messages (m :: blockchain));
        UnorderedSet.add out_buffer (Some m) );
    ( Spontaneous,
      Conf ({ entity with identity = Id 1 }, !role, peer),
      None,
      fun () ->
        let al, _ = !time in
        if al then () else time := (true, Unix.gettimeofday ()) );
  ]
```

The two first rules are the ones discussed in 4.2.1.4, and the two last rules are the new spontaneous and alarm rules, that make sure that the node with *id* equal to 1 can create the messages every 2 seconds.

And that's it! The full implementation is in Appendix F and the execution logs for 10 seconds in Appendix G.

4.2.3.7 Conclusions and Insight Gained

The revised modular structure of our system has brought about several key improvements and insights. In this chapter we have achieved the following:

1. **Simplified Code Generation**: The modular structure now aligns more closely with the AST of our language. The necessity for extensive code generation by the middle layer was reduced. This change streamlines the development process and minimizes potential errors associated with code generation and integration.

2. **Effective Spontaneous Messaging**: Modeled spontaneous and alarm events within our system using the event type. This enhancement allows for the simulation of real-world network behaviors, where unex-

pected events can occur at any moment, thereby increasing the robustness and flexibility of our protocols.

3. **Configurable Network Topologies**: The *Net* modules gives us the ability to define and switch between different network topologies so that the user can test his protocol in various network configurations.

4. **Dynamic Rule Evaluation**: The *Rule* modules and the `filter_possible` function enables dynamic evaluation of rules based on the current state of entities and the messages they receive or send. This was a big requirement, as it is integral to distributed systems.

Overall, the improvements introduced by the modular structure have significantly enhanced the functionality, flexibility, and reliability of Lupin. The changes addressed the issues identified in the previous section and a good foundation was built for future improvements.

4.3 Conclusions

Chapter 4 concludes with significant advancements made through the revised modular structure of the system, enhancing functionality, flexibility, and reliability. Simplified code generation aligned with the AST of our language, reducing potential errors and streamlining the development process.

In this chapter, we also proved that *Domainslib* exhibits the required nondeterminism for us to use it and its Channels, and implemented various scenarios that helped consolidate this fact.

We were able to include effective spontaneous and alarm-based events, and messages modeling real-world network behaviors. Our *Net* modules are configurable and enable users to test protocols in various configurations, and our *Rule* module includes dynamic rule evaluation based on current entity states and messages.

Chapter

5

Conclusions and Future Work

5.1 Main Conclusions

Until today there hasn't been a good way to programmatically define consensus protocols, but now there is!

The modular approach was an extremely good idea and the foundation it lays for future work is incredible, as it allows us to add more modules as needed to our functors without extra overhead, to allow the switching of modules based on requirements without modifying dependant code, and the wonders of abstraction (when done well).

Using a proven mathematical model like LTS and CCS as the base for our semantics is a huge advantage because they are well understood and work well with OCaml's functional environment. The process of creating a way to go from inference rules to actual execution was extremely satisfying and rewarding.

The main objectives for this project were to learn more about consensus protocols, category theory and to create a language that people found useful. I think it was a success on all fronts, and throughout the process, a personal passion to learn more about the topics covered was lit, and the continuation of this project will allow me to keep expanding that knowledge.

5.2 Future Work

There were a few objectives that couldn't be met in the project's time frame.

As this project is being financed by a research initiation fellowship, granted by the COPES project, it will continue for at least 3 more months. Here is what will be done over that period, ordered by importance:

1. Integration with the parser;
2. Various protocol implementations using presented definition, like the one presented at F;
3. Byzantine node simulation;
4. Checking for deadlocks, livelocks and impossible rules;
5. Let users choose node peers and create network topology, using an adjacency graph;
6. Options for changing the default behavior of the rule-checking algorithm;
7. Type-checking for Lupin code;
8. Modularity of Lupin code;
9. Code generation for MOBS.

Appendix

A

Lupin Coq code

```
Require Import CoqOfOCaml.CoqOfOCaml.
Require Import CoqOfOCaml.Settings.
Require Import List.
Require Import Coq.Logic.FinFun.
Require Import Coq.Sets.Ensembles.
Require Import Coq.Floats.Floats.
Require Import stateMonad.

Import ListNotations.
Import Notations.

Set Implicit Arguments.
Set Maximal Implicit Insertion.

Notation key := string.

(** Lupin message constructors *)
Section Constructors.
  Variable A : Type.
  Parameter A_beq_dec : A -> A -> bool.
  Definition _constructor := A -> Prop.
End Constructors.

Definition lupin_constr := _constructor key.

Definition In {A} (f: A -> Prop) (x:A) : Prop := f x.
```

```
Inductive single_constr (x : key) : lupin_constr :=
  in_single_constr: In (single_constr x) x.

Example proposal := single_constr "proposal".

(** Raw Data *)
Parameter raw: Type.

(** Identifiers *)
Notation id := nat.

(** Numbers *)
Notation number := float.

Inductive channel : Set :=
| chan : id -> channel.

Variable net : channel.

Scheme Equality for channel.

Inductive message : Set :=
| make_msg : lupin_constr -> key -> arguments -> message

with arguments : Set :=
| empty_arg : arguments
| cons_arg : value -> arguments -> arguments

with value : Set :=
| id_value : id -> value
| number_value :  number -> value
| raw_value : raw -> value
| bool_value : bool -> value
| boolOp_value : operator -> value -> value -> value
| boolPred_value: predicate -> value -> value
| messages_value: messages -> value
| channel_value : channel -> value

with operator : Set :=
| natOp : (nat -> nat -> bool) -> operator
| boolOp : (bool -> bool -> bool) -> operator
```

```
with predicate : Set :=
| boolP : (bool -> bool) -> predicate

with messages :=
| empty_msgs : messages
| cons_msgs : message -> messages -> messages.


(** Lupin roles *)
Section Roles.
  Variable B : Type.
  Parameter beq_dec_B : B -> B -> bool.
  Definition _role := B -> Prop.
End Roles.

Definition lupin_role := _role key.
Inductive single_role (x : key) : lupin_role :=
  in_single_role: In (single_role x) x.



(** Lupin entity fields *)
Section Entities.
  Variable C : Type.
  Parameter beq_dec_C : C -> C -> bool.
  Definition _field := C -> Prop.
End Entities.

Definition lupin_field := _field key.
Inductive single_field (x : key) : lupin_field :=
  in_single_field: In (single_field x) x.


Record entity : Set :=
  mkEntity
    {
      identity : value;
      f1 : value;
      f2 : value;
      f3 : value;
```

```
      f4 : value;
      f5 : value;
      f6 : value;
      f7 : value;
      f8 : value;
      f9 : value;
      f10 : value;
    }.


Variable programmed_fields : lupin_field.

Axiom field_index :
  exists g,
    Injective g  /\
      forall x,
        In programmed_fields x ->
        g x = f1 \/
          g x = f2 \/
          g x = f3 \/
          g x = f4 \/
          g x = f5 \/
          g x = f6 \/
          g x = f7 \/
          g x = f8 \/
          g x = f9 \/
          g x = f10.



Section Predicates.

  Reserved Notation "'selection".
  Reserved Notation "'msgs_pred".

  Definition entity_field := entity -> value.

  Inductive exp : Set :=
  | VarA : arguments -> nat -> exp
  | VarE : entity_field -> exp
  | Nat : nat -> exp
```

```
| Select : 'selection -> exp ->  exp
| Exist : 'msgs_pred -> exp -> exp
| Lt : exp -> exp -> exp
| Eq : exp -> exp -> exp
| And : exp -> exp -> exp
| Or : exp -> exp -> exp
| Neg : exp -> exp

where  "'selection" := (messages -> message -> bool)
                         and "'msgs_pred" := (messages -> bool).


Definition isBoolean (v : value) :=
  match v with
  | boolOp_value _ _ _
  | boolPred_value _ _
  | bool_value _ =>
    true
  | _  =>
    false
  end.

Parameter nth_arg : arguments -> nat -> t value.

Fixpoint eval (E : entity) (m : t message) (e : exp) {struct e}
  : t value :=
  match e with
  | VarA args f =>
    nth_arg args f
  | VarE f =>
    _return E.(f)
  | Select s (VarE f) =>
    match m, E.(f) with
    | Ret m1, messages_value msgs  =>
      _return (bool_value (s msgs m1))
    | _, _ =>
      raise "error: message must be passed to binary
             msg predicate"
    end
  | Select s e =>
    match (eval E m e), m with
```

```
    | Ret (messages_value msgs), Ret m1 =>
        _return (bool_value (s msgs m1))
    | _, _ =>
        raise "error: message must be passed to binary
               msg predicate"
    end
| Exist p (VarE f) =>
    match E.(f) with
    | messages_value msgs =>
        _return (bool_value (p msgs))
    | _ =>
        raise "error: message must be passed to binary
               msg predicate"
    end
| Lt e1 e2 =>
    match eval E m e1, eval E m e2 with
    | Ret v1, Ret v2 =>
        _return (boolOp_value (natOp Nat.ltb)  v1 v2)
    | _, _ =>
        raise "arguments of Nat.ltb must evaluate to value"
    end
| Eq e1 e2 =>
    match eval E m e1, eval E m e2 with
    | Ret v1, Ret v2 =>
        _return (boolOp_value (natOp Nat.eqb)  v1 v2)
    | _, _ =>
        raise "arguments of Nat.eqb must evaluate to value"
    end
| And e1 e2 =>
    match eval E m e1, eval E m e2 with
    | Ret (boolOp_value _ _ _ as v1), Ret (boolOp_value _ _ _ as v2)
    | Ret (boolOp_value _ _ _ as v1), Ret (bool_value _  as v2)
    | Ret (bool_value _ as v1), Ret (boolOp_value _ _ _  as v2)
    | Ret (bool_value _ as v1), Ret (bool_value _  as v2) =>
        _return (boolOp_value (boolOp andb) v1 v2)
    | _,  _ =>
        raise "arguments of andb must evaluate to boolean"
    end
| Or e1 e2 =>
    match eval E m e1, eval E m e2 with
    | Ret v1, Ret v2 =>
```

```
          if andb (isBoolean v1) (isBoolean v2)
          then
            _return (boolOp_value (boolOp orb) v1 v2)
          else
            raise "arguments of orb must evaluate to boolean"
      | _, _ =>
          raise "arguments of orb must evaluate to boolean"
      end
  | Neg e1 =>
      match eval E m e1  with
      | Ret v1 =>
          if isBoolean v1
          then
            _return (boolPred_value (boolP negb) v1)
          else
            raise "argument of negb must evaluate to boolean"
      | _  =>
          raise "argument of negb must evaluate to boolean"
      end
  | _ =>
      raise "expression not supported"

  end.

Fixpoint evalId (v : value) : t nat :=
  match v with
  | id_value n =>
      _return n
  | _ =>
      raise" value cannot evaluate to nat"
  end.

(** Resolution of  predicates *)
Fixpoint solver (v : value) : t bool :=
  match v with
  | boolOp_value (natOp op) v1 v2 =>
      match evalId v1, evalId v2 with
      | Ret b1, Ret b2 =>
          _return (op b1 b2)
      | _, _ =>
          raise "arguments must evaluate to nat"
```

```
      end
    | boolOp_value (boolOp op) v1 v2 =>
      match solver v1, solver v2 with
      | Ret b1, Ret b2 =>
          _return (op b1 b2)
      | _, _ =>
          raise "arguments must evaluate to boolean"
      end
    | bool_value b =>
      _return b
    | _ =>
      raise "boolean evaluation failed"
    end.

  Definition solve_expression  (E : entity) (m : t message)
    (e : exp) : t bool :=
    match eval E m e with
    | Ret v =>
      solver v
    | Err err =>
      raise err
    | _ =>
      raise "unsupported"
    end.

End Predicates.

Section LabelledSemantics.

  Inductive direction : Set :=
  | input_direction : direction
  | output_direction : direction
  | tau_direction : direction.

  (** Transition function: [pre_msg_post lconstr k dir] returns
      a tuple of expressions (e1, e2, e3, e4) where e1 is the
      pre-condition of the entity, e2 is the condition of the message,
      e3 is the post-condition of the entity,
      and e4 is the communication channel  *)
  (* Example: Algorand, constructor Proposal, output direction *)
  (* Precondition:
```

```
E = {id = i, step = 1,  round = r, period = p,  priority = t, ...} *)
  (* Msg condition:
m = Proposal{r, p, n, v, i, t} *)
  (* Postcondition:
E' = E {step <- 2}*)
  (* Channel: net *)
  (* Example: Chandra-Toueg, constructor RBC_BROADCAST,
     output direction *)
  (* Precondition :
E {id = i, phase = 4, belief = b, granted_votes = l] }
enough_votes l *)
  (* Msg condition:
m = RBC_BROADCAST {id = i, belief = b, .. } *)
  (* Postcondition: true *)
  (* Channel: net *)
  Parameter pre_msg_post:
    lupin_constr -> key -> direction -> (exp * exp * exp * exp).

  Definition constructor_precondition (E : entity) (lc : lupin_constr)
    (k : key) (d : direction) :=
    match pre_msg_post lc k d with
    | (e, _, _, _)  => solve_expression E (Err "")  e
    end.

  Definition constructor_msgcondition (E : entity) (lc : lupin_constr)
    (k : key)  (d : direction) (m : message) :=
    match pre_msg_post lc k d with
    | (_, e, _, _)  => solve_expression E (Ret m)  e
    end.

  Definition constructor_postcondition (E : entity) (lc : lupin_constr)
    (k : key) (d : direction) :=
    match pre_msg_post lc k d with
    | (_, _, e, _)  => solve_expression E (Err "")  e
    end.

  Definition constructor_channel (E : entity) (lc : lupin_constr)
    (k : key) (d : direction):=
    match pre_msg_post lc k d with
    | (_, _, _, e)  => eval E (Err "")  e
    end.
```

```
Definition find_value (v : t value) :=
  match v with
  | Ret (channel_value c) => c
  | _ => net
  end.

Definition mailbox := channel -> Ensemble message.
Notation "'{|' x '|}'" := (Singleton message x) (at level 40).
Notation "A '~+' x" := (Add message A x) (at level 10).
Notation "A '\' x" := (Subtract message A x) (at level 40).
Notation "x '' A" := (Ensembles.In message A x) (at level 40).
Notation "A '' B" := (Union message A B) (at level 40).
Definition add (M : mailbox) (c : channel) (m : message)
  : mailbox :=
  fun x =>
    if channel_beq x c then M c ~+ m else M c.
Notation "M '@' c '&' m" := (add M c m) (at level 40).

Inductive configuration : Set :=
| conf : mailbox -> peer -> configuration

with peer : Set :=
| single_peer : entity -> (lupin_role * id) -> peer
| parallel_peer: peer -> peer -> peer.

Inductive action : Set :=
| tau_action : action
| input_action : channel -> message -> action
| output_action : channel -> message -> action.

(* Used in case tau *)
Variable empty_key : key.

(** Lts semantics *)
Inductive lts : configuration -> action ->
                configuration -> Prop :=
| par_left_lts
    M F1 M' F' a F2:
  lts (conf M F1) a (conf M' F') ->
  lts (conf M (parallel_peer F1 F2)) a
```

```
    (conf M' (parallel_peer F'  F2))
| par_right_lts
    M F1 M' F' a F2:
  lts (conf M F1) a (conf M' F') ->
  lts (conf M (parallel_peer F2 F1)) a
      (conf M' (parallel_peer F2  F'))
| send_lts
    M G E R i lc m k args E' G' c R':
  G = single_peer E (R, i) ->
  E.(identity) = id_value i ->
  G' = single_peer E' (R', i) ->
  E'.(identity) = id_value i ->
  m = make_msg lc k args ->
  constructor_precondition E lc k output_direction
      = Ret true ->
  constructor_msgcondition E  lc k output_direction m
      = Ret true ->
  constructor_postcondition E' lc k output_direction
      = Ret true ->
  find_value (constructor_channel E lc k output_direction)
      = c ->
  lts (conf M G) (output_action c m) (conf (M @ c&m) G')
| receive_lts
    M G E R i lc m k args E' G' c R':
  G = single_peer E (R, i) ->
  E.(identity) = id_value i ->
  G' = single_peer E' (R', i) ->
  E'.(identity) = id_value i ->
  m = make_msg lc k args ->
  constructor_precondition E lc k input_direction
      = Ret true ->
  constructor_msgcondition E  lc k input_direction m
      = Ret true ->
  constructor_postcondition E' lc k input_direction
      = Ret true ->
  find_value (constructor_channel E lc k input_direction)
      = c ->
  m  M c ->
  lts (conf M G) (input_action c m) (conf M G')
| tau_lts
    G M E R i G' lc R' E':
```

```
    G = single_peer E (R, i) ->
    E.(identity) = id_value i ->
    G' = single_peer E' (R', i) ->
    E'.(identity) = id_value i ->
    constructor_precondition E lc empty_key tau_direction
        = Ret true ->
    constructor_postcondition E' lc empty_key tau_direction
        = Ret true ->
    lts (conf M G) tau_action (conf M G').

End LabelledSemantics.


Section TypeChecking.
  (** Type-checking messages *)
  (* messages and predicates not allowed inside messages *)

  (* TO BE CONTINUED *)
  Inductive typ : Set :=
  | id_typ : typ
  | number_typ :  typ
  | raw_typ :  typ
  | bool_typ : typ.

  Inductive type_arg (L : list typ) : arguments -> Prop :=
  | empty_arg_t :
    L = [] ->
    type_arg L empty_arg
  | id_arg_t n L' arg':
    L = id_typ :: L' ->
    type_arg L' arg' ->
    type_arg L  (cons_arg (id_value n) arg')
  | number_arg_t n L' arg':
    L = number_typ :: L' ->
    type_arg L' arg' ->
    type_arg L  (cons_arg (number_value n) arg')
  | raw_arg_t n L' arg':
    L = id_typ :: L' ->
    type_arg L' arg' ->
    type_arg L  (cons_arg (raw_value n) arg')
  | bool_arg_t n L' arg':
```

```
    L = id_typ :: L' ->
    type_arg L' arg' ->
    type_arg L  (cons_arg (bool_value n) arg').


  Inductive type_message (G : lupin_constr -> list typ)
    : message -> Prop :=
  | type_msg c L a s:
    G c = L ->
    type_arg L a ->
    type_message G (make_msg c s a).

End TypeChecking.
```

Appendix

B

Algorand, Bitcoin and Chandra-Toueg as LTS's

```
(*** LTS SEMANTICS OF ALGORAND ***)

(* Messages *)
m := c (a_1 : t1, ..., a_n : tn)
c := Proposal | SoftVote  | CertVote  | NextVote
M :=  emptyset | M + m @ a
a \in channels
t = Nat | M

(* Entities *)
E := {f1 : t1, ..., fn : tn }

(* Roles : B := BOOTSTRAP, C := COMMITTEE, P := PROPOSER *)
G := B | C | P
G_n := (G, n)
n \in Nat

(* Predicates *)
D : = \phi n M E (f1, .., fn)
\phi \in  Nat ->  M -> E -> (f1, ..., fn) -> Prop

(* Configurations *)
F := E > G_n | (F | F)
```

```
(* Actions *)
a := tau | net ? m | net ! m

LTS:  (M * F) -> a -> (M * F) : Prop :=
| PAR-L:
| BOOTSTRAP:
| MSG_PROPOSAL:
| RCV_PROPOSAL:
| MSG_SOFTVOTE
(*| RCV_SOFTVOTE_MSG_CERTVOTE *)
| RCV_SOFTVOTE
| MSG_CERTVOTE
| RCV_CERTVOTE_OK
| RCV_CERTVOTE
| MSG_NEXTVOTE_OK
| MSG_NEXTVOTE_FAIL
| RCV_NEXTVOTE

where "M @ F -- a --> M' @ F'" := LTS (M, F) a (M', F').


M @ F1 -- a --> M' @ F'
-------------------------------------------  [PAR-L]
M @ F1 | F2  -- a --> M' @ F' | F2


E = {id = i, ... }
E' = E { step <- 1}
-------------------------------------------   [BOOTSTRAP]
M @ E > B_i -- tau --> M @ E' > P_i


E = {id = i, step = 1,  round = r, period = p,  priority = t, ...}
Random.value = v (* \phi = fun : x _ _ _ -> x = v *)
m = Proposal{r, p, n, v, i, t}
E' = E {step <- 2}
------------------------------------------- [MSG_PROPOSAL]
M @ E > P_i  -- net ! m --> M + m @ E' > C_i


E = {id = i, round = r_e, period = p_e, proposals = l, ...}
```

```
m = Proposal{r, p, ...  }
m \in M
(* r < r_e \/ p < p_e *)
valid _ [m] E (round, period)
valid = fun : _ M E (f1, f2) ->
  M = [Proposal{ } as m] && (m.round < E.f1 || m.period < E.f2)
E' = E { round <- r_e + 1, proposals <- l + m  }
___________________________________________ [RCV_PROPOSAL]
M @ E > C_i  -- net ? m  --> M @ E' > C_i


E = {id = i, step = 2, num_proposers = n, clock = t, proposals = l, ... }
(* | l | = n \/ t <= 0 *)
valid _ _ E (num_proposers, proposals, clock)
valid = fun : _ _ E (f1, f2, f3) -> E.f1 = |E.f2| && E.f3 <= 0
(* v = highest_priority l *)
highest_priority v _ E  proposals
highest_priority = fun : x _ M E f =
  In Proposal{v, p} E.f && \forall Proposal {_, p'}\in E.f p' <= p
E' = E { value <- v, step <- 3}
m = Soft_vote{E.round, E.period, 3, v, i}
____________________________________________ [MSG_SOFTVOTE]
M @ E > C_i  -- net ! m --> M + m @ E' > C_i


(* cannot be done while observing output *)
(*
E = {id = i, round = r_e, period = p_e, softvotes = l, ...}
m = Soft_vote{r, p, v, ... }
m \in M
r < r_e \/ p < p_e
E' = E { round <- r_e + 1, softvotes <- l + m ] }
enough_softvotes v l
m' = CertVote{r, p, v, i}
____________________________________________ [RCV_SOFTVOTE_MSG_CERTVOTE]
M @ E > C_i  -- net ? m  --> M + m' @ E' > C_i
*)

E = {id = i, round = r_e, period = p_e, softvotes = l, ...}
m = Soft_vote{r, p, v, ... }
m \in M
```

```
r < r_e \/ p < p_e
E' = E { round <- r_e + 1, softvotes <- l + m ] }
~enough_softvotes v l
_____________________________________________ [RCV_SOFTVOTE]
M @ E > C_i  -- net ? m  --> M @ E' > C_i


E = {id = i, step = 3, softvotes = l,... }
| l | >= network_size / 2 \/ t <= 0
E' = E { value <- v, step <- 4}
m = Cert_vote{E.round, E.period, 4, v, i}
enough_softvotes v l
_____________________________________________ [MSG_CERTVOTE]
M @ E > C_i  -- net ! m  --> M + m @ E' > C_i


E = {id = i, round = r_e, period = p_e, certvotes = l, ...}
m = Cert_vote{r, p, v, ...}
m \in M
r < r_e \/ p < p_e
E' = E { certvotes <- l + m, round <- r_e + 1 }
certify_result v l
_____________________________________________ [RCV_CERTVOTE_OK]
M @ E > C_i  -- net ? m --> M @ E' > B_i


E = {id = i, round = r_e, period = p_e, certvotes = l, ...}
m = Cert_vote{r, p, v, ...}
m \in M
r < r_e \/ p < p_e
E' = E { certvotes <- l + m }
~certify_result v l
_____________________________________________ [RCV_CERTVOTE]
M @ E > C_i  -- net ? m --> M @ E' > C_i


E = {id = i, step = s, certvotes = lc, softvotes = ls ...}
s > 3
| lc | > | ls |
m = Next_vote{E.round, E.period, s + 1, v, i}
E' = E { step <- s + 1}
```

```
------------------------------------------ [MSG_NEXTVOTE_OK]
M @ E > C_i  -- net ! m --> M + m @ E' > C_i


E = {id = i, step = s, certvotes = lc, softvotes = ls ...}
s > 3
| lc | <= | ls |
m = Next_vote{E.round, E.period, s + 1, -1, i}
E' = E { step <- s + 1}
------------------------------------------ [MSG_NEXTVOTE_FAIL]
M @ E > C_i  -- net ! m  --> M + m @ E' > C_i


E = {id = i,  round = r_e, period = p_e, nextvotes = l, ...}
m = Next_vote{r, p ... }
m \in M
r < r_e \/ p < p_e
E' = E { nextvotes <- l + m }
------------------------------------------ [RCV_NEXTVOTE]
M @ E > C_i  -- net ? m --> M @ E' > C_i


(* Well-Formedness *)
WF F := NoDup (identities F).

(* Role equivalence *)
(=R) F1 F2 (_ : WF F1) (_ : WF F2) :=
 identities F1 = identities F2 /\
 \forall i \in (identities F1),
   F1(i) = _ > G_i iff  F2(i) = _ > G_i.

(* Entity equivalence *)
(=E) F1 F2 (_ : WF F1) (_ : WF F2) :=
 identities F1 = identies F2 /\
 \forall i \in (identities F1),
   (F1(i) = E1 > _  ->
             F2(i) = E_2 > _ /\ E1.step = E2.step
            /\ E1.round = E2.round) /\
   (F2(i) = E2 > _  ->
             F1(i) = E_1 > _ /\ E2.step = E1.step
            /\ E2.round = E1.round).
```

```
(* Asynchronous_Bisimulation ~  *)
M |= F1 ~ F2

iff

F1 =R  F2 and

F1 =E  F2 and

i) M @ F1 -- n ! m --> M + m @ F' implies
   M @ F2 == n ! m ==> M + m @ F'' and
   M + m |= F' ~ F''

ii) M @ F1 -- n ? m --> M @ F' implies
   M @ F2 == n ? m ==> M @ F'' or
   M @ F2 ====> M @ F'' and
   M |= F' ~ F''

iii) M @ F1 -- tau --> M @ F' implies
   M @ F2 ====> M @ F'' and
   M |= F' ~ F''


(* Soundness *)
Assume encoding [[ . ]]_DSL mapping ALG Calculus into Lambda calculus.

If [[ M ]] |= [[F1]] ~ [[F2]] then M |= F1 ~ F2.


(*** LTS SEMANTICS OF BITCOIN ***)

(* Messages *)
m := c (a_1 : t1, ..., a_n : tn)
c := Block | Inv | Rec
M :=  emptyset | M + m
t = Nat | M

(* Entities *)
E := {f1 : t1, ..., fn : tn }
```

```
(* Roles : N := MINER  *)
G := N
G_n := (G, n)
n \in Nat

(* Predicates *)
D : = \phi n M E (f1, .., fn)
\phi \in  Nat ->  M -> E -> (f1, ..., fn) -> Prop

(* Configurations *)
F := E > G_n | (F | F)

(* Actions *)
a := tau | net ? m | net ! m

LTS:  (M * F) -> a -> (M * F) : Prop :=
| PAR-L
| TAU-MINT
| MSG-MINT
| MSG_INV
| RCV_NEW_VALID_BLOCK
| RCV_VALID_BLOCK
| RCV_NEW_BLOCK
| RCV_BLOCK
| RCV_NEW_INV
| RCV_INV
| MSG_REC
| RCV_REC
| MSG_BLOCK

where "M @ F -- a --> M' @ F'" := LTS (M, F) a (M', F').


M @ F1 -- a --> M' @ F'
_____________________________________________  [PAR-L]
M @ F1 | F2  -- a --> M' @ F' | F2


E = { id = i, chain = l, minting = true ... }
~finished (minted l)
```

```
--------------------------------------------- [TAU_MINT]
M @ E > N_i -- tau  --> M @ E  > N_i


E = { id = i, chain = l, minting = true... }
finished (minted l)
minted l = b
(* \phi = fun x _ E f = x = b && minted_bool b _ E f *)
m = Block {i, b}
c = neighbours i
E' = { minting <- false }
--------------------------------------------- [MSG_MINT]
M @ E > N_i -- c ! m  --> M + m*c @ E'  > N_i


E = { id = i, new_blocks = bn + Block{_, b}, ... }
m = Inv {i, b}
c = neighbours i
E' = { new_blocks <- bn }
--------------------------------------------- [MSG_INV]
M @ E > N_i -- c ! m  --> M + m*c @ E'  > N_i


E = { id = i, received_blocks = br, chain = bc, new_blocks = bn, ... }
c = neighbours i
m = Block {j, b}
m*c \in M
m \not\in br
valid b
E' = { chain <- bc + m,  new_blocks = bn + m,
       received_blocks <- br + m, , minting <- true }
--------------------------------------------- [RCV_NEW_VALID_BLOCK]
M @ E > N_i -- c ? m  --> M @ E' > N_i


E = { id = i, received_blocks = br, chain = bc ... }
c = neighbours i
m = Block {b, ...}
m*c \in M
m \in br
valid b
```

```
E' = { chain <- bc + m,  received_blocks <- br + m,  minting <- true}
____________________________________________ [RCV_VALID_BLOCK]
M @ E > N_i -- c ?m  --> M @ E' > N_i


E = { id = i, received_blocks = br, chain = bc, new_blocks = bn, ... }
c = neighbours i
m = Block {_, b}
m*c \in M
m \not\in br
~valid b
E' = { new_blocks = bn + m, received_blocks <- br + m, }
____________________________________________ [RCV_NEW_BLOCK]
M @ E > N_i -- c ?m  --> M @ E' > N_i


E = { id = i, received_blocks = br, chain = bc, new_blocks = bn, ... }
c = neighbours i
m = Block {_, b}
m*c \in M
m \in br
~valid b
____________________________________________ [RCV_BLOCK]
M @ E > N_i -- c ?m  --> M @ E > N_i


E = {id = i,  received_blocks = br, downloading_blocks = bd, ... }
c = neighbours i
m = Inv { i, b }
m*c \in M
m \not\in br
E' { downloading_blocks <- bd + m }
____________________________________________ [RCV_NEW_INV]
M @ E > N_i -- c ? m  --> M @ E' > N_i


E = {id = i,  received_blocks = br, inv_blocks = bi, ... }
c = neighbours i
m = Inv { i, b }
m*c \in M
m \in br
```

```
____________________________________________ [RCV_INV]
M @ E > N_i -- c ? m  --> M @ E > N_i


E = { id = i, inv_blocks = bi + Inv{_, b}, ... }
m = Rec {i, b}
c = neighbours i
E' = { inv_blocks <- bi }
____________________________________________ [MSG_REC]
M @ E > N_i -- c ! m  --> M + m @ E'  > N_i

E = {id = i, downloading_blocks = bd, ... }
c = neighbours i
m = Rec { }
m*c \in M
E' = E { downloading_blocks <- bd + m }
____________________________________________ [RCV_REC]
M @ E > N_i -- c ? m  --> M @ E' > N_i


E = {id = i, downloading_blocks = bd + Rec{, b}, ... }
c = neighbours i
m = Block {i, b}
E' = E { downloading_blocks <- bd}
____________________________________________ [MSG_BLOCK]
M @ E > N_i -- c ? m  --> M + m @ E' > N_i

(*** LTS SEMANTICS OF CHANDRA-TOUEG ***)

(* Messages *)
m := c (a_1 : t1, ..., a_n : tn)
c := ABLF | CBLF | RBC_BROADCAST
M :=  emptyset | M + m
t = Nat | M

(* Entities *)
E := {f1 : t1, ..., fn : tn }

(* Roles : C := COORDINATOR, A := AGENT , 0 = END *)
G := C | A | 0
G_n := (G, n)
```

```
n \in Nat

(* Predicates *)
D : = \phi n M E (f1, .., fn)
\phi \in  Nat ->  M -> E -> (f1, ..., fn) -> Prop

(* Configurations *)
F := E > G_n | (F | F)

(* Actions *)
a := tau | net ? m | net ! m

LTS:  (M * F) -> a -> (M * F) : Prop :=
| PAR-L
| MSG-ABLF
| RCV-ABLF-OK
| RCV-ABLF-BOT
| RCV-CBLF
| TAU-CPHASE2
| MSG-BEST
| MSG-OWN
| MSG-ACK
| TAU-CPHASE4
| MSG-BROADCAST-OK
| MSG-BROADCAST-FAIL
| RCV-BROADCAST-OK
| RCV-BROADCAST-FAIL

where "M @ F -- a --> M' @ F'" := LTS (M, F) a (M', F').


M @ F1 -- a --> M' @ F'
____________________________________________  [PAR-L]
M @ F1 | F2  -- a --> M' @ F' | F2

E = { id = i, role = NON_COORDINATOR, round = r, ... }
n = network_size
r % n  i
E = E { role <- COORDINATOR }
____________________________________________  [COORD]
M @ E > N_i    > M @ E > N_i
```

```
E = {id = i, phase = 1,  round = r, belief = b, ...}
m = ABLF{i, r, 1, b}
E' = E {phase <- 2}
------------------------------------------ [MSG-ABLF]
M @ E > A_i  -- net ! m --> M + m @ E' > A_i


E = {id = i, phase = 2, round = r_e, ... }
m = CBLF{id = j, belief = b, ...   } (* b = \bot or b = best beliefs *)
m \in M
b <> \bot
E' = E { phase <- 3, current_leader <- j, belief <- b, ack = true  }
------------------------------------------ [RCV-CBLF-OK]
M @ E > A_i  -- net ? m  --> M @ E' > A_i


E = {id = i, phase = 2, round = r_e, ... }
m = CBLF{id = j, belief = \bot, ...  }
m \in M
E' = E { phase <- 3, ack = false }
------------------------------------------ [RCV-CBLF-BOT]
M @ E > A_i  -- net ? m  --> M @ E' > A_i


E = {id = i, phase = 1, votes = l ...}
m = ABLF{phase = 1, belief = b, ... }
m \in M
E' = E { votes <- l + m }
------------------------------------------ [RCV-ABLF]
M @ E > C_i  -- net ? m  --> M @ E' > C_i


E = {id = i, phase = 1 }
E' = E {phase <- 2 }
------------------------------------------ [TAU-CPHASE2]
M @ E > C_i  -- tau  --> M @ E' > C_i


E {id = i, phase = 2, votes = l ] }
```

```
enough_votes l
(* \phi = fun _ _ E f -> |E.f| > treshold *)
best l = b
(* \phi = fun x _ E f -> x = b  && best_bool x _ E f *)
E' = E { phase <- 3, belief <- b }
m = CBLF {id = i, belief = b, .. }
___________________________________________ [MSG-BEST]
M @ E > C_i  -- net ! m  --> M + m @ E' > C_i


E {id = i, phase = 2, belief = b, votes = l ] }
~enough_votes l
E' =  E { phase <- 3 }
m = CBLF {id = i, belief = b, .. }
___________________________________________ [MSG-OWN]
M @ E > C_i  -- net ! m  --> M + m @ E > C_i


E = {id = i, phase = 3, ack = v  }
m = ABLF{ack = v, ... }
E' = E {phase <- 4}
___________________________________________ [MSG-ACK]
M @ E > A_i  -- net ! m --> M + m @ E' > A_i


E = {id = i, phase = 3, grantedvotes = l ...}
m = ABLF{ack = true, ... }
m \in M
E' = E { grantedvotes <- l + m ] }
___________________________________________ [RCV-ACK]
M @ E > C_i  -- net ? m  --> M @ E' > C_i


E = {id = i, phase = 3 }
E' = E {phase <- 4 ] }
___________________________________________ [TAU-CPHASE4]
M @ E > C_i  -- tau  --> M @ E' > C_i


E {id = i, phase = 4, belief = b, granted_votes = l] }
enough_votes l
```

```
m = RBC_BROADCAST {id = i, belief = b, .. }
___________________________________________ [MSG-BROADCAST-OK]
M @ E > C_i  -- net ! m  --> M + m @ _ > 0


E {id = i, phase = 4, belief = b, granted_votes = l, round = r ] }
~enough_votes l
E' = E { round <- r + 1, phase <- 1 }
m = RBC_BROADCAST {id = i, belief = \bot, .. }
___________________________________________ [MSG-BROADCAST-FAIL]
M @ E > C_i  -- net ! m  --> M + m @ E' > C_i


E = {id = i, phase = 4, ... }
m = CBLF{belief = b, ...  } (* b = \bot or b = best beliefs *)
m \in M
b <> \bot
E' = E { decision <- b  }
___________________________________________ [RCV-BROADCAST-OK]
M @ E > A_i  -- net ? m  --> M @ E' > 0


E = {id = i, phase = 4, round = r, ... }
m = CBLF{belief = \bot, ...  }
m \in M
E' = E { round <- r + 1, phase <- 1 }
___________________________________________ [RCV-BROADCAST-FAIL]
M @ E > A_i  -- net ? m  --> M @ E' > A_i
```

Appendix

C

Domainslib's Channel Tests

Code Excerpt C.1: Unbounded Channel

```
open Domainslib

type message = Message of int | Ended of int

let chan = Chan.make_unbounded ()
let () = Random.init (Unix.time () |> int_of_float)

let sending id chan p u arr () =
  for i = p to u - 1 do
    Format.printf "Sending from id:%d value:%d@." id arr.(i);
    Chan.send chan (Message arr.(i))
  done;
  Chan.send chan (Ended id)

let receive chan c () =
  let rec h acc count () =
    match Chan.recv chan with
    | Message i ->
        Format.printf "Received: %d | Times: %d@." i acc;
        h (acc + 1) count ()
    | Ended _id ->
        let count = count + 1 in
        if count = c then () else h acc count ()
  in
  h 0 0 ()

```

```
let () =
  let size = 12 in
  let arr = Array.make size 0 |> Array.mapi (fun i _ -> i) in
  let pool = Task.setup_pool ~name:"Teste 1" ~num_domains:5 () in
  let _ =
    Task.run pool (fun () ->
        let t1 = Task.async pool (sending 1 chan 0 3 arr) in
        let t2 = Task.async pool (sending 2 chan 3 6 arr) in
        let t3 = Task.async pool (sending 3 chan 6 9 arr) in
        let t4 = Task.async pool (sending 4 chan 9 12 arr) in
        let t5 = Task.async pool (receive chan 4) in
        Task.await pool t1;
        Task.await pool t2;
        Task.await pool t3;
        Task.await pool t4;
        Task.await pool t5)
  in
  Task.teardown_pool pool;
  ()
```

Code Excerpt C.2: Bounded Channel

```
(* same as previous *)
...
let chan = Chan.make_bounded 2
...
(* same as previous *)
```

Code Excerpt C.3: *N* Readers and Writers

```
open Domainslib

type message = Message of int

let chan = Chan.make_unbounded ()
let () = Random.init (Unix.time () |> int_of_float)

module S = Saturn.Stack

let stack = S.create ()

let sending id chan p u arr () =
  for i = p to u - 1 do
    Format.printf "Sending from id:%d value:%d@." id arr.(i);
```

```
    S.push stack arr.(i);
    Chan.send chan (Message arr.(i))
  done

let receive id chan p u outarry () =
  let rec h j () =
    if j = u then ()
    else
      match Chan.recv chan with
      | Message i ->
          Format.printf "Received on id:%d value:%d@." id i;
          outarry.(j) <- i;
          h (j + 1) ()
  in
  h p ()

let () =
  let size = 12 in
  let input_arr = Array.make size 0 |> Array.mapi (fun i _ -> i) in
  Format.printf "Initial Array: ";
  let () = Array.iter (Format.printf "%d ") input_arr in
  Format.printf "@.";
  let output_arr = Array.make size 0 in
  let pool = Task.setup_pool ~name:"Teste 1" ~num_domains:8 () in
  let _ =
    Task.run pool (fun () ->
        let threads =
          Array.init 8 (fun i ->
              let f, i, inp =
                if i > 3 then (receive, i - 4, output_arr)
                else (sending, i, input_arr)
              in
              let t1 = Task.async pool (f i chan (i * 3) ((i * 3) + 3) inp) in
              t1)
        in
        Array.iter (Task.await pool) threads)
  in
  Task.teardown_pool pool;
  let rec h acc =
    match S.pop stack with None -> acc | Some e -> h (e :: acc)
  in
  let l = h [] in
  Format.printf "Lock-free Stack (just to get the order we insert into chan): ";
```

```
let () = List.iter (Format.printf "%d ") l in
Format.printf "@.";
Format.printf "Channel: ";
let () = Array.iter (Format.printf "%d ") output_arr in
Format.printf "@.";
()
```

Appendix

D

Small Blockchain with Domainslib

```
open Domainslib

type block = { creator_id : int; hash : string }
and blockchain = block list
and entity = { mutable blockchain : blockchain }

and node = {
  id : int;
  entity : entity;
  in_channel : block Chan.t;
  mutable out_channels : block Chan.t list;
}
(* in a blockchain the neighbours are not always
relevant to the identity so lets keep them separate for now *)

let ctx = Digestif.SHA256.init ()

let create_node id =
  {
    id;
    entity = { blockchain = [] };
    in_channel = Chan.make_unbounded ();
    out_channels = [];
  }

let node1 = create_node 1
let node2 = create_node 2
let node3 = create_node 3
```

```
let node4 = create_node 4
let node5 = create_node 5
let node6 = create_node 6
let node7 = create_node 7

let () =
  node1.out_channels <- [ node2.in_channel; node3.in_channel; node4.in_channel ];
  node2.out_channels <-
    [ node1.in_channel; node3.in_channel; node5.in_channel; node6.in_channel ];
  node3.out_channels <-
    [ node1.in_channel; node2.in_channel; node4.in_channel; node5.in_channel ];
  node4.out_channels <- [ node1.in_channel; node3.in_channel; node7.in_channel ];
  node5.out_channels <- [ node2.in_channel; node3.in_channel; node7.in_channel ];
  node6.out_channels <- [ node2.in_channel ];
  node7.out_channels <- [ node4.in_channel; node5.in_channel ]

let handle_node node () =
  let rec h () =
    let msg = Chan.recv node.in_channel in
    let exists = List.exists (fun a -> a = msg) node.entity.blockchain in
    if exists then h ()
    else begin
      Format.printf "Node %d: got hash %s with creator %d@." node.id msg.hash
        msg.creator_id;
      node.entity.blockchain <- msg :: node.entity.blockchain;
      List.iter (fun a -> Chan.send a msg) node.out_channels;
      h ()
    end
  in
  h ()

let rec handle_input nodes =
  let node_id = read_int () in
  let node = nodes.(node_id - 1) in
  let content = read_line () in
  let ctx = Digestif.SHA256.feed_string ctx content |> Digestif.SHA256.get in
  Chan.send node.in_channel
    { creator_id = node_id; hash = Digestif.SHA256.to_hex ctx };
  handle_input nodes

let () =
  let pool = Task.setup_pool ~num_domains:7 () in
  let nodes = [| node1; node2; node3; node4; node5; node6; node7 |] in
```

```
Task.run pool (fun () ->
    let _ = Task.async pool (handle_node node1) in
    let _ = Task.async pool (handle_node node2) in
    let _ = Task.async pool (handle_node node3) in
    let _ = Task.async pool (handle_node node4) in
    let _ = Task.async pool (handle_node node5) in
    let _ = Task.async pool (handle_node node6) in
    let _ = Task.async pool (handle_node node7) in

    let _ = Task.async pool (handle_input nodes) in
    ());
Task.teardown_pool pool
```

Appendix

E

Chandra-Toueg with Preliminary Modular Approach

```
let network_size = ref 0
let n = 5

module Net = struct
  module C = Domainslib.Chan

  type 'a t = (int, 'a C.t) Hashtbl.t

  let create () = Hashtbl.create 0
  let add bc id = Hashtbl.add bc id (C.make_unbounded ())

  let send_all bc sender_id msg =
    Hashtbl.iter (fun idn c -> if idn <> sender_id then C.send c msg) bc

  let send bc receiver_id msg =
    let c = Hashtbl.find bc receiver_id in
    C.send c msg

  let recv bc id = Hashtbl.find bc id |> C.recv
  let recv_opt bc id = Hashtbl.find_opt bc id |> Option.map (fun c -> C.recv c)
end


module type Ct_type = sig
  type role = [ `Coordinator | `Non_Coordinator ]

```

```
  type message =
    [ `ABLF of int * int * int * int
    | `CBLF of int * int * int * int
    | `RBC_Broadcast of int * int * int * int
    | `Spontaneous of int ]

  type net = message Net.t
  (* type action = Tau | Receive of net * message | Send of net * message *)

  val init_state : net -> int -> role -> unit
  val run : unit -> unit
end

module type Ct_funct = functor (Net : module type of Net) -> sig
  type state
  and role = [ `Coordinator | `Non_Coordinator ]

  type message =
    [ `Spontaneous of int
    | `ABLF of int * int * int * int
    | `CBLF of int * int * int * int
    | `RBC_Broadcast of int * int * int * int ]

  type net = message Net.t
  type action = Tau | Receive of net * message | Send of net * message

  val init_state : net -> int -> role -> unit
  val run : unit -> unit
end

module Ct : Ct_funct =
functor
  (Net : module type of Net)
  ->
  struct
    type state = {
      id : int;
      role : role;
      mutable round : int;
      mutable phase : int;
      mutable belief_value : int;
      mutable decision : int;
      mutable votes_rcvd : int;
```

```
    mutable grantedvotes_rcvd : int list;
    mutable current_leader : int;
  }

  and role = [ `Coordinator | `Non_Coordinator ]

  type message =
    [ `Spontaneous of int
    | `ABLF of int * int * int * int
    | `CBLF of int * int * int * int
    | `RBC_Broadcast of int * int * int * int ]

  type net = message Net.t
  type action = Tau | Receive of net * message | Send of net * message

  let self =
    ref
      {
        id = -1;
        role = `Non_Coordinator;
        round = 0;
        phase = 1;
        belief_value = 0;
        decision = 0;
        votes_rcvd = 0;
        grantedvotes_rcvd = [];
        current_leader = 0;
      }

  let id = ref 0
  let net : net option ref = ref None

  let init_state ne i role =
    Net.add ne i;
    net := Some ne;
    network_size := !network_size + 1;
    id := i;
    let e =
      {
        id = !id;
        role;
        round = 0;
        phase = 1;
```

```
        belief_value = 0;
        decision = 0;
        votes_rcvd = 0;
        grantedvotes_rcvd = [];
        current_leader = 1;
      }
    in
    self := e

  let majority () = !self.votes_rcvd >= (!network_size / 2) + 1

  let best () =
    let hs = Hashtbl.create 0 in
    List.iter
      (fun a ->
        match Hashtbl.find_opt hs a with
        | Some s -> Hashtbl.replace hs a (s + 1)
        | None -> Hashtbl.add hs a 1)
      !self.grantedvotes_rcvd;
    let b, _ =
      Hashtbl.fold
        (fun k v (k2, acc) -> if v > acc then (k, v) else (k2, acc))
        hs (0, 0)
    in
    List.iter
      (fun a -> a |> string_of_int |> print_endline)
      !self.grantedvotes_rcvd;
    Format.printf "%d: BEST = %d@." !self.id b;
    !self.belief_value <- b

  let send_message msg =
    let net = Option.get !net in
    match msg with
    | `ABLF _ as x ->
        Format.printf "%d: Sending ABLF@." !self.id;
        Net.send_all net !self.id x
    | `CBLF (id, _, _, _) as x ->
        Format.printf "%d: Sending CBLF@." !self.id;
        Net.send net id x
    | `RBC_Broadcast _ as x ->
        Format.printf "%d: Sending RBC@." !self.id;
        Net.send_all net !self.id x
```

```
let handle_message = function
  | `Spontaneous b ->
      Format.printf "%d: Got Spontaneous@." !self.id;
      if !self.phase = 1 then (
        !self.round <- !self.round + 1;
        send_message
          (`CBLF (!self.current_leader, !self.round, !self.phase, b));
        !self.phase <- 3)
  | `ABLF (id, _, phase, belief) when !self.role = `Non_Coordinator ->
      Format.printf "%d: Got ABLF@." !self.id;
      if !self.phase = 3 then
        if phase = 2 then (
          !self.current_leader <- id;
          send_message
            (`CBLF (!self.current_leader, !self.round, !self.phase, 1));
          !self.belief_value <- belief)
        else
          send_message
            (`CBLF (!self.current_leader, !self.round, !self.phase, 0))
  | `CBLF (_, _, phase, belief) when !self.role = `Coordinator -> (
      Format.printf "%d: Got CBLF, phase = %d@." !self.id !self.phase;
      match phase with
      | 1 ->
          !self.votes_rcvd <- !self.votes_rcvd + 1;
          !self.grantedvotes_rcvd <- belief :: !self.grantedvotes_rcvd;
          !self.phase <- 2;
          if majority () then (
            best ();
            send_message
              (`ABLF
                (!self.id, !self.round, !self.phase, !self.belief_value)))
      | 3 when belief = 1 ->
          Format.printf "%d: Got CBLF, phase = 3, belief = 1@." !self.id;
          Format.printf "%d: Votes rcvd = %d@." !self.id !self.votes_rcvd;
          !self.phase <- 4;
          !self.grantedvotes_rcvd <- belief :: !self.grantedvotes_rcvd;
          if majority () then (
            Format.printf "%d: majority, belief = %d@." !self.id
              !self.belief_value;
            !self.decision <- !self.belief_value;
            send_message
              (`RBC_Broadcast
                (!self.id, 4, !self.phase, !self.belief_value));
```

```
                !self.votes_rcvd <- 0;
                !self.grantedvotes_rcvd <- []);
              if !self.votes_rcvd = !network_size - 1 then (
                send_message
                  (`RBC_Broadcast (!self.id, !self.round, !self.phase, 0));
                !self.phase <- 1;
                !self.round <- !self.round + 1)
          | _ -> ())
      | `RBC_Broadcast (_, round, _, belief) when !self.role = `Non_Coordinator
        ->
          Format.printf "%d: Got RBC, round = %d, belief = %d@." !self.id round
            belief;
          if round = 4 && belief = !self.belief_value then (
            !self.decision <- belief;
            Format.printf "%d: Decision = %d@." !self.id !self.decision)
          else (
            !self.phase <- 1;
            !self.round <- !self.round + 1)
      | _ -> failwith (Format.sprintf "%d: Unknown message" !self.id)

    let rec run () =
      let net = Option.get !net in
      let msg = Net.recv net !id in
      handle_message msg;
      run ()
  end

module MainEntity (Net : module type of Net) (Node : Ct_funct) = struct
  let start () =
    let arr = Array.make n 0 in
    let arr = Array.map (fun _ -> (module Node (Net) : Ct_type)) arr in
    (* let net = Net.create () in *)
    let net = Net.create () in
    Array.iteri
      (fun i (module M : Ct_type) ->
        let c = if i + 1 = 1 then `Coordinator else `Non_Coordinator in
        M.init_state net (i + 1) c)
      arr;
    let pool = Domainslib.Task.setup_pool ~num_domains:5 () in
    Domainslib.Task.run pool (fun () ->
        let async = Domainslib.Task.async in
        let await = Domainslib.Task.await in
        let arr =
```

```
          Array.map (fun (module M : Ct_type) -> async pool M.run) arr
        in
        Net.send net 2 (`Spontaneous 4);
        Net.send net 3 (`Spontaneous 4);
        Net.send net 4 (`Spontaneous 1);
        Net.send net 5 (`Spontaneous 4);
        Array.iter (await pool) arr)
end

module Runner = MainEntity (Net) (Ct)

let () =
  let () = Format.printf "Starting VM!@." in
  Runner.start ()
```

Appendix

F

LTS + Modular Approach (Scenario 3 Example)

Code Excerpt F.1: *MainEntity* and *Node* Modules

```
open Buffer
open Types
module Common = Common
open Common
module Net = Net_pcg
module Rules = Rules

let network_size = ref 0
let n = 5
let net : message Net.t option ref = ref None

module type Node_t = sig
  type 'a net = 'a Net.t

  val self : entity ref
  val role : role ref
  val in_buffer : message option UnorderedSet.t
  val out_buffer : message option UnorderedSet.t
  val rules : Rules.rules option ref
  val init : int -> unit
  val set_peer : entity -> role -> int -> unit
  val make_rules : unit -> unit
  val run : unit -> unit
end
```

```

module type Node_f = functor
  (Net : module type of Net)
  (Rules : module type of Rules)
  -> sig
  type 'a net = 'a Net.t

  val self : entity ref
  val role : role ref
  val in_buffer : message option UnorderedSet.t
  val out_buffer : message option UnorderedSet.t
  val rules : Rules.rules option ref
  val init : int -> unit
  val set_peer : entity -> role -> int -> unit
  val make_rules : unit -> unit
  val run : unit -> unit
end

module Node : Node_f =
functor
  (Net : module type of Net)
  (Rules : module type of Rules)
  ->
  struct
    type 'a net = 'a Net.t

    let self = Common.new_entity () |> ref
    let role = Common.new_role () |> ref
    let peer = ref None
    let in_buffer = UnorderedSet.create ()
    let out_buffer = UnorderedSet.create ()
    let rules = ref None
    let time = ref (false, 0.)

    (** Tries to reveive a message from the Channel, adds it to the in_buffer
        if it is successful
    *)
    let try_receive net =
      match Net.recv_poll net (!self.identity |> get_id) with
      | Some msg -> UnorderedSet.add in_buffer (Some msg)
      | None -> ()

    (** Initialization function *)
```

```
  let init id =
    Net.add !!net id;
    self := { !self with identity = Id id }

  let set_peer (peer_entity : Types.entity) (peer_role : Types.role) peer_id =
    peer := Some (Single (peer_entity, (peer_role, peer_id)))

  let make_rules () =
    rules :=
      Some
        (Rules.make_rules !self role !!peer time !!net None None out_buffer)

  (** Send and receive loop
    – Tries to grab messages from the buffers and executes
      rules based on their presence or absence
  *)
  let rec send_rcv () =
    let t =
      match !time with
      | true, tr –>
          let tmp = Unix.gettimeofday () –. tr in
          Format.printf "%d: alarm time: %f@." (!self.identity |> get_id) tmp;
          (true, tmp)
      | false, _ –> !time
    in
    try_receive !!net;
    let in_msg_opt = UnorderedSet.remove_random in_buffer in
    let out_msg_opt = UnorderedSet.remove_random out_buffer in
    match (in_msg_opt, out_msg_opt) with
    | Some in_msg, Some out_msg –>
        rules :=
          Some
            (Rules.make_rules !self role !!peer time !!net in_msg out_msg
               out_buffer);
        let filtered =
          Rules.filter_possible (Lts Send) !self !role !!peer t out_msg
            !!rules
          @ Rules.filter_possible (Lts Receive) !self !role !!peer t in_msg
              !!rules
        in
        let len = List.length filtered in

        Unix.sleepf 0.5;
```

```

        if len = 0 then (
          UnorderedSet.add in_buffer in_msg;
          UnorderedSet.add out_buffer out_msg;
          send_rcv ());
        (match List.nth filtered (Random.int len) with _, _, _, f -> f ());
        send_rcv ()
    | Some in_msg, None ->
        let r =
          Rules.make_rules !self role !!peer time !!net in_msg None out_buffer
        in
        rules := Some r;
        let filtered =
          Rules.filter_possible (Lts Receive) !self !role !!peer t in_msg
            !!rules
        in
        let len = List.length filtered in

        Unix.sleepf 0.5;

        if len = 0 then (
          UnorderedSet.add in_buffer in_msg;
          send_rcv ());
        (match List.nth filtered (Random.int len) with _, _, _, f -> f ());
        send_rcv ()
    | None, Some out_msg ->
        rules :=
          Some
            (Rules.make_rules !self role !!peer time !!net None out_msg
               out_buffer);
        let filtered =
          Rules.filter_possible (Lts Send) !self !role !!peer t out_msg
            !!rules
        in
        let len = List.length filtered in

        Unix.sleepf 0.5;

        if len = 0 then (
          UnorderedSet.add in_buffer out_msg;
          send_rcv ());
        (match List.nth filtered (Random.int len) with _, _, _, f -> f ());
        send_rcv ()
```

```
      | None, None ->
          let r =
            Rules.make_rules !self role !!peer time !!net None None out_buffer
          in
          rules := Some r;
          let filtered =
            Rules.filter_possible Spontaneous !self !role !!peer t None !!rules
          in
          let len = List.length filtered in

          Unix.sleepf 0.5;

          if len = 0 then send_rcv ();
          (match List.nth filtered (Random.int len) with _, _, _, f -> f ());
          send_rcv ()

    let run () = send_rcv ()
  end

module MainEntity
    (Net : module type of Net)
    (Rules : module type of Rules)
    (Node : Node_f) =
struct
  let start () =
    let arr = Array.make n 0 in
    let arr = Array.map (fun _ -> (module Node (Net) (Rules) : Node_t)) arr in
    net := Some (Net.create n);
    let peers = Net.peers n in

    List.iter (fun (a, b) -> Format.printf "%d -> %d@." a b) peers;

    (* Node initialization *)
    Array.iteri (fun i (module M : Node_t) -> M.init i) arr;

    (* Node initialization part two, peers and rules *)
    Array.iteri
      (fun i (module M : Node_t) ->
        let _, peer_id = List.nth peers i in
        let (module Peer : Node_t) = arr.(peer_id) in
        M.set_peer !Peer.self !Peer.role peer_id;
        M.make_rules ())
      arr;
```

```
    let pool = Domainslib.Task.setup_pool ~num_domains:5 () in
    Domainslib.Task.run pool (fun () ->
        let async = Domainslib.Task.async in
        let await = Domainslib.Task.await in
        let thread_arr =
          Array.map (fun (module M : Node_t) -> async pool M.run) arr
        in
        Array.iter (await pool) thread_arr)
end

module Runner = MainEntity (Net) (Rules) (Node)

let () =
  let () = Format.printf "Starting VM!@." in
  Runner.start ()
```

Code Excerpt F.2: *UnorderedSet* Module

```
module UnorderedSet = struct
  type 'a t = ('a, unit) Hashtbl.t

  let create () = Hashtbl.create 10
  let add set elem = Hashtbl.add set elem ()
  let remove set elem = Hashtbl.remove set elem
  let mem set elem = Hashtbl.mem set elem

  let remove_random set =
    let l = Hashtbl.fold (fun k _ acc -> k :: acc) set [] in
    match l with
    | [] -> None
    | _ ->
        let index = Random.int (List.length l) in
        let elem = List.nth l index in
        Hashtbl.remove set elem;
        Some elem

  let filter f set =
    Hashtbl.fold
      (fun elem _ acc ->
        if f elem then (
          Hashtbl.add acc elem ();
          acc)
        else acc)
      (Hashtbl.create 0) set
```

```
end
```

Code Excerpt F.3: *Types*

```
type lupin_constr = Constructor of string
type role = Role of string

type message = Make of lupin_constr * string * arguments
and arguments = value list
(* and channel = *)

and value =
  | Id of int
  | Nat of int
  | Number of float
  | String of string
  | Bool of bool
  | BoolOp of operator * value * value
  | BoolPredicate of predicate * value
  | Messages of message list

and operator =
  | NatOp : (int -> int -> bool) -> operator
  | BoolOp : (bool -> bool -> bool) -> operator

and predicate = BoolP of bool * bool

type entity = {
  identity : value;
  mutable f1 : (string * value) option;
  mutable f2 : (string * value) option;
  mutable f3 : (string * value) option;
  mutable f4 : (string * value) option;
  mutable f5 : (string * value) option;
  mutable f6 : (string * value) option;
  mutable f7 : (string * value) option;
  mutable f8 : (string * value) option;
  mutable f9 : (string * value) option;
  mutable f10 : (string * value) option;
}

type configuration = Conf of entity * role * peer
and peer = Single of entity * (role * int) | Parallel of peer * peer
```

```
type action = Tau | Input of message | Output of message
type direction = Input | Output | Tau
type lts = ParLeft | ParRight | Send | Receive | Tau
type event = Lts of lts | Spontaneous | Alarm of float
```

Code Excerpt F.4: *Common* Module: Scenario 3

```
open Types

let ( !! ) s = !s |> Option.get

let new_entity () =
  {
    identity = Id (−1);
    f1 = Some ("blockchain", Messages []);
    f2 = None;
    f3 = None;
    f4 = None;
    f5 = None;
    f6 = None;
    f7 = None;
    f8 = None;
    f9 = None;
    f10 = None;
  }

let new_role () = Role "MINER"
let get_id = function Id id −> id | _ −> assert false
let get_constr = function Make (l, _, _) −> l
let get_key = function Make (_, k, _) −> k
let get_args = function Make (_, _, a) −> a
let get_peer_entity = function Single (e, _) −> e | _ −> assert false

let get_field entity k =
  let lst =
    [
      entity.f1;
      entity.f2;
      entity.f3;
      entity.f4;
      entity.f5;
      entity.f6;
      entity.f7;
      entity.f8;
```

```
      entity.f9;
      entity.f10;
    ]
  in
  List.find_opt
    (fun a ->
      match a with Some (ident, _) when ident = k -> true | _ -> false)
    lst
  |> fun a -> match a with Some (Some (_, f)) -> Some f | _ -> None
```

Code Excerpt F.5: *Net* Module: Fully Connected Undirected

```
(* file net_fg.ml *)
(* Complete graph *)
module C = Domainslib.Chan

type 'a t = (int, 'a C.t) Hashtbl.t

let create num_nodes = Hashtbl.create num_nodes
let add bc id = Hashtbl.add bc id (C.make_unbounded ())
let size bc = Hashtbl.length bc

let send_all bc sender_id msg =
  Hashtbl.iter (fun idn c -> if idn <> sender_id then C.send c msg) bc

let send bc _sender_id receiver_id msg =
  let c = Hashtbl.find bc receiver_id in
  C.send c msg

let recv bc id = Hashtbl.find bc id |> C.recv
let recv_opt bc id = Hashtbl.find_opt bc id |> Option.map (fun c -> C.recv c)

let recv_poll bc id =
  match Hashtbl.find_opt bc id with Some id -> C.recv_poll id | None -> None

(* shuffles a list *)
let shuffle lst =
  let nd = List.map (fun c -> (Random.bits (), c)) lst in
  let sond = List.sort compare nd in
  List.map snd sond

(* generate pairs of node id and its peer id *)
let peers num_nodes =
  if num_nodes < 2 then failwith "Number of nodes must be at least 2";
```

```
  let nodes = List.init num_nodes (fun i -> i) in
  let shuffled_nodes = shuffle nodes in
  let rec make_pairs = function
    | [] -> []
    | [ x ] -> [ (x, List.hd shuffled_nodes) ]
    | x :: (y :: _ as tl) -> (x, y) :: make_pairs tl
  in
  make_pairs shuffled_nodes |> List.sort (fun (a, _) (b, _) -> compare a b)
```

Code Excerpt F.6: *Net* Module: Partially Connected Directed

```
(* file net_pcg.ml *)
(* Partially complete directed graph (like the real internet) *)
module C = Domainslib.Chan

type 'a t = (int, 'a C.t) Hashtbl.t

let () = Random.init (Unix.gettimeofday () |> int_of_float)

(* shuffles a list *)
let shuffle lst =
  let l = List.map (fun c -> (Random.bits (), c)) lst |> List.sort compare in
  List.map snd l

(* generate a random adjacency list for a given number of nodes *)
let create_adj_list num_nodes =
  if num_nodes < 2 then failwith "Number of nodes must be at least 2";
  let nodes = List.init num_nodes (fun i -> i) in
  List.map
    (fun node ->
      let other_nodes = List.filter (( <> ) node) nodes in
      let shuffled_nodes = shuffle other_nodes in
      let num_peers = Random.int (num_nodes - 1) + 1 in
      (node, List.filteri (fun i _ -> i < num_peers) shuffled_nodes))
    nodes

let adj_list = ref []

let create num_nodes =
  adj_list := create_adj_list num_nodes;
  List.iter
    (fun (node, adj_nodes) ->
      Format.printf "[%d; [%s]]@." node
        (List.fold_left
```

```
              (fun acc a -> string_of_int a ^ "; " ^ acc)
              "" adj_nodes))
    !adj_list;
  Hashtbl.create num_nodes

let add bc id = Hashtbl.add bc id (C.make_unbounded ())
let size bc = Hashtbl.length bc

let send_all bc sender_id msg =
  Format.printf "%d: Sending to all@." sender_id;
  let _, connected_nodes = List.nth !adj_list sender_id in
  List.iter
    (fun idn ->
      Format.printf "%d: Sending to %d@." sender_id idn;
      try
        let c = Hashtbl.find bc idn in
        C.send c msg
      with Not_found ->
        Format.printf "%d: %d not found@." sender_id idn;
        exit 1)
    connected_nodes

let send bc sender_id receiver_id msg =
  Format.printf "%d: Sending to %d@." sender_id receiver_id;
  let _, connected_nodes = List.nth !adj_list sender_id in
  if List.mem receiver_id connected_nodes then
    let c = Hashtbl.find bc receiver_id in
    C.send c msg
  else ()

let recv bc id = Hashtbl.find bc id |> C.recv
let recv_opt bc id = Hashtbl.find_opt bc id |> Option.map (fun c -> C.recv c)

let recv_poll bc id =
  match Hashtbl.find_opt bc id with Some id -> C.recv_poll id | None -> None

(* generate the peers list for a partially connected graph *)
let peers _num_nodes =
  List.fold_left
    (fun acc (node, peers) ->
      let random = Random.int (List.length peers) in
      (node, List.nth peers random) :: acc)
    [] !adj_list
```

```
|> List.sort (fun (a, _) (b, _) -> compare a b)
```

Code Excerpt F.7: *Rule* Module: Scenario 3

```
module Rules = struct
  open Types
  open Buffer
  module Common = Common
  module Net = Net_pcg
  open Common

  let ctx = Digestif.SHA256.init ()

  type t = event * configuration * lupin_constr option * (unit -> unit)
  type rules = t list

  let make_rules entity role peer time net in_msg out_msg out_buffer : t list =
    [
      ( Lts Send,
        Conf (entity, !role, peer),
        Some (Constructor "Block"),
        fun () ->
          let m = match out_msg with Some msg -> msg | None -> assert false in
          Net.send_all net (entity.identity |> get_id) m );
      ( Lts Receive,
        Conf (entity, !role, peer),
        Some (Constructor "Block"),
        fun () ->
          let m = match in_msg with Some msg -> msg | None -> assert false in
          let content =
            match m with
            | Make (_, _, [ _; String content ]) -> content
            | _ -> assert false
          in
          let blockchain =
            get_field entity "blockchain" |> fun a ->
            match a with Some (Messages lst) -> lst | _ -> assert false
          in

          if List.mem m blockchain then ()
          else (
            Format.printf "%d: Got Block -> %s@."
              (entity.identity |> get_id)
              content;
```

```
                UnorderedSet.add out_buffer (Some m);
                entity.f1 <– Some ("blockchain", Messages (m :: blockchain))) );
        ( Alarm 2.0,
          Conf (entity, !role, peer),
          None,
          fun () –>
            time := (false, Unix.gettimeofday ());
            let m_str =
              Digestif.SHA256.feed_string ctx (Random.int 1000 |> string_of_int)
              |> Digestif.SHA256.get |> Digestif.SHA256.to_hex
            in
            let m =
              Make (Constructor "Block", "Block", [ entity.identity; String m_str ])
            in
            let blockchain =
              get_field entity "blockchain" |> fun a –>
              match a with Some (Messages lst) –> lst | _ –> assert false
            in
            entity.f1 <– Some ("blockchain", Messages (m :: blockchain));
            UnorderedSet.add out_buffer (Some m) );
        ( Spontaneous,
          Conf ({ entity with identity = Id 1 }, !role, peer),
          None,
          fun () –>
            let al, _ = !time in
            if al then () else time := (true, Unix.gettimeofday ()) );
      ]

  let filter_possible event entity role peer time msg rules =
    let alarm_switch, time = time in
    List.filter
      (fun ((e, conf, m, _) : t) –>
        (match e with
        | Alarm t when alarm_switch && t < time –> true
        | _ –> false)
        || (e = event || e = Spontaneous)
           && (match msg with
              | Some (Make (c, _, _)) –> Some c = m
              | None –> m = None)
           &&
           match conf with
           | Conf (e, r, p) when e = entity && r = role && p = peer –> true
           | _ –> false)
```

```
        rules
end
```

Appendix

G

Modular Scenario 3 Logs

```
Starting VM!
[0; [2; ]]
[1; [3; ]]
[2; [4; 1; 0; 3; ]]
[3; [1; 2; ]]
[4; [2; ]]
0 –> 2
1 –> 3
2 –> 1
3 –> 1
4 –> 2
1: alarm time: 0.000011
1: alarm time: 0.504005
1: alarm time: 1.005250
1: alarm time: 1.508955
1: alarm time: 2.009099
1: alarm time: 2.509438
1: Sending to all
1: Sending to 3
1: alarm time: 0.000000
1: alarm time: 0.503292
3: Got Block –> 6ffbae9aaff664bd4739f51a6c7883a2c3ce74e9227a6aff728d0d57ad56f234
3: Sending to all
3: Sending to 2
3: Sending to 1
1: alarm time: 1.008321
1: alarm time: 1.512859
2: Got Block –> 6ffbae9aaff664bd4739f51a6c7883a2c3ce74e9227a6aff728d0d57ad56f234
```

```
1: alarm time: 2.017883
2: Sending to all
2: Sending to 3
2: Sending to 0
2: Sending to 1
2: Sending to 4
1: alarm time: 2.519344
1: alarm time: 3.020205
0: Got Block -> 6ffbae9aaff664bd4739f51a6c7883a2c3ce74e9227a6aff728d0d57ad56f234
4: Got Block -> 6ffbae9aaff664bd4739f51a6c7883a2c3ce74e9227a6aff728d0d57ad56f234
0: Sending to all
0: Sending to 2
4: Sending to all
4: Sending to 2
1: Sending to all
1: Sending to 3
1: alarm time: 0.000000
3: Got Block -> 28dae7c8bde2f3ca608f86d0e16a214dee74c74bee011cdfdd46bc04b655bc14
1: alarm time: 0.505028
3: Sending to all
3: Sending to 2
3: Sending to 1
1: alarm time: 1.009304
1: alarm time: 1.509441
2: Got Block -> 28dae7c8bde2f3ca608f86d0e16a214dee74c74bee011cdfdd46bc04b655bc14
1: alarm time: 2.013343
2: Sending to all
2: Sending to 3
2: Sending to 0
2: Sending to 1
2: Sending to 4
0: Got Block -> 28dae7c8bde2f3ca608f86d0e16a214dee74c74bee011cdfdd46bc04b655bc14
4: Got Block -> 28dae7c8bde2f3ca608f86d0e16a214dee74c74bee011cdfdd46bc04b655bc14
4: Sending to all
4: Sending to 2
0: Sending to all
0: Sending to 2
1: alarm time: 0.000001
1: alarm time: 0.502972
1: alarm time: 1.008063
1: alarm time: 1.512309
1: alarm time: 2.013577
1: alarm time: 2.515211
```

```
1: alarm time: 3.016930
1: Sending to all
1: Sending to 3
1: alarm time: 0.000000
3: Got Block -> 48f89b630677c2cbb70e2ba05bf7a3633294e368a45bdc2c7df9d832f9e0c941
1: alarm time: 0.502755
...
```

Bibliography

[1] Yang Xiao, Ning Zhang, Wenjing Lou, and Y Thomas Hou. A survey of distributed consensus protocols for blockchain networks. *IEEE Communications Surveys & Tutorials*, 22(2):1432–1465, 2020.

[2] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.

[3] Jan A Bergstra, Alban Ponse, and Scott A Smolka. *Handbook of process algebra*. Elsevier, 2001.

[4] Robin Milner. *A calculus of communicating systems*. Springer, 1980.

[5] Pedro Fouto, Pedro Ákos Costa, Nuno Preguiça, and João Leitão. Babel: a framework for developing performant and dependable distributed protocols. In *2022 41st International Symposium on Reliable Distributed Systems (SRDS)*, pages 146–155. IEEE, 2022.

[6] Cezara Dragoi, Constantin Enea, Srinidhi Nagendra, and Mandayam Srivas. A domain specific language for testing consensus implementations. *arXiv preprint arXiv:2303.05893*, 2023.

[7] Nomadic Labs. Teztale – a Dashboard for Tezos Consensus, 2023. [Online] `https://research-development.nomadic-labs.com/introducing-teztale.html`. Last access at 11th of June 2024.

[8] Miguel Alves. MOdular Blockchain Simulator, 2022. [Online] `https://github.com/mce-alves/MOBS`. Last access at 11th of June 2024.