Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

Causal-Consistent Debugging of Distributed Erlang

Relatore:
Prof. Ivan Lanese

Presentata da:
Giovanni Fabbretti

Sessione II
Anno Accademico 2019/2020

A Domenica, per l'incondizionato amore che mi hai sempre dato.

Acknowledgements

My first and deepest thanks goes to my supervisor, prof. Ivan Lanese, for giving me the chance to work together with him, for sharing with me his incredible enthusiasm about this beautiful subject and for being the kind of supervisor that I wish everyone to have.
Then, I would like to thank my parents, Cristina and Giordano, and my sister, Gaia, because even if they do not understand what I am doing they do understand why I am doing it and that provides me a great support. Moreover, I thank them for always pushing me to chase my dreams while following my heart, I believe that this freedom that you always ensured me has made me a better person.
A mention is also necessary to all the incredible colleagues who accompanied me during this master, Simone, Federico, Pietro and Luigi, thank you for being such great companions.
Finally, the last mention goes to my roommates, and dear friends, of Via Mazzini 3, who have contributed to make this journey memorable, sharing an apartment with you during this last two years has exceeded every expectation that I had.

Abstract

The main purpose of this work is to study the reversibility in a concurrent and distributed environment. Reversing the execution of a program in such environment is not straightforward, indeed in order to undo a step performed by one of the, potentially many, processes of the system one has to make sure that also *all* of its consequences, if any, have been undone beforehand. This notion is usually referred to as *causal consistency*. Here, we study reversibility in the context of the Erlang language. In order to reverse the execution of a concurrent and distributed erlang program we define a *forward* semantics which, at each step performed by the program, stores in an external device, called history, auxiliary pieces of information that later on can be used to undo the step. In addition to the forward semantics we also develop a *backward* semantics and a *rollback* semantics, the former tells how and when we can undo a step while the latter allows us to undo several steps in an automatic manner. Lastly, we present a proof-of-concept implementation of a debugger able to reverse the execution of a concurrent and distributed program written in Erlang.

Italian summary

Lo scopo del seguente elaborato è quello di studiare le relazioni di dipendenza che intercorrono tra l'utilizzo di costrutti appartenenti ad un linguaggio concorrente e distribuito al fine di realizzare un debugger reversibile per tale linguaggio. Un debugger reversibile è un debugger che, tra le altre funzionalità, permette all'utente di interrompere l'esecuzione *in avanti* del codice e eseguirlo all'*indietro*.

Quando si parla di programmi sequenziali la nozione di reversibilità è particolarmente intuitiva, infatti sappiamo con precisione qual è l'ultima azione che il programma ha eseguito e per annullarne l'effetto è sufficiente effettuare l'azione inversa. Ad esempio, se l'azione che stiamo considerando è l'incremento di una variabile per annullarne l'effetto è sufficiente sottrarre alla variabile la stessa quantità che in precedenza vi è stata aggiunta.

La nozione si complica leggermente quando consideriamo istruzioni con perdita di memoria, come ad esempio l'assegnamento di un nuovo valore ad una variabile. In tal caso per poter disfare l'azione è necessario, prima di effettuare l'assegnamento, salvare il valore della variabile, in modo tale che poi quando si vorrà ripristinare il valore precedente sarà sufficiente consultare la *storia* del processo.

La nozione di reversibilità si complica in maniera considerevole qualora la volessimo applicare a sistemi concorrenti e distribuiti. Infatti, quando consideriamo un sistema distribuito e concorrente, per definizione, non è possibile avere una nozione di tempo o una nozione di ultima azione eseguita. In aggiunta, siccome il sistema è composto da diversi processi è necessario considerare che il comportamento di uno di questi potrebbe influire sul comportamento dei rimanenti.

Il seguente elaborato mira alla realizzazione di un debugger reversibile per programmi scritti in Erlang.

Erlang è un linguaggio funzionale, concorrente, distribuito basato su message-passing.

Per poter realizzare il debugger reversibile innanzitutto abbiamo definito la semantica formale del linguaggio, estendendo quella già presentata in [1], aggiungendo costrutti per programmazione distribuita.

Una volta definita la semantica del sistema abbiamo realizzato una semantica *forward* che ci permette di eseguire il codice in avanti e al tempo stesso salvare informazioni che poi in seguito potranno essere utilizzate per ripristinare stati precedenti della computazione. Successivamente, abbiamo definito la semantica *backward* che ci permette

di capire quando e come è possibile disfare l'effetto di una particolare azione. Infatti, siccome il sistema è composto di molti processi e ognuno di questi può influenzare il comportamento dei rimanenti, prima di poter disfare un'azione è necessario assicurarsi che tutte le sue *conseguenze*, se ce ne sono, siano state disfatte.

Infine, dato che riavvolgere l'esecuzione dell'intero sistema un passo alla volta non è efficiente dal punto di vista dell'utente abbiamo introdotto una semantica *rollback*. La semantica rollback permette all'utente di disfare un numero arbitrario di azioni in maniera automatica. Più precisamente la semantica rollback permette all'utente di specificare uno stato precedente della computazione, sotto forma di azione compiuta da uno dei processi del sistema, che desidera raggiungere, dopodiché la semantica si occupa di disfare tutte le conseguenze dell'azione fino a disfare l'azione stessa.

Una volta realizzate le tre semantiche una proof-of-concept del debugger è stata implementata. Il debugger permette di effettuare il caricamento di file sorgenti erlang, una volta che il caricamento è stato completato l'utente sceglie quale funzione fungerà da entry-point e ne specificherà gli eventuali argomenti. A tal punto, l'utente può eseguire il programma in avanti, quando si renderà conto che il comportamento del programma non è quello desiderato potrà usufruire delle varie funzionalità del debugger per cercare di identificarne la causa. Tra le funzionalità offerte possiamo identificare: ispezione dello stato di ogni processo, ispezione della storia di ogni processo, stampe di debug, ed infine eventuali stampe del programma (disponibili nella console in cui il debugger è stato lanciato).

Contents

Chapter 1

Introduction

Writing code is a complicated art. To write a good quality software a developer needs to have a deep understanding of different subjects (logic, mathematics, language theory, programming, to mention a few) and possibly a long experience in the field. Millions and millions of people know how to code nowadays and yet, not even a single one of them can write completely bug-free code; actually, it is quite the contrary. Additionally, errors are even more frequent and hard to discover when it comes to distributed and concurrent programming; this is mainly due to the nature of this kind of task. In fact, while in sequential programming we have only one thread executing instructions in concurrent and distributed programming we have several thread working at the same time. Moreover, these thread are cooperating together, in some scenarios sharing resources, in other exchanging messages, therefore the programmer must ensure that not only every single thread is performing correctly its work but also that once that they are working all together they are still behaving as expected.

Many programming languages address some of these problems through the paradigm they implement: let us consider the case of a race condition. A race condition occurs when two or more threads can access shared data and they try to change it at the same time. This problem does not occur in a functional programming language, in fact when using the functional paradigm it is not possible to change the value of a variable after that this one has been bounded to a value the first time, therefore it is impossible to face the problem. Unfortunately, this is not enough to write bug-free code.

Erlang [2], a functional and concurrent programming language based on asynchronous message passing, is a perfect example of such a language. Erlang implements the *actor model* [3], a mathematical model that treats actors as the universal primitive of concurrent programming. An actor is a process that can: communicate with other actors through messages, modificate his own state, create more actors. Thanks to the actor model many problems, like shared resources, are addressed by the language, yet it is still possible to introduce errors in the code. For instance, since the semantics of an erlang program is mostly based on how the actors involved communicate with each other if

there is an error in the implementation of the protocol that they are supposed to obey, then the program will not behave as expected.

Therefore, since writing software is not an error-free activity, we need tools to ensure that the code we write, at least, contains the smallest number of errors. Many of these tools already exist; we have IDEs to help writing, git to collaborate with colleagues, compilers to translate programs, debuggers and the list keeps going on.

In particular, a debugger is a tool meant to help users to discover errors in a given program by running it in a controlled environment that permits the programmer to track the operations in progress and monitor changes in the computer resources. This work aims to provide the community with a new causal-consistent debugger for a subset of the Erlang language, to help them during the process of seeking the bug in a distributed system written in Erlang. The rest of the chapter will give a brief overview to the reader of the basics concepts on which the whole project heavily relies. The code of the proof-of-concept debugger is publicly available at: https://github.com/gfabbretti8/cauder

1.1 Concurrent and distributed programming

Concurrent programming is a technique in which two or more processes start, run in an interleaved fashion through context switching and complete in an overlapping time period by managing access to shared resources, e.g., on a single core of CPU. Context switching is the procedure of saving the states of the registers and of the program counter of a process, so that its computation can be resumed at a later point, this allows to run several processes on a single core.

In [4], published in 2005, it is well underlined that the leading companies manufacturing CPUs have reached the limits of a single core CPU; however, the demand for more computational power was and will always be ubiquitous. To answer this omnipresent demand of computational power, since it has become almost impossible to improve the performances of a single CPU, the trend moved towards multiple CPUs.

Nowadays every computer relies on several cores, even the domestic ones, each of these cores can run several threads, hence to fully exploit the computing power of a calculator, we have to use those cores together in a concurrent fashion. Thanks to these multiple cores that we have in our machine, it is possible to have many applications running simultaneously, allowing us to perform seamlessly even tasks that require heavy computations.

Unfortunately, concurrent programming comes with inherent problems due to its nature; since two or more processes share the same memory and the same resources, we have to deal with situations that are not present in sequential programming. The main problem is caused by the fact that two or more processes share the same resources, to have a better understanding of why this might cause problems let us analyze what happens in the following example.

```
p1 {                                p2 {
...                                 ...
if(x == 5){                         x = 4;
   x = x*10;                        ...
 }                                  }
 ...
}
```

Figure 1.1: Two concurrent processes

Example 1.1. The two processes in figure 1.1 run in a concurrent fashion, this means that they can execute instructions at the same time, potentially involving the same resource. Since the two processes are concurrent the following is possible: p1 could perform the check on the variable x, checking that its value is 5, then proceed to multiply it by 10; the problem is that we have no warranty that the actual value of x is 5 when the multiplication will be performed. Indeed, after the check of the if clause p2 could have assigned the value 4 to x, so p1's behaviour is no longer what we were initially expecting, this situation is called *race condition*.

To address this problem usually a *lock* it used. A lock is a synchronisation mechanism which ensures that if a process is able to get the lock of a variable then it is also the only one allowed to perform operations on it. The introduction of the lock mechanism brings up new scenarios:

- Deadlock: each process is waiting for another process to unlock a resource, without which it cannot proceed with the computation.
- Starvation: a group of processes are locking and unlocking a certain resource and one or more processes cannot get the lock of the resource.

Some programming languages tackled these problems at the root, solving some of them by construction; this is particularly the case for Erlang [2]. Erlang is a functional concurrent programming language that uses the actor model; it has been designed for systems with one, or more, of the following traits:

- Soft-real time
- Distributed
- Fault tolerant
- Hot swapping
- Highly available, non-stop application

Hot swapping is a particularly interesting feature: by *hot-swap* it is intended the ability of a system to update one, or more, of its modules without halting the system. Over a long time a system might, for instance, face the need to change the exposed APIs, this might be due to an update of the data handled or to adapt the behaviour to a new protocol. Restarting the system, to make an update effective, is a time-consuming process and during that process the system it is not available to its users. Thanks to the hot swapping Erlang offers the possibility to update modules without interrupting the system. In order to do it Erlang use the *code server*. The code server is basically a VM process that stores in memory two versions of the same module, one is the newest version of the module loaded and the other one is the one currently used (they might or might not coincide). Whenever a function call, prefixed by the module, is performed the virtual machine checks if a newer version of the module is available, if that is the case the latest version is then used, if the current version is already the latest the function it is normally called. If the function call is performed omitting the module then the current version of the module it is used, even if a newer version is available.

Erlang is also called the language of the nine zeros [5]; this is because the project AXD301, where Erlang has been employed to develop the system, over a period of 20 years, has been available 99.9999999% of the time. Erlang owes its success to some of its characteristics, like: the actor model, the 'let it fail policy', and the functional paradigm.

The actor model is a paradigm where every process is considered an *actor*, every actor can communicate with other actors through asynchronous messages, and every actor has a queue of received messages. In this model the only way that an actor has to affect the behaviour of another actor is through a message, indeed it is impossible for the actor A to change directly the state of the actor B. Thanks to this model some problems, like race conditions, are addressed directly from the language, in fact since every actor's state is private it is impossible to have two actors trying to manipulate the same resource.
The actor paradigm is also excellent for distributed programming since every process of the system can be seen as an actor, with the only difference, compared to concurrent programming, that two actors are possibly running on different machines.

Although thanks to such an approach it is possible to avoid some issues, it is still possible to write buggy programs; some of these bugs can be considered logical errors of the program.

A logical error can be defined as a misbehaviour of the program that causes it to operate in an undesired fashion, but it is not detected from a static analysis. When it comes to a concurrent, distributed system it is easy to introduce bugs in the code and potentially hard to spot them because some of these faulty behaviours might only show up in some interleaving of the actors. For instance, the faulty behaviour might depend on the different speed of execution of the processes, which means that it is particularly complicated to spot the error and then solve it, or simply reproduce such circumstance during the debugging phase.

Let us consider once more Example 1.1: we already established that the program

is not safe due to the fact that the second process could modify the value of variable x while the first process is performing operations on it. The fact that the program contains this bug does not imply that the faulty behaviour will show up in every execution of the program. As a matter of fact, the first process might be significantly faster than the second one, for instance 1000x times faster, so the likelihood that the second process will change the value of x in that critical section is very low. Consequently, since the faulty behaviour might arise only in some interleaving, potentially very few of them, and it is a challenging task to find the source of the problem.

1.2 Reversible computation

A program can be considered as a list of instructions that a computer performs. Usually, instructions are performed in an incremental way, this means that $istr_i$ is executed before $istr_{i+1}$, this kind of computation is also referred to as *forward computation*.

Although forward computation is the standard kind of computation, it is not the only one; in fact, as a dual of the forward computation, also *backward* computation exists. Backward computation allows one to execute instructions in a reverse way, by undoing the effects of the forward computation previously performed.

Given a generic program, it is not guaranteed that it is possible to execute it in a backward manner. In fact, during its execution, a generic program, unless reversible or without loss of information, erases intermediate data while computing the final output.

Let us consider the instruction `x = x + 1` and the instruction `x = 42`. The first one is lossless, indeed whichever the previous value of `x` was, it can be retrieved by applying the inverse function `x = x - 1`. The second one is with loss of information: after performing the assignment to `x`, we will lose once and for all the previous value of `x`. Given the latter situation, it is impossible to execute the program in a backward fashion because we have no way to restore the previous value of `x`.

We can now distinguish between programming languages that allow instructions with information loss and languages that only allow instructions without information loss; among the ones belonging to the second category, there is Janus, firstly described in [6].

Janus only allows instructions that have a mathematical inverse function, this way it is straightforward to execute a program in both a forward and a backward manner.

However, every mainstream programming language allows instructions with information loss. The central challenge is that given the state of a program, if we want to reach the previous state and the step that must be undone belongs to the ones that erase information, it is impossible to know which one is the predecessor state and consequentially how to restore it.

Landauer first in [7] and then Bennet in [8] tackled this problem by adding information history to the computation of each state, making it possible to recover to previous states regardless of the instruction performed. This simple idea is compelling; let us consider

once more the example `x = 42`. If we are also provided with a history of the program, including the previous values that `x` has had, reversing the instruction becomes a simple task: we can consult the history to know which one was the previous value before the assignment and then restore the previous state. A considerable drawback of this approach is the amount of memory necessary to store all the intermediate information necessary to the program's execution.

A naive approach to save the history of a given process is the following: for each step that the process performs, we save its environment and a map to keep track of every variable's actual value. Such an approach is computationally expensive because, at each step, a new copy of the environment and the map is saved regardless of the fact that any of the values in both objects have been changed. For instance, if the action performed is a print we will not introduce new variables or change the value of the existing ones, so it is not necessary to save again a copy of both objects, which are identical to the previous ones. Although more sophisticated ways of keeping the history exist, in this work we will use the naive approach since reducing the memory used to store information is orthogonal to the goals of this project.

Reversible computation could be applied to improve how some tasks are solved, to mention a few of them: error recovery, exploring different computations, state exploration, and finally, debugging.

1.3 Reversible debugger

Debugging is the activity to analyze a program, find a *bug* (i.e. a mistake) and solve it. Sometimes bugs are evident, they are fatal errors and it is easy to find and fix them, sometimes they are trickier, sometimes the faulty behaviour only arises under certain circumstances and it is painful to reproduce it and then find a solution.

A debugger is a program meant to help developers to analyze providing by providing pieces of information that are generally not available. Usually, a debugger runs a program under precise environment conditions in a controlled manner, keeping track of every change in the memory to help the programmer spot the faulty behaviour.

According to [9] more than 70% of software budget is spent on debugging and over 75% of software professionals do program maintenance of some sort. From these numbers, it is evident that debugging is an activity that cannot be underestimated, and it is crucial to have efficient tools to simplify the task.

A typical feature offered by a debugger is the possibility of inserting *breakpoints* in the code. Often the programmer suspects in what part of the code the bug is located, so (s)he can insert a breakpoint in the code, then the computation can be started through the debugger. When the computation reaches the instruction marked by the breakpoint the debugger stops the computation and shows the environment state to the user. This way the user can check the values of the variables, then (s)he can choose to resume the

computation, eventually to another breakpoint, or (s)he can proceed the computation step by step to have a better understanding of the program's behaviour. This latter task is particularly tedious: the user has to press the button to execute the forward step, check that everything is ok and then continue, if (s)he goes too far in the execution and misses the faulty behaviour, the process of debugging needs to be restarted.

A reversible debugger is a tool that, among other things, allows the user to execute the code in a backward manner. If the debugger can reverse the code's execution, then it is easy for the programmer to go back, spot the error, and fix it without restarting the process from scratch.

On one hand, this notion is particularly natural when it comes to sequential code; the instructions are executed in an incremental way, so to reverse the execution it is just necessary to execute them backward, using, for instance, the technique described in Section 1.2.

On the other hand, this notion is less natural when it comes to concurrent programming; in fact, in concurrent programming, there are several processes running simultaneously, potentially interacting with each other, so it is impossible to have a notion of time and a notion of last instruction executed. What we would like to do is to undo the action(s) of a particular process, possibly the one which we suspect has the faulty behaviour, but in order to undo the actions of a process we need to undo also *all* of its *consequences*.

This notion is called *causal consistency* [10]. If we do not undo the consequences of the action, we will reach a state of the system which is not reachable with a forward computation, entering thus a state of inconsistency. For instance, let us consider the scenario where we have a system composed by two processes, namely p1 and p2, where p1 has spawned p2. If we desire to undo the spawn of p2 before proceeding to actually undo the spawn we first need to undo all the actions performed by p2, otherwise we would enter a state of the system formerly not reachable.

1.4 CauDEr

A causal-consistent debugger is a tool which allows us to debug a concurrent program, ensuring causal-consistency. As discussed in Section 1.1, it is far from trivial to spot a faulty behaviour in a concurrent system, so it is beneficial to have tools that help us. In [1], CauDEr is introduced, CauDEr is a causal-consistent debugger for a functional and concurrent subset of the Core Erlang language [11].

The program works as follow: the user loads an Erlang source file, the Erlang source file is translated into Core Erlang and then the resulting code is shown in the Code window, if no error has been found by the compiler. Then, the user specifies which function is the entry-point, and, eventually, the function's arguments and then, after pressing the start button, the system is booted. After booting the system, the user can

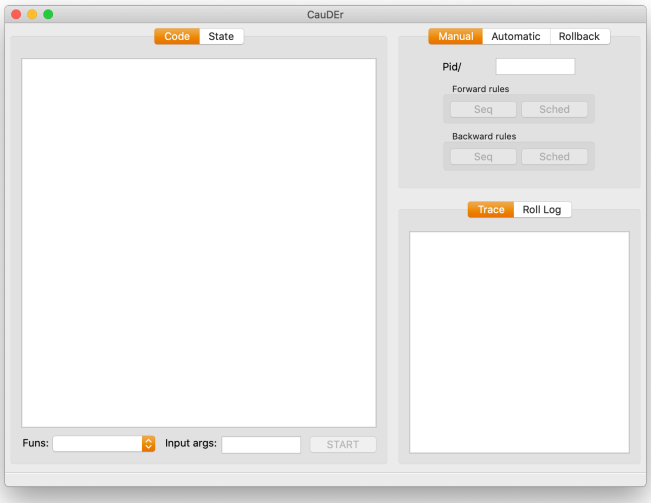


Figure 1.2: Screenshot of the application

execute the program in a forward manner automatically, specifying how many forward steps need to be performed, or, alternatively, the user can select a process and perform forward steps of that particular process.

Moreover, the user can select a process and perform a backward computation on it, given that all the consequences of the action that the user desire to undo are all undone.

Additionally, the user can *rollback* the process to a particular point where an observable action has been performed, with the guarantee that causal-consistency has been ensured.

A tool that allows its user to focus only on the desired process, ignoring the others, is powerful, it reduces the amount of information that the user perceives in a single moment, reducing the cognitive load and so allowing him to focus solely on what he needs. Let us consider a debugger that keeps a trace of the whole system and then allows us to move it back and forward; such a tool would not allow us to focus on a particular process but would force us to rollback and replay the whole system, even processes that are not linked to the one we are interested in. Such an approach would overwhelm the user with information which are not useful, increasing thus the probabilities that (s)he will miss the bug causing the misbehaviour.

In [1], a semantics for the chosen subset of Erlang is given, followed by a reversible one which can go both forward and backward, in a non-deterministic fashion. Then, the control on the previous semantics is added through the rollback operator and finally an implementation of CauDEr is presented.

1.5 Distributed CauDEr

A distributed system is a system whose components are located on different networked computers, communicating and coordinating their actions by passing messages to one another. A distributed system can be seen as an edge case of a concurrent system where different processes could run in different machines.

Nowadays we are surrounded by distributed systems, and we use them on a daily basis; telephone networks, cellular networks, peer-to-peer applications, distributed databases, network file systems are all examples of distributed systems. Erlang is a language particularly suited to do concurrent and distributed programming, and since distributed programming shares some common problems with concurrent programming, it is crucial to have tools to help the programmers developing such systems.

As we have already mentioned in Section 1.4, CauDEr supports a functional, concurrent subset of the Core Erlang language; the aim of this work is to extend that subset to a functional, concurrent distributed subset of the Core Erlang language, by extending the, already supported, subset with constructs that allow to run and coordinate processes running different physical machines. In other words the goal is to realise a new version of CauDEr that is also able to debug distributed programs in a causal-consistent manner. Before listing the set of commands added we informally introduce the notion of *node* and *spawn* because the knowledge of these terms is essential to understand the work done.

A node can be seen as a pool of processes, nodes are virtual and two different nodes could be located in the same physical machine; nonetheless, the convention is that each machine connected to the network contains just one node.

Spawn is the term we will use to indicate that an actor has started another actor; a spawn can be remote, i.e., the actor is located in a different node, or local, i.e., the spawned actor is located on the same node of his parent.

After introducing these fundamental notions we can proceed with the list of constructs:

- `node/0`: returns on which node the actor is located
- `nodes/0` returns the list of all visible nodes the actor is connected to
- `start/2`: allows an actor to start a new node, given that a node with the same name does not belong already to the network
- `spawn/4`: allows an actor to perform a remote spawn

Obviously, there are numerous more constructs designed to perform distributed programming, but the ones added are sufficient to perform every kind of distributed computation. Following the approach in [1] the various semantics, including the new constructs, have

been defined, paying attention to preserve causal-consistency, and finally the debugger has been expanded adding such distributed constructs.

Chapter 2

Background

In the following of this chapter, we will discuss and analyse the work done in [1], where the first version of CauDEr, a reversible debugger for (a subset) of Erlang, has been presented. The paper first presents the syntax of the language, then the reversible and the rollback semantics, and finally several properties of the semantics are proven. Presenting the paper is needful because that work constitutes the basis of the distributed version of CauDEr.

2.1 Language: syntax

Erlang is a functional, message-passing, concurrent, distributed programming language. Core Erlang is an intermediate representation of an Erlang program during the process of its compilation. In a nutshell, Core Erlang is a much simpler version of Erlang, indeed from a language perspective many of the constructs that Erlang offers are nothing but syntactic sugar from a compiler perspective, therefore working with Core Erlang is much simpler. In this section we will provide the syntax and the semantic of the subset of Core Erlang supported by CauDEr.

In Figure 2.1, one can find the syntax of the language. Here, a module is a sequence of function definitions, a function can be identified by *f/n*, i.e., an *atom*, representing the name, followed by an integer representing the number of arguments required. Every function has associated a body represented by an *expression*, which might include variables, literals, function calls, built-in calls, lists, tuples, let bindings, receive functions, spawn functions, self functions and "!".

In the let binding, Var, as is standard practice, always has to be considered a fresh variable. In this syntax, expressions, patterns and values carry different meanings. A pattern is composed of variables, literals, lists and tuples, while values are composed by literals, lists and tuples without variables (i.e., they are *ground*). Expressions are denoted by $e, e', e_1e_2, e_3, ...$, patterns by $pat, pat', pat_1, pat_2, pat_3, ...$ and values by $v, v', v_1, v_2, v_3, ...$,

$$
\begin{array}{rcl}
module & ::= & fun_1 \ \ldots \ fun_n \\
fun & ::= & fname = \mathsf{fun}\ (X_1, \ldots, X_n) \rightarrow expr \\
fname & ::= & Atom/Integer \\
lit & ::= & Atom \mid Integer \mid Float \mid [\,] \\
expr & ::= & Var \mid lit \mid fname \mid [expr_1|expr_2] \mid \{expr_1, \ldots, expr_n\} \\
 & \mid & \mathsf{call}\ expr\ (expr_1, \ldots, expr_n) \mid \mathsf{apply}\ expr\ (expr_1, \ldots, expr_n) \\
 & \mid & \mathsf{case}\ expr\ \mathsf{of}\ clause_1; \ldots; clause_m\ \mathsf{end} \\
 & \mid & \mathsf{let}\ Var = expr_1\ \mathsf{in}\ expr_2 \mid \mathsf{receive}\ clause_1; \ldots; clause_n\ \mathsf{end} \\
 & \mid & \mathsf{spawn}(expr, [expr_1, \ldots, expr_n]) \mid expr_1 \,!\, expr_2 \mid \mathsf{self}() \\
clause & ::= & pat\ \mathsf{when}\ expr_1 \rightarrow expr_2 \\
pat & ::= & Var \mid lit \mid [pat_1|pat_2] \mid \{pat_1, \ldots, pat_n\}
\end{array}
$$

Figure 2.1: Language syntax rules

variables start with a capital letter and atoms are represented by lower-case letters, or are enclosed in quotes if they start with a capital letter or they contain special characters.

To bind variables with expressions a *substitution* θ is used, where θ is a mapping between the variable and the expression. A substitution is usually a set of associations, like $\{X_1 \mapsto v_1, ..., X_n \mapsto v_n\}$, meaning that variable X_i has to be replaced by v_i. The identity substitution is represented by id. Two substitutions can be composed by juxtaposition, i.e., $\theta\theta'$ indicates $\theta'(\theta(X))$ for all $X \in Var$. A substitution can be updated, and it is denoted by $\theta[X_1 \mapsto v_1, ..., X_n \mapsto v_n]$, the resulting substitution θ' has the following semantic: $\theta(X) = v_i$ if $X = X_i$ where $i \in \{1, ..., n\}$, $\theta'(X) = \theta(X)$ otherwise.

In a case expression, $\mathsf{case}\ e\ \mathsf{of}\ pat_1\ \mathsf{when}\ e_1 \rightarrow e'_1; ...;\ pat_n\ \mathsf{when}\ e_n \rightarrow e'_n\ \mathsf{end}$, first e is evaluated to a value v, then a pattern that matches v is searched among $pat_1, ..., pat_n$. A pattern that matches v is always found, indeed, if the user has not provided the case with a *catch-all* clause the Erlang compiler will, and that clause will be the one responsible for throwing the error *badmatch*. Afterwards, when a pattern that matches v is found the clause $\mathsf{when}\ e_i$ is evaluated, if e_i evaluates to true then e'_i is evaluated, if e_i evaluates to false then a new pattern that matches v will be searched. Due to a decision taken while developing Erlang, the e_i expression can only contain calls to built-in functions (usually arithmetical and relational operators).

Now let us describe the concurrent features of the language, first of all, we introduce the notion of *system*. A system is as a pool of processes, where each process is uniquely identified by its *pid*, the only way that a process has to interact with the other processes is through the exchange of messages. The send of a message is asynchronous, this means that after sending a message the process proceeds with its computation, in other words the send of a message is a non-blocking operation. The syntax used to send a message is $e_1 \,!\, e_2$, where e_1 evaluates to a pid and e_2 evaluates to v_2, which represents the message.

Finally, the whole expression evaluates to v_2 and as a side effect the message will be stored, eventually, to the queue of messages of the receiver. In this work we assume that the expression e_1 always evaluates to a pid which has been generated by means of a spawn or to the pid of the first process of the system. As opposed to the send, the receive function is a blocking operation, when the receive is performed the process' queue of messages is traversed, searching for a message that matches a branch of the receive statement, if no message matches then the computation of that process is suspended until the reception of a message that will match. Moreover, when a receive is performed and matches a message successfully, as a side effect, this last one will be removed from the queue of messages.

As already discussed, each process is uniquely identified by a pid, and self is the function that returns the pid of the current process. New processes can be added to the system by means of a spawn, indeed, through spawn$(f/n, [v_1, ..., v_n])$ a process can spawn another process and this last one will start its computation by applying f/n to $[v_1, ..., v_n]$. The evaluation of spawn$(f/n, [v_1, ..., v_n])$ will also return the pid of the newly spawned process.

Example 2.1. Let us consider the program in Fig. 2.2. With the symbol "_" we indicate an *anonymous variable*, i.e., a variable whose value we are not interested in. The computation starts with "apply $main/0$ ()". This creates a process, say m, then m proceeds its own computation by spawning two processes, say p_1 and p_2, and successively saves in the variable S its pid. Then, it proceeds by sending a message to both p_1 and p_2 and terminates its computation. In the meantime, both p_1 and p_2 are waiting to receive a message, once this is received they reply back with a simple message containing the atom *pong*. In this language, there is no guarantee about the order of the messages, (a), (b), (c), and (d) of Fig. 2.9 are all admissible interleaving of the messages (note that the four interleavings shown in the figure are not all the possible ones).

2.2 The language semantics

In this section we will formalise the semantics of the language, following the same approach used in [1], however before doing so we formally introduce some fundamental notions.

Definition 2.1 (Process)**.** A process is a tuple $\langle p, (\theta, e), q\rangle$, where p represents the process' pid, θ is the environment (i.e., a substitution), e is the next expression that needs to be evaluated and q is the process' *mailbox*, a FIFO queue of messages that have been sent to the process. The following operations are permitted on the mailbox. Given a message v and a local mailbox q, we identify $v : q$ as a new local mailbox with the message v on top. On the other hand, by means of $q \backslash\backslash v$ we indicate the queue resulting by removing

$$
\begin{aligned}
\text{main}/0 = \mathsf{fun}\ () \rightarrow\ & \mathsf{let}\ P1 = \mathsf{spawn}(\text{ping}/0, [\,]) \\
& \mathsf{in\ let}\ P2 = \mathsf{spawn}(\text{ping}/0, [\,]) \\
& \mathsf{in\ let}\ S = \mathsf{self}() \\
& \mathsf{in\ let}\ \_ = P1\ !\ \{S, \text{ping}\}, \\
& \mathsf{in}\ P2\ !\ \{S, \text{ping}\} \\
\text{ping}/0 = \mathsf{fun}\ () \rightarrow\ & \mathsf{receive} \\
& \quad \{P, \text{ping}\} \rightarrow P\ !\ \text{pong} \\
& \mathsf{end}
\end{aligned}
$$

Figure 2.2: A simple multiple ping-pong program

v from q, it is important to notice that v not necessarily is the oldest message in the queue.

Definition 2.2 (System)**.** A system is denoted by $\Gamma; \Pi$, where Γ, the global mailbox, is a multiset of the form *(target_process_pid,message)*, and Π is a pool of running processes, denoted by an expression of the form

$$\langle p_1, (\theta_1, e_1), q_1 \rangle \mid ... \mid \langle p_n, (\theta_n, e_n), q_n \rangle$$

where "|" represents a commutative and associative operator. Given a global mailbox Γ we denote $\Gamma' = \Gamma \cup \{(p, v)\}$ as the new global mailbox which also stores the pair (p, v). Often, in the rest of this work we will denote a system with the expression

$$\Gamma; \langle p, (\theta, e), q \rangle \mid \Pi$$

to indicate the process p, which is an arbitrary process of the pool (this is possible thanks to the associativity and commutativity of the operator "|").

The presence of Γ is needed to guarantee that every possible interleaving of the messages is admissible, in fact, without the presence of the mailbox when a process performs a send the only place where the message could be stored would be the queue of the receiver. Such an approach would imply that the order of the messages is always guaranteed, which is not always true in real systems (e.g., a message can be lost in the network or simply delayed). Thanks to the global mailbox one can easily simulate every possible interleaving of an asynchronous model.

Example 2.2. In this example we will show how it is possible to obtain different interleaving by means of the global mailbox. Let us consider two processes, p_1 and p_2, now p_1 performs a send to p_2, then the message, say v_1, is stored in Γ, then p_1 sends another message to p_2, say v_2, and v_2 is also stored in Γ. Now Γ contains both messages and v_2 can be delivered to p_2 before v_1, in other words, Γ allows us to simulate the unpredictable behaviour of a network.

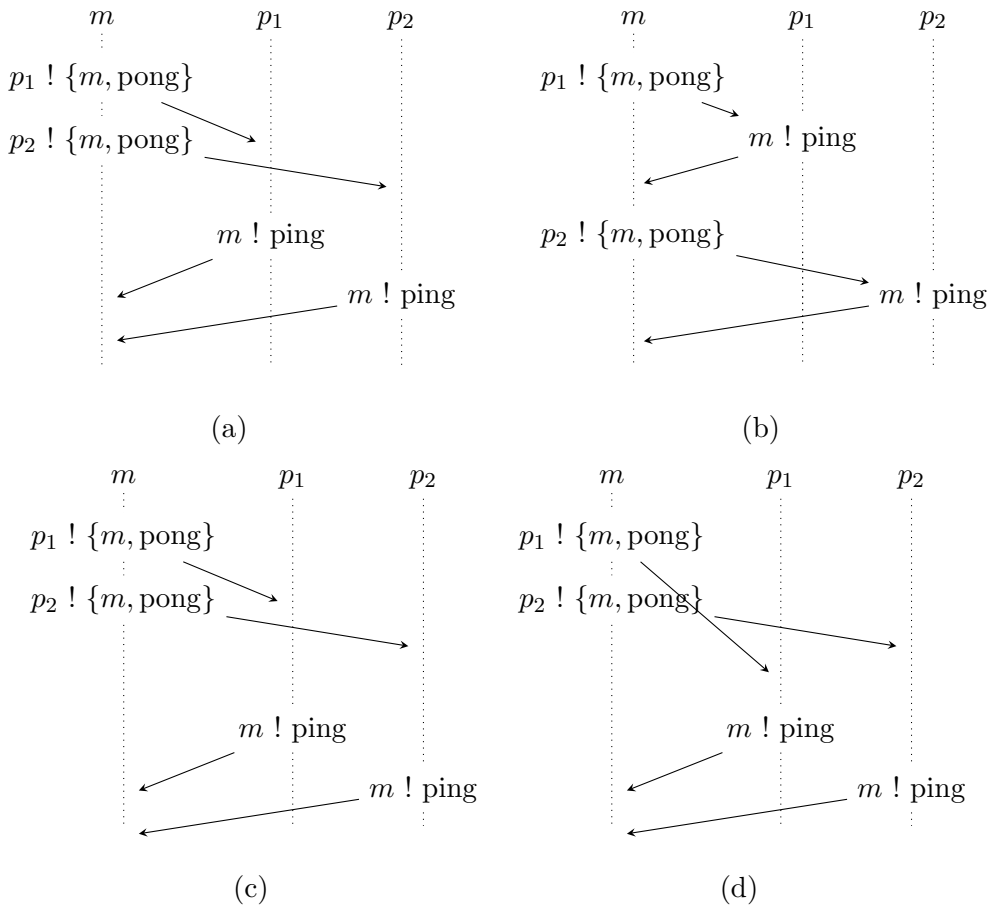


Figure 2.3: Some admissible intearlvings for the program of Fig. 2.2

In the following, we denote by $\overline{o_n}$ a sequence of n syntactic objects for some n, by $\overline{o_{i,j}}$ we denote the sequence $o_i, ..., o_j$ where $i \leq j$. We use $\overline{o}$ when the length of the sequence is not relevant.

The semantics of the expressions is defined by the following labeled transition relation:

$$\longrightarrow : \{Env, Exp\} \times \{Label\} \times \{Env, Exp\}$$

where Env is the domain of environments (i.e. substitutions) and Exp is the domain of expressions, and $Label$ represents an element of the following set:

$$\{\tau, \mathsf{send}(v_1, v_2), \mathsf{rec}(\kappa, \overline{cl_n}), \mathsf{spawn}(\kappa, a/n, [\overline{v_n}]), \mathsf{self}(\kappa)\}$$

The letter ℓ is used to range over the labels. As one can notice, rules have been divided, for simplicity, in two sets, Fig. 2.4 shows the set of rules for sequential expressions, and Fig. 2.5 shows the set of rules for concurrent expressions.

$$Var\ \frac{}{\theta, X \xrightarrow{\tau} \theta, \theta(X)} \qquad Tuple\ \frac{\theta, e_i \xrightarrow{\ell} \theta', e_i'}{\theta, \{\overline{v_{1,i-1}}, e_i, \overline{e_{i+1,n}}\} \xrightarrow{\ell} \theta', \{\overline{v_{1,i-1}}, e_i', \overline{e_{i+1,n}}\}}$$

$$List1\ \frac{\theta, e_1 \xrightarrow{\ell} \theta', e_1'}{\theta, [e_1|e_2] \xrightarrow{\ell} \theta', [e_1'|e_2]} \qquad List2\ \frac{\theta, e_2 \xrightarrow{\ell} \theta', e_2'}{\theta, [v_1|e_2] \xrightarrow{\ell} \theta', [v_1|e_2']}$$

$$Let1\ \frac{\theta, e_1 \xrightarrow{\ell} \theta', e_1'}{\theta, \mathsf{let}\ X = e_1\ \mathsf{in}\ e_2 \xrightarrow{\ell} \theta', \mathsf{let}\ X = e_1'\ \mathsf{in}\ e_2} \qquad Let2\ \frac{}{\theta, \mathsf{let}\ X = v\ \mathsf{in}\ e \xrightarrow{\tau} \theta[X \mapsto v], e}$$

$$Case1\ \frac{\theta, e \xrightarrow{\ell} \theta', e'}{\theta, \mathsf{case}\ e\ \mathsf{of}\ cl_1; ...; cl_n\ \mathsf{end} \xrightarrow{\ell} \theta', \mathsf{case}\ e'\ \mathsf{of}\ cl_1; ...; cl_n\ \mathsf{end}}$$

$$Case2\ \frac{\mathsf{match}(\theta, v, cl_1, ..., cl_n) = \langle \theta_i, e_i \rangle}{\theta, \mathsf{case}\ v\ \mathsf{of}\ cl_1; ...; cl_n\ \mathsf{end} \xrightarrow{\tau} \theta\theta_i, e_i}$$

$$Call1\ \frac{\theta, e_i \xrightarrow{\ell} \theta', e_i' \quad i \in \{1, ..., n\}}{\theta, \mathsf{call}\ \mathsf{op}(\overline{v_{1,i-1}}, e_i, \overline{e_{i+1,n}}) \xrightarrow{\ell} \theta, \mathsf{call}\ \mathsf{op}(\overline{v_{1,i-1}}, e_i', \overline{e_{i+1,n}})}$$

$$Call2\ \frac{\mathsf{eval}(\mathsf{op}, v_1, ..., v_n) = v}{\theta, \mathsf{call}\ \mathsf{op}(v_1, ..., v_n) \xrightarrow{\tau} \theta, v}$$

$$Apply1\ \frac{\theta, e_i \xrightarrow{\ell} \theta', e_i' \quad i \in \{1, ..., n\}}{\theta, \mathsf{apply}\ a/n(\overline{v_{1,i-1}}, e_i, \overline{e_{i+1,n}}) \xrightarrow{\ell} \theta', \mathsf{apply}\ a/n(\overline{v_{1,i-1}}, e_i', \overline{e_{i+1,n}})}$$

$$Apply2\ \frac{\mu(a/n) = \mathsf{fun}(X_1, ..., X_n) \to e}{\theta, \mathsf{apply}\ a/n(v_1, ..., v_n) \xrightarrow{\tau} \theta \cup \{X_1 \mapsto v_1, ..., X_n \mapsto v_n\}, e}$$

Figure 2.4: Standard semantics: evaluation of sequential expressions.

Transitions are labelled either with τ, i.e., transitions without side-effect, or with the label that identifies the reduction of an action with some side-effect. As it is in Erlang, we consider that the order of evaluation of the arguments is fixed from left to right. Let us now analyse some rules that might not be immediately understandable. For the case rule, in Fig. 2.4, the auxiliary function match has been used. The function match, given an environment θ, a value v, and $\overline{cl_n}$ clauses, selects the first clause cl_i such that v matches pat_i, and the guard holds. One may notice that if no pattern matches v then the computation is suspended, in reality, that is never the case since, as already discussed, the Erlang compiler, if not already present, always adds a catch-all clause to the case, which will be used to throw the *badmatch* error.

The semantics also distinguishes between functions which have been defined by the user and functions that are built-ins. The former are evaluated thanks to the auxiliary

$$Send1\ \frac{\theta, e_1 \xrightarrow{\ell} \theta', e_1'}{\theta, e_1\ !\ e_2 \xrightarrow{\ell} \theta', e_1'\ !\ e_2} \qquad Send2\ \frac{\theta, e_2 \xrightarrow{\ell} \theta', e_2'}{\theta, v_1\ !\ e_2 \xrightarrow{\ell} \theta', v_1\ !\ e_2'}$$

$$Send3\ \frac{}{\theta, v_1\ !\ v_2 \xrightarrow{\mathsf{send}(v_1,v_2)} \theta', v_2}$$

$$Receive\ \frac{}{\theta, \mathsf{receive}\ cl_1; ...; cl_n\ \mathsf{end} \xrightarrow{\mathsf{rec}(\kappa,\overline{cl_n})} \theta, \kappa}$$

$$Spawn1\ \frac{\theta, e_i \xrightarrow{\ell} \theta', e_i' \quad i \in \{1, ..., n\}}{\theta, \mathsf{spawn}(a/n, [\overline{v_{1,i-1}}, e_i, \overline{e_{i+1,n}}]) \xrightarrow{\ell} \theta', \mathsf{spawn}(a/n, [\overline{v_{1,i-1}}, e_i', \overline{e_{i+1,n}}])}$$

$$Spawn2\ \frac{}{\theta, \mathsf{spawn}(a/n, [\overline{v_n}]) \xrightarrow{\mathsf{spawn}(\kappa,a/n,[\overline{v_n}])} \theta, \kappa}$$

$$Self\ \frac{}{\theta, \mathsf{self}() \xrightarrow{\mathsf{self}(\kappa)} \theta, \kappa}$$

Figure 2.5: Concurrent semantics: evaluation of concurrent expressions.

function μ, which acts like a map between a function name and its body, while the latter are evaluated with the auxiliary function call.

Now let us move our focus on the rules of Fig. 2.5; one can divide the rules in two different sets, according to whether it is known locally to what the expression evaluates or not. Rules $Send1, Send2, Send3$, which are for "!", belong to the first set, in fact, it is always known to what they evaluate. The remaining rules belong to the second set, indeed, a fresh symbol κ is returned, then the duty of binding κ to the right value is left to the rules of Fig. 2.6, i.e., the selected expression in case of a *Receive* rule or a pid in rules *Spawn* and *Self*. In these cases, the transition's label contains all the information needed to perform the evaluation at a system level, including which one is the value of κ.

Ultimately, let us consider the system rules of Fig. 2.6. As already mentioned, Γ indicates the global mailbox, where a message is stored after being sent from a process, the tuple $\langle p, (\theta, e), q\rangle$ indicates an arbitrary process of the system and Π denotes the remaining processes.
Rule *Seq* represents a sequential action, it only affects the environment and the expression of the process.
Rule *Send* adds the pair (p'', v) to the global mailbox, it indicates that the process p has performed a send. As already discussed in Section 2.2, Γ is necessary to make sure that every possible interleaving is admissible.
Rule *Receive*, through the auxiliary function matchrec, allows a process to receive a message that has been stored in its own queue. The auxiliary function matchrec is

$$Seq\frac{\theta, e \xrightarrow{\tau} \theta', e'}{\Gamma; \langle p, (\theta, e), q\rangle \mid \Pi \hookrightarrow \Gamma; \langle p, (\theta', e'), q\rangle \mid \Pi}$$

$$Send\frac{\theta, e \xrightarrow{\mathsf{send}(p'', v)} \theta', e'}{\Gamma; \langle p, (\theta, e), q\rangle \mid \Pi \hookrightarrow \Gamma \cup (p'', v); \langle p, (\theta', e'), q\rangle \mid \Pi}$$

$$Receive\frac{\theta, e \xrightarrow{\mathsf{rec}(\kappa, \overline{cl_n})} \theta', e' \quad \mathsf{matchrec}(\theta, \overline{cl_n}, q) = (\theta_i, e_i, v)}{\Gamma; \langle p, (\theta, e), q\rangle \mid \Pi \hookrightarrow \Gamma; \langle p, (\theta'\theta_i, e'\{\kappa \mapsto e_i\}), q \backslash\backslash v\rangle \mid \Pi}$$

$$Spawn\frac{\theta, e \xrightarrow{\mathsf{spawn}(\kappa, a/n, [\overline{v_n}])} \theta', e' \quad p' \text{ is a fresh pid}}{\Gamma; \langle p, (\theta, e), q\rangle \mid \Pi \hookrightarrow \Gamma; \langle p, (\theta', e'\{\kappa \mapsto p'\}), q\rangle \mid \langle p', [\,], (id, \mathsf{apply}\ a/n(\overline{v_n}), [\,]\rangle \mid \Pi}$$

$$Self\frac{\theta, e \xrightarrow{\mathsf{self}(\kappa)} \theta', e'}{\Gamma; \langle p, (\theta, e), q\rangle \mid \Pi \hookrightarrow \Gamma; \langle p, (\theta', e'\{\kappa \mapsto p\}), q\rangle \mid \Pi}$$

$$Sched\frac{}{\Gamma \cup \{(p, v)\}; \langle p, (\theta, e), q\rangle \mid \Pi \hookrightarrow \Gamma; \langle p, (\theta, e), v : q\rangle \mid \Pi}$$

Figure 2.6: Standard semantics: system rules.

similar to match used in rule *Case2* in Fig. 2.4; the difference is that in this scenario, instead of the value of the case, the queue of messages q is taken in input, and the output contains, in addition to θ' and e_i, v, which is the first message that has matched the pattern of one of the clauses. Finally, κ is bounded to the expression of the selected clause, e_i, and the environment is enriched with the matching substitution.
Rule *Spawn* creates a new process and adds it to the system, the newly created process' environment is represented by id, since no variable has been possibly bounded to a value yet, the process' expression is denoted by the application of the function a/n to $\overline{v_n}$, and, ultimately, the process' mailbox is denoted by the empty queue. Here, κ is bounded to the newly spawned process' pid.
Rule *Sched* is the rule which, non-deterministically, delivers a message from the global mailbox to the receiving process, it is by applying rule *Sched* in different orders that we can simulate different interleaving of the system.

Example 2.3. Figures 2.7, and 2.8 show the derivation for the program depicted in Fig. 2.2. The derivation coincides with the interleaving shown in Fig. 2.9 (c), therefore it is not the only one. For clarity in each transition we have underlined the reduced expression.

$$
\begin{array}{lll}
 & \{\,\}; & \langle \mathrm{m}, (id, \underline{\mathsf{apply}\ \mathrm{main}/0\ [\,]}), [\,]\rangle \\
\hookrightarrow_{Seq} & \{\,\}; & \langle \mathrm{m}, (id, \mathsf{let}\ P1 = \underline{\mathsf{spawn}(\mathrm{ping}/0, [\,])}\ \mathsf{in}\ \ldots, [\,]\rangle \\
\hookrightarrow_{Spawn} & \{\,\}; & \langle \mathrm{m}, (id, \underline{\mathsf{let}\ P1 = \mathrm{p}_1}\ \mathsf{in}\ \ldots, [\,]\rangle \\
 & & \mid \langle \mathrm{p}_1, (id, \overline{\mathsf{apply}\ \mathrm{ping}/0\ [\,], [\,]}\rangle \\
\hookrightarrow_{Seq} & \{\,\}; & \langle \mathrm{m}, (\{P1 \mapsto \mathrm{p}_1\}, \mathsf{let}\ P2 = \underline{\mathsf{spawn}(\mathrm{ping}/0, [\,])}\ \mathsf{in}\ \ldots, [\,]\rangle \\
 & & \mid \langle \mathrm{p}_1, (id, \overline{\mathsf{apply}\ \mathrm{ping}/0\ [\,], [\,]}\rangle \\
\hookrightarrow_{Spawn} & \{\,\}; & \langle \mathrm{m}, (\{P1 \mapsto \mathrm{p}_1\}, \underline{\mathsf{let}\ P2 = \mathrm{p}_2\ \mathsf{in}\ \ldots}, [\,]\rangle \\
 & & \mid \langle \mathrm{p}_1, (id, \mathsf{apply}\ \mathrm{ping}/0\ [\,], [\,]\rangle \mid \langle \mathrm{p}_2, (id, \mathsf{apply}\ \mathrm{ping}/0\ [\,], [\,]\rangle \\
\hookrightarrow_{Seq} & \{\,\}; & \langle \mathrm{m}, (\{P1 \mapsto \mathrm{p}_1, P2 \mapsto \mathrm{p}_2\}, \mathsf{let}\ S = \underline{\mathsf{self}()}\ \mathsf{in}\ \ldots, [\,]\rangle \\
 & & \mid \langle \mathrm{p}_1, (id, \mathsf{apply}\ \mathrm{ping}/0\ [\,], [\,]\rangle \mid \langle \mathrm{p}_2, (id, \mathsf{apply}\ \mathrm{ping}/0\ [\,], [\,]\rangle \\
\hookrightarrow_{Self} & \{\,\}; & \langle \mathrm{m}, (\{P1 \mapsto \mathrm{p}_1, P2 \mapsto \mathrm{p}_2\}, \underline{\mathsf{let}\ S = \mathrm{m}\ \mathsf{in}\ \ldots}, [\,]\rangle \\
 & & \mid \langle \mathrm{p}_1, (id, \mathsf{apply}\ \mathrm{ping}/0\ [\,], [\,]\rangle \mid \langle \mathrm{p}_2, (id, \mathsf{apply}\ \mathrm{ping}/0\ [\,], [\,]\rangle \\
\hookrightarrow_{Seq} & \{\,\}; & \langle \mathrm{m}, (\{P1 \mapsto \mathrm{p}_1, P2 \mapsto \mathrm{p}_2, S \mapsto \mathrm{m}\}, \mathsf{let}\ \_ = \underline{P1}\ !\ \{S, \mathrm{ping}\}\ \mathsf{in}\ \ldots, [\,]\rangle \\
 & & \mid \langle \mathrm{p}_1, (id, \mathsf{apply}\ \mathrm{ping}/0\ [\,], [\,]\rangle \mid \langle \mathrm{p}_2, (id, \mathsf{apply}\ \mathrm{ping}/0\ [\,], [\,]\rangle \\
\hookrightarrow_{Seq} & \{\,\}; & \langle \mathrm{m}, (\{P1 \mapsto \mathrm{p}_1, P2 \mapsto \mathrm{p}_2, S \mapsto \mathrm{m}\}, \mathsf{let}\ \_ = \mathrm{p}_1\ !\ \{\underline{S}, \mathrm{ping}\}\ \mathsf{in}\ \ldots, [\,]\rangle \\
 & & \mid \langle \mathrm{p}_1, (id, \mathsf{apply}\ \mathrm{ping}/0\ [\,], [\,]\rangle \mid \langle \mathrm{p}_2, (id, \mathsf{apply}\ \mathrm{ping}/0\ [\,], [\,]\rangle \\
\hookrightarrow_{Seq} & \{\,\}; & \langle \mathrm{m}, (\{P1 \mapsto \mathrm{p}_1, P2 \mapsto \mathrm{p}_2, S \mapsto \mathrm{m}\}, \mathsf{let}\ \_ = \underline{\mathrm{p}_1\ !\ \{\mathrm{m}, \mathrm{ping}\}}\ \mathsf{in}\ \ldots, [\,]\rangle \\
 & & \mid \langle \mathrm{p}_1, (id, \mathsf{apply}\ \mathrm{ping}/0\ [\,], [\,]\rangle \mid \langle \mathrm{p}_2, (id, \mathsf{apply}\ \mathrm{ping}/0\ [\,], [\,]\rangle \\
\hookrightarrow_{Send} & \{m_1\}; & \langle \mathrm{m}, (\{P1 \mapsto \mathrm{p}_1, P2 \mapsto \mathrm{p}_2, S \mapsto \mathrm{m}\}, \underline{\mathsf{let}\ \_ = \{\mathrm{m}, \mathrm{ping}\}\ \mathsf{in}\ \ldots}, [\,]\rangle \\
 & & \mid \langle \mathrm{p}_1, (id, \mathsf{apply}\ \mathrm{ping}/0\ [\,], [\,]\rangle \mid \langle \mathrm{p}_2, (id, \mathsf{apply}\ \mathrm{ping}/0\ [\,], [\,]\rangle \\
\hookrightarrow_{Seq} & \{m_1\}; & \langle \mathrm{m}, (\{P1 \mapsto \mathrm{p}_1, P2 \mapsto \mathrm{p}_2, S \mapsto \mathrm{m}\}, \underline{P2}\ !\ \{M, \mathrm{ping}\}\rangle, [\,] \\
 & & \mid \langle \mathrm{p}_1, (id, \mathsf{apply}\ \mathrm{ping}/0\ [\,], [\,]\rangle\ \mid \langle \mathrm{p}_2, (id, \mathsf{apply}\ \mathrm{ping}/0\ [\,], [\,]\rangle \\
\hookrightarrow_{Seq} & \{m_1\}; & \langle \mathrm{m}, (\{P1 \mapsto \mathrm{p}_1, P2 \mapsto \mathrm{p}_2, S \mapsto \mathrm{m}\}, \mathrm{p}_2\ !\ \{\underline{M}, \mathrm{ping}\}, [\,]\rangle \\
 & & \mid \langle \mathrm{p}_1, (id, \mathsf{apply}\ \mathrm{ping}/0\ [\,], [\,]\rangle \mid \langle \mathrm{p}_2, (id, \mathsf{apply}\ \mathrm{ping}/0\ [\,], [\,]\rangle \\
\hookrightarrow_{Send} & \{m_1\}; & \langle \mathrm{m}, (\{P1 \mapsto \mathrm{p}_1, P2 \mapsto \mathrm{p}_2, S \mapsto \mathrm{m}\}, \underline{\mathrm{p}_2\ !\ \{\mathrm{m}, \mathrm{ping}\}}, [\,]\rangle \\
 & & \mid \langle \mathrm{p}_1, (id, \mathsf{apply}\ \mathrm{ping}/0\ [\,], [\,]\rangle \mid \langle \mathrm{p}_2, (id, \mathsf{apply}\ \mathrm{ping}/0\ [\,], [\,]\rangle \\
\hookrightarrow_{Seq} & \{m_1, m_2\}; & \langle \mathrm{m}, (\{P1 \mapsto \mathrm{p}_1, P2 \mapsto \mathrm{p}_2, S \mapsto \mathrm{m}\}, \{\mathrm{m}, \mathrm{ping}\}, [\,]\rangle \\
 & & \mid \langle \mathrm{p}_1, (id, \underline{\mathsf{apply}\ \mathrm{ping}/0\ [\,]}, [\,]\rangle \\
 & & \mid \langle \mathrm{p}_2, (id, \overline{\mathsf{apply}\ \mathrm{ping}/0\ [\,], [\,]}\rangle \\
\hookrightarrow_{Seq} & \{m_1, m_2\}; & \langle \mathrm{m}, (\{P1 \mapsto \mathrm{p}_1, P2 \mapsto \mathrm{p}_2, S \mapsto \mathrm{m}\}, \{\mathrm{m}, \mathrm{ping}\}, [\,]\rangle \\
 & & \mid \langle \mathrm{p}_1, (id, \mathsf{receive}\{P, \mathrm{ping}\} \to P\ !\ \mathrm{pong}\ \mathsf{end}, [\,]\rangle \\
 & & \mid \langle \mathrm{p}_2, (id, \mathsf{apply}\ \mathrm{ping}/0\ [\,], [\,]\rangle
\end{array}
$$

Figure 2.7: A possible interleaving for the program in Fig. 2.2, where $m_1 = (\mathrm{p}_1, \{\mathrm{m}, \mathrm{ping}\}), m_2 = (\mathrm{p}_1, \{\mathrm{m}, \mathrm{ping}\})$ (part 1/2)

$\hookrightarrow_{Sched}$ $\{m_2\}$; $\langle \mathrm{m}, (\{P1 \mapsto \mathrm{p}_1, P2 \mapsto \mathrm{p}_2, S \mapsto \mathrm{m}\}, \{\mathrm{m}, \mathrm{ping}\}, [\,]\rangle$
$\mid \langle \mathrm{p}_1, (id, \underline{\mathsf{receive}\{P, \mathrm{ping}\} \to P \mathbin{!} \mathrm{pong}\ \mathsf{end}}, [\{\mathrm{m}, \mathrm{ping}\}]\rangle$
$\mid \langle \mathrm{p}_2, (id, \mathsf{apply}\ \mathrm{ping}/0\ [\,], [\,]\rangle$

$\hookrightarrow_{Receive}$ $\{m_2\}$; $\langle \mathrm{m}, (\{P1 \mapsto \mathrm{p}_1, P2 \mapsto \mathrm{p}_2, S \mapsto \mathrm{m}\}, \{\mathrm{m}, \mathrm{ping}\}, [\,]\rangle$
$\mid \langle \mathrm{p}_1, (\{P \mapsto \mathrm{m}\}, \underline{P} \mathbin{!} \mathrm{pong}, [\,]\rangle$
$\mid \langle \mathrm{p}_2, (id, \mathsf{apply}\ \mathrm{ping}/0\ [\,], [\,]\rangle$

$\hookrightarrow_{Seq}$ $\{m_2\}$; $\langle \mathrm{m}, (\{P1 \mapsto \mathrm{p}_1, P2 \mapsto \mathrm{p}_2, S \mapsto \mathrm{m}\}, \{\mathrm{m}, \mathrm{ping}\}, [\,]\rangle$
$\mid \langle \mathrm{p}_1, (\{P \mapsto \mathrm{m}\}, \underline{\mathrm{m} \mathbin{!} \mathrm{pong}}, [\,]\rangle$
$\mid \langle \mathrm{p}_2, (id, \mathsf{apply}\ \mathrm{ping}/0\ [\,], [\,]\rangle$

$\hookrightarrow_{Send}$ $\{m_2, m_3\}$; $\langle \mathrm{m}, (\{P1 \mapsto \mathrm{p}_1, P2 \mapsto \mathrm{p}_2, S \mapsto \mathrm{m}\}, \{\mathrm{m}, \mathrm{ping}\}, [\,]\rangle$
$\mid \langle \mathrm{p}_1, (\{P \mapsto \mathrm{m}\}, \mathrm{pong}, [\,]\rangle$
$\mid \langle \mathrm{p}_2, (id, \underline{\mathsf{apply}\ \mathrm{ping}/0\ [\,]}, [\,]\rangle$

$\hookrightarrow_{Seq}$ $\{m_2, m_3\}$; $\langle \mathrm{m}, (\{P1 \mapsto \mathrm{p}_1, P2 \mapsto \mathrm{p}_2, S \mapsto \mathrm{m}\}, \{\mathrm{m}, \mathrm{ping}\}, [\,]\rangle$
$\mid \langle \mathrm{p}_1, (\{P \mapsto \mathrm{m}\}, \mathrm{pong}, [\,]\rangle$
$\mid \langle \mathrm{p}_2, (id, \mathsf{receive}\{P, \mathrm{ping}\} \to P \mathbin{!} \mathrm{pong}\ \mathsf{end}, [\,]\rangle$

$\hookrightarrow_{Sched}$ $\{m_3\}$; $\langle \mathrm{m}, (\{P1 \mapsto \mathrm{p}_1, P2 \mapsto \mathrm{p}_2, S \mapsto \mathrm{m}\}, \{\mathrm{m}, \mathrm{ping}\}, [\,]\rangle$
$\mid \langle \mathrm{p}_1, (\{P \mapsto \mathrm{m}\}, \mathrm{pong}, [\,]\rangle$
$\mid \langle \mathrm{p}_2, (id, \underline{\mathsf{receive}\{P, \mathrm{ping}\} \to P \mathbin{!} \mathrm{pong}\ \mathsf{end}}, [\{\mathrm{m}, \mathrm{ping}\}]\rangle$

$\hookrightarrow_{Receive}$ $\{m_3\}$; $\langle \mathrm{m}, (\{P1 \mapsto \mathrm{p}_1, P2 \mapsto \mathrm{p}_2, S \mapsto \mathrm{m}\}, \{\mathrm{m}, \mathrm{ping}\}, [\,]\rangle$
$\mid \langle \mathrm{p}_1, (\{P \mapsto \mathrm{m}\}, \mathrm{pong}, [\,]\rangle$
$\mid \langle \mathrm{p}_2, (\{P \mapsto \mathrm{m}\}, \underline{P} \mathbin{!} \mathrm{pong}, [\,]\rangle$

$\hookrightarrow_{Seq}$ $\{m_3\}$; $\langle \mathrm{m}, (\{P1 \mapsto \mathrm{p}_1, P2 \mapsto \mathrm{p}_2, S \mapsto \mathrm{m}\}, \{\mathrm{m}, \mathrm{ping}\}, [\,]\rangle$
$\mid \langle \mathrm{p}_1, (\{P \mapsto \mathrm{m}\}, \mathrm{pong}, [\,]\rangle$
$\mid \langle \mathrm{p}_2, (\{P \mapsto \mathrm{m}\}, \underline{\mathrm{m} \mathbin{!} \mathrm{pong}}, [\,]\rangle$

$\hookrightarrow_{Send}$ $\{m_3, m_4\}$; $\langle \mathrm{m}, (\{P1 \mapsto \mathrm{p}_1, P2 \mapsto \mathrm{p}_2, S \mapsto \mathrm{m}\}, \{\mathrm{m}, \mathrm{ping}\}, [\,]\rangle$
$\mid \langle \mathrm{p}_1, (\{P \mapsto \mathrm{m}\}, \mathrm{pong}, [\,]\rangle$
$\mid \langle \mathrm{p}_2, (\{P \mapsto \mathrm{m}\}, \mathrm{pong}, [\,]\rangle$

$\hookrightarrow_{Sched}$ $\{m_4\}$; $\langle \mathrm{m}, (\{P1 \mapsto \mathrm{p}_1, P2 \mapsto \mathrm{p}_2, S \mapsto \mathrm{m}\}, \{\mathrm{m}, \mathrm{ping}\}, [\mathrm{pong}]\rangle$
$\mid \langle \mathrm{p}_1, (\{P \mapsto \mathrm{m}\}, \mathrm{pong}, [\,]\rangle$
$\mid \langle \mathrm{p}_2, (\{P \mapsto \mathrm{m}\}, \mathrm{pong}, [\,]\rangle$

$\hookrightarrow_{Sched}$ $\{\,\}$; $\langle \mathrm{m}, (\{P1 \mapsto \mathrm{p}_1, P2 \mapsto \mathrm{p}_2, S \mapsto \mathrm{m}\}, \{\mathrm{m}, \mathrm{ping}\}, [\mathrm{pong}, \mathrm{pong}]\rangle$
$\mid \langle \mathrm{p}_1, (\{P \mapsto \mathrm{m}\}, \mathrm{pong}, [\,]\rangle$
$\mid \langle \mathrm{p}_2, (\{P \mapsto \mathrm{m}\}, \mathrm{pong}, [\,]\rangle$

Figure 2.8: A possible interleaving for the program in Fig. 2.2, where $m_1 = (\mathrm{p}_1, \{m, \mathrm{ping}\}), m_2 = (\mathrm{p}_2, \{m, \mathrm{ping}\}), m_3 = (m, \mathrm{pong})$, and $m_4 = (m, \mathrm{pong})$ (part 2/2)

2.2.1 Erlang concurrency

To be able to define causal consistency we need to provide also a definition of concurrency, in this case, again following [1], we will accomplish this by providing the opposite notion of conflict. Given two systems s_1, s_2 we denote with $s_1 \hookrightarrow^* s_2$ a *derivation*, a one step derivation is called *transition*. By $d, d', d_1, d_2, ...$ we denote derivations, by $t, t', t_1, t_2, ...$ we denote transitions. Transitions are labelled as follows: $s_1 \hookrightarrow_{p,r} s_2$ where

- p represents the pid of the process performing the action or the pid of the process receiving the message if the selected rule is *Sched*
- r represents the rule applied in order to perform the transition and spans over the rules of Fig. 2.6

Given a derivation $d = (s_1 \hookrightarrow^* s_2)$ we define $\mathsf{init}(d) = s_1$ and $\mathsf{final}(d) = s_2$, two derivations are composable, and we denote it by $d_1; d_2$, if $\mathsf{final}(d_1) = \mathsf{init}(d_2)$. Two derivations, d_1 and d_2, are *cofinal* if $\mathsf{final}(d_1) = \mathsf{final}(d_2)$, and *coinitial* if $\mathsf{init}(d_1) = \mathsf{init}(d_2)$. With ϵ_s we denote the empty derivation, i.e., $s \hookrightarrow^* s$. In the following we will restrict our attention only to reachable systems, the formal definition is provided below.

Definition 2.3 (Concurrent transitions)**.** Given two coinitial transitions $t_1 = (s \hookrightarrow_{p_1,r_1} s_1)$ and $t_2 = (s \hookrightarrow_{p_2,r_2} s_2)$, we say that they are *in conflict* if they consider the same process, i.e., $p_1 = p_2$, and either $r_1 = r_2 = Sched$ or one transition applies rule *Sched* and the other applies rule *Receive*. Two coinitial transitions are *concurrent* if they are not in conflict.

2.3 A reversible semantics

As we mentioned in Section 1.2, in order to perform reversible computation we need to keep a history of the computation of each process. In this section two transition relations will be introduced, $\rightharpoonup$, which denotes the *forward* semantics, and $\leftharpoondown$, which denotes the *backward* semantics. The forward semantics is a conservative extension of the semantics presented in Fig. 2.6, indeed it enriches the already presented semantics with a typical Landauer embedding, to keep track of previous states of each process of the system. In contrast, the reversible semantics allows us to *undo* the computation of each process step by step.

One crucial difference between the system semantics already presented and the forward semantics, which we will soon introduce, is the need to uniquely identify, with a time-stamp, each message exchanged between two processes of the system. Before presenting the two semantics and diving into technical details, we will illustrate, with an example, why it is fundamental to have a unique identifier for each message.

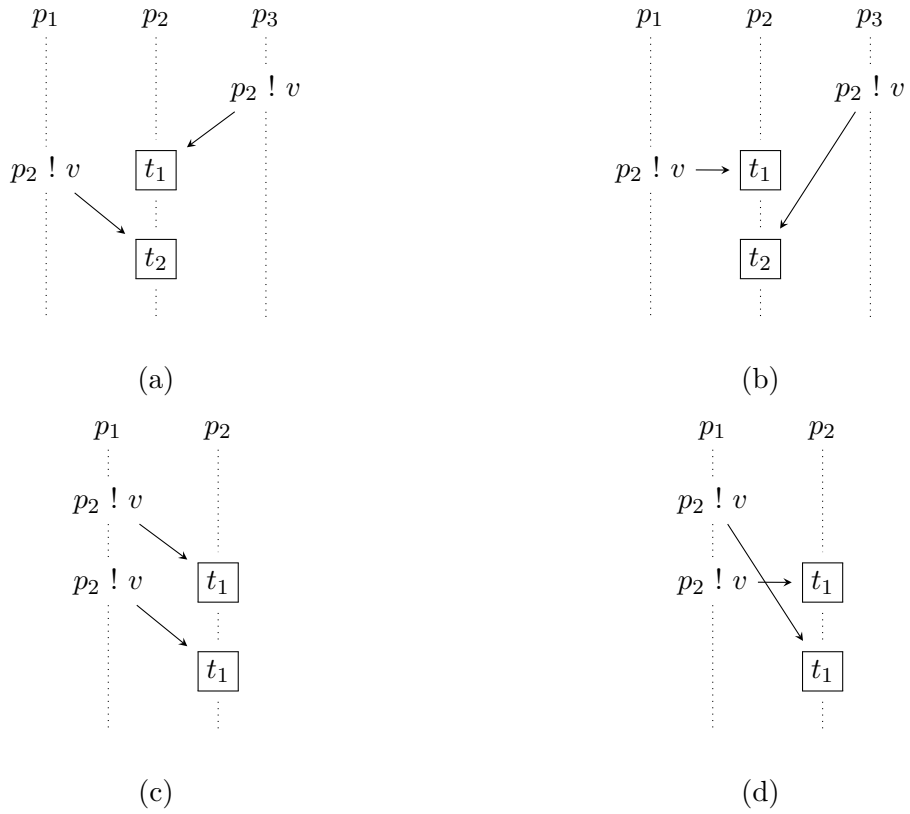


Figure 2.9: Different interleavings

As we mentioned already, in order to ensure causal-consistency before undoing an action we have to undo all its consequences, this also means that we have to undo the exchange of messages following the original order. Now let us consider the examples (a) and (b) of Fig. 2.9, two processes, namely p_1 and p_3, send the same message, namely v, to p_2. Now consider the case where one has to undo the action of p_3 up to $p_2 ! v$, to ensure causal-consistency we need to undo all the consequences of $p_2 ! v$, but with the current semantic it is impossible to determine whether it is necessary to undo the computation of p_2 up to $\boxed{t_2}$ or to $\boxed{t_1}$. Indeed, if we consider scenario (a) we need to undo p_2's computation up to $\boxed{t_1}$, conversely, if we consider scenario (b) we need to undo p_2's computation up to $\boxed{t_2}$. One possible solution consists in storing the pid of the sender, thanks to this extra information it becomes trivial to know whether to undo up to $\boxed{t_1}$ or to $\boxed{t_2}$ (it is enough to check if $\boxed{t_2}$ corresponds to the message sent by p_3, if yes then it is enough to undo up to $\boxed{t_2}$, otherwise p_2's computation need to be undone up to $\boxed{t_1}$).

However, the solution just presented is not general enough, to prove it let us discuss scenario (c) and (d). In both scenarios, p_1 sends two, identical, messages to p_2, now in

$$(Seq)\qquad \frac{\theta, e \xrightarrow{\tau} \theta', e'}{\Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi \rightharpoonup \Gamma; \langle p, \tau(\theta, e) : h, (\theta', e'), q\rangle \mid \Pi}$$

$$(Send)\qquad \frac{\theta, e \xrightarrow{\mathsf{send}(p'', v)} \theta', e' \quad \lambda \text{ is a fresh identifier}}{\Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi \rightharpoonup \Gamma \cup (p'', \{\lambda, v\}); \langle p, \mathsf{send}(\theta, e, p'', \{\lambda, v\}) : h, (\theta', e'), q\rangle \mid \Pi}$$

$$(Receive)\qquad \frac{\theta, e \xrightarrow{\mathsf{rec}(\kappa, \overline{cl_n})} \theta', e' \quad \mathsf{matchrec}(\theta, \overline{cl_n}, q = (\theta_i, e_i, \{\lambda, v\})}{\Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi \rightharpoonup \Gamma; \langle p, \mathsf{rec}(\theta, e, \{\lambda, v\}, q) : h, (\theta'\theta_i, e'\{\kappa \mapsto e_i\}), q \backslash\backslash \{\lambda, v\}\rangle \mid \Pi}$$

$$(Spawn)\qquad \frac{\theta, e \xrightarrow{\mathsf{spawn}(\kappa, a/n, [\overline{v_n}])} \theta', e' \quad p' \text{ is a fresh identifier}}{\begin{array}{l}\Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi \rightharpoonup \Gamma; \langle p, \mathsf{spawn}(\theta, e, p') : h, (\theta', e'\{\kappa \mapsto p'\}), q\rangle \\ \qquad\qquad \mid \langle p', [\,], (id, \mathsf{apply}\ a/n(\overline{v_n})), [\,]\rangle \mid \Pi\end{array}}$$

$$(Self)\qquad \frac{\theta, e \xrightarrow{\mathsf{self}(\kappa)} \theta', e'}{\Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi \rightharpoonup \Gamma; \langle p, \mathsf{self}(\theta, e) : h, (\theta', e'\{\kappa \mapsto p\}), q\rangle \mid \Pi}$$

$$(Sched)\qquad \frac{}{\Gamma \cup \{(p, \{\lambda, v\})\}; \langle p, h, (\theta, e), q\rangle \mid \Pi \rightharpoonup \Gamma; \langle p, h, (\theta, e), \{\lambda, v\} : q\rangle \mid \Pi}$$

Figure 2.10: Forward reversible semantics.

order to undo the consequences of p_1's first send it is necessary to undo p_2's actions up to $\boxed{t_1}$. Unfortunately, with just the sender, as meta-data of the message, it is impossible to distinguish $\boxed{t_1}$ and $\boxed{t_2}$, therefore, some additional information must be taken into account, namely, a unique identifier for each message. One could argue that there are different semantics, and it is indeed true, one of them, for instance, is the semantics that just considers the message v. However, using such less precise semantics would imply that the backward semantics is unpredictable, i.e., we could undo the wrong message. Moreover, defining the notion of concurrency would become much trickier, indeed, one would like to have conflict only between the message v and the "last" delivery of such message, therefore it has been decided that every message is unique.

Now let us analyse the rules of the forward semantics, depicted in Fig. 2.10. As one can notice, these rules are an extension of the rules of Fig. 2.6, where each process is enriched with a history. Precisely, every time that a process performs a step the history keeps track of the action executed and of the piece of information necessary to restore the previous state. As already discussed, the approach used to store information could be optimised, but this goal would be orthogonal to both the purpose of [1] and ours. Before proceeding, let us observe that in rule *Receive* the auxiliary function matchrec now is able to deal with messages of the form $\{\lambda, v\}$, this result is obtained by merely

```
parent/0 = fun () → let C1 = spawn(child/0, [ ])
                    let C2 = spawn(child/0, [ ])
                    in let S = self()
                    in let _ = C1 ! {S, {sum, 2, 3}},
                    in let _ = C2 ! {S, {divide, 6, 3}},
                    in let Res1 = receive
                       {R1} → R1
                    end in let Res2 = receive
                       {R2} → R2
                    end

child/0 = fun () → receive
                       {P, {sum, A, B}} → P ! A + B
                       {P, {divide, A, B}} → P ! A/B
                    end
```

Figure 2.11: A simple program with two processes

ignoring λ while searching for the first matching message.

Example 2.4. Let us now focus on the program described in Fig. 2.11. Fig. 2.13 shows a possible interleaving under the forward reversible semantics. In order to contain the derivation's dimension we omitted every rule *Seq* both from the trace and from the processes' history. More precisely, we consider the following conventions:

- Processes parent, child1 and child2 are shortened to p, c_1, and c_2
- For convenience we do not show the full expression that needs to be evaluated by each process, instead we show $C[e]$, where C is the context and e is the redex which need to be evaluated.
- Moreover, with "_" we indicate some arguments which are not relevant

Now, let us prove formally that the forward semantics is an extension of the system semantics. First of all, we define del', an auxiliary function that removes the history of each process of a given system; formally:

$$\mathsf{del}'(\langle p, h, (\theta, e), q\rangle) = \langle p, (\theta, e), q\rangle$$

$$\mathsf{del}'(\langle p, h, (\theta, e), q\rangle \mid \Pi) = \langle p, (\theta, e), q\rangle \mid \mathsf{del}'(\Pi)$$

where we assume that Π is not empty. We can now state the conservative extension result.

Theorem 2.1. Let s_1 be a system of the reversible semantics and $s_1 = \mathsf{del}'(s_1')$ a system of the standard semantics. Then, $s_1 \rightharpoonup^* s_2$ iff $s_1' \hookrightarrow^* s_2'$ and $\mathsf{del}'(s_2) = s_2'$.

$$(\overline{Seq}) \qquad \Gamma; \langle p, \tau(\theta, e) : h, (\theta', e'), q\rangle \mid \Pi \leftharpoondown \Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi$$

$$(\overline{Send}) \quad \Gamma \cup \{(p'', \{\lambda, v\})\}; \langle p, \mathsf{send}(\theta, e, p'', \{\lambda, v\}) : h, (\theta', e'), q\rangle \mid \Pi \leftharpoondown \Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi$$

$$(\overline{Receive}) \qquad \Gamma; \langle p, \mathsf{rec}(\theta, e, \{\lambda, v\}, q) : h, (\theta', e'), q \backslash\backslash \{\lambda, v\}\rangle \mid \Pi \leftharpoondown \Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi$$

$$(\overline{Spawn}) \quad \Gamma; \langle p, \mathsf{spawn}(\theta, e, p') : h, (\theta', e'), q\rangle \mid \langle p, [\,], (id, e''), [\,]\rangle \mid \Pi \leftharpoondown \Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi$$

$$(\overline{Self}) \qquad \Gamma; \langle p, \mathsf{self}(\theta, e) : h, (\theta', e'), q\rangle \mid \Pi \leftharpoondown \Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi$$

$$(\overline{Sched}) \qquad \begin{array}{l} \Gamma; \langle p, h, (\theta, e), \{\lambda, v\} : q\rangle \mid \Pi \leftharpoondown \Gamma \cup \{(p, \{\lambda, v\})\}; \langle p, h, (\theta, e), q\rangle \mid \Pi \\ \qquad \text{if the topmost } \mathsf{rec}(...) \text{ item in h (if any) has the} \\ \qquad \text{form } \mathsf{rec}(\theta', e', \{\lambda', v'\}, q') \text{ with } q' \backslash\backslash \{\lambda', v'\} \neq \{\lambda, v\} : q \end{array}$$

Figure 2.12: Backward reversible semantics.

For the proof of this result the interested reader can refer to [1].

Now let us pose our focus on the rules of Fig. 2.12, i.e., the backward semantic rules. As it is evident from the figure, the rules for the backward semantics are meant to restore the previous state of a given process through the history of this one. Let us discuss the most interesting cases:

- Rule $\overline{Send}$ can be undone only when the message is available in Γ, i.e., the global mailbox. If the message is not present in Γ it means that we need to apply first the rule $\overline{Sched}$ to undo the insertion of the message in the receiver's queue of messages. If rule $\overline{Sched}$ is not applicable to the process for the given message, it means that we need to undo some steps of the receiver process, at least until the rule $\overline{Sched}$ becomes available. This is required to ensure causal-consistency, i.e., an action can be undone *iff* every consequence of the action is undone.

- When it comes to rule $\overline{Spawn}$ we need to ensure that the mailbox of the process is empty, as well as its history, this again is required to ensure causal-consistency. It is important to notice that if we undo the spawn of a process it is impossible to have orphan messages, meant to be delivered to him. This is possible thanks to the assumption that every pid used in the program is the result of the evaluation of a spawn, therefore, thanks to causal-consistency, if we are about to undo the spawn it means that we already have undone every possible use of value produced by the evaluation of the spawn expression.

- Rule $\overline{Receive}$ can be applied only when the restored queue will be exactly the same

$$
\begin{array}{lll}
 & \{\,\}; & \langle \mathrm{p}, [\,], (id, C[\underline{\mathsf{apply}\ \mathrm{main}/0\ [\,]})], [\,]\rangle \\
\rightharpoonup * & \{\,\}; & \langle \mathrm{p}, [\,], (id, C[\underline{\mathsf{spawn}(\mathrm{child}/0, [\,])})], [\,]\rangle \\
\rightharpoonup_{Spawn} & \{\,\}; & \langle \mathrm{p}, [\mathsf{spawn}(\_,\_,c_1)], (\_, C[\underline{\mathsf{spawn}(\mathrm{child}/0, [\,])})], [\,]\rangle \\
 & & \mid \langle \mathrm{c}_1, [\,], (id, C[\mathsf{receive}\ldots]), [\,]\rangle \\
\rightharpoonup_{Spawn} & \{\,\}; & \langle \mathrm{p}, [\mathsf{spawn}(\_,\_,c_1), \mathsf{spawn}(\_,\_,c_2)], (\_, C[\underline{\mathsf{self}()}], [\,]\rangle \\
 & & \mid \langle \mathrm{c}_1, [\,], (id, C[\mathsf{receive}\ldots]), [\,]\rangle \mid \langle \mathrm{c}_2, [\,], (id, C[\mathsf{receive}\ldots]), [\,]\rangle \\
\rightharpoonup_{Self} & \{\,\}; & \langle \mathrm{p}, [\mathsf{spawn}(\_,\_,c_1), \mathsf{spawn}(\_,\_,c_2), \mathsf{self}(\_,\_)], \\
 & & (\_, C[\underline{c_1\ !\ \{p, \{sum, 2, 3\}\}}], [\,]\rangle \\
 & & \mid \langle \mathrm{c}_1, [\,], (id, C[\mathsf{receive}\ldots]), [\,]\rangle \mid \langle \mathrm{c}_2, [\,], (id, C[\mathsf{receive}\ldots]), [\,]\rangle \\
\rightharpoonup_{Send} & \{(c_1, m_1)\}; & \langle \mathrm{p}, [\mathsf{spawn}(\_,\_,c_1), \mathsf{spawn}(\_,\_,c_2), \mathsf{self}(\_,\_), \mathsf{send}(\_,\_,c_1,m_1)], \\
 & & (\_, C[\underline{c_2\ !\ \{p, \{divide, 6, 3\}\}}], [\,]\rangle \\
 & & \mid \langle \mathrm{c}_1, [\,], (id, C[\mathsf{receive}\ldots]), [\,]\rangle \mid \langle \mathrm{c}_2, [\,], (id, C[\mathsf{receive}\ldots]), [\,]\rangle \\
\rightharpoonup_{Send} & \{(c_1, m_1), (c_2, m_2)\}; & \langle \mathrm{p}, [\mathsf{spawn}(\_,\_,c_1), \mathsf{spawn}(\_,\_,c_2), \mathsf{self}(\_,\_), \mathsf{send}(\_,\_,c_1,m_1), \\
 & & \mathsf{send}(\_,\_,c_2,m_2)], (\_, C[\mathsf{receive}\ldots], [\,]\rangle \\
 & & \mid \langle \mathrm{c}_1, [\,], (id, C[\mathsf{receive}\ldots]), [\,]\rangle \mid \langle \mathrm{c}_2, [\,], (id, C[\mathsf{receive}\ldots]), [\,]\rangle \\
\rightharpoonup_{Sched} & \{(c_2, m_2)\}; & \langle \mathrm{p}, [\mathsf{spawn}(\_,\_,c_1), \mathsf{spawn}(\_,\_,c_2), \mathsf{self}(\_,\_), \mathsf{send}(\_,\_,c_1,m_1), \\
 & & \mathsf{send}(\_,\_,c_2,m_2)], (\_, C[\mathsf{receive}\ldots], [\,]\rangle \\
 & & \mid \langle \mathrm{c}_1, [\,], (id, C[\underline{\mathsf{receive}\ldots}]), [m_1]\rangle \mid \langle \mathrm{c}_2, [\,], (id, C[\mathsf{receive}\ldots]), [\,]\rangle \\
\rightharpoonup_{Receive} & \{(c_2, m_2)\}; & \langle \mathrm{p}, [\mathsf{spawn}(\_,\_,c_1), \mathsf{spawn}(\_,\_,c_2), \mathsf{self}(\_,\_), \mathsf{send}(\_,\_,c_1,m_1), \\
 & & \mathsf{send}(\_,\_,c_2,m_2)], (\_, C[\mathsf{receive}\ldots], [\,]\rangle \\
 & & \mid \langle \mathrm{c}_1, [\mathsf{rec}(\_,\_,m_1,[m_1]], (\_, C[\underline{p\ !\ 5}]), [\,]\rangle \mid \langle \mathrm{c}_2, [\,], (id, C[\mathsf{receive}\ldots]), [\,]\rangle \\
\rightharpoonup_{Send} & \{(c_2, m_2), (p, m_3)\}; & \langle \mathrm{p}, [\mathsf{spawn}(\_,\_,c_1), \mathsf{spawn}(\_,\_,c_2), \mathsf{self}(\_,\_), \mathsf{send}(\_,\_,c_1,m_1), \\
 & & \mathsf{send}(\_,\_,c_2,m_2)], (\_, C[\mathsf{receive}\ldots], [\,]\rangle \\
 & & \mid \langle \mathrm{c}_1, [\mathsf{rec}(\_,\_,m_1,[m_1]), \mathsf{send}(\_,\_,p,m_3)], (\_, C[5]), [\,]\rangle \\
 & & \mid \langle \mathrm{c}_2, [\,], (id, C[\mathsf{receive}\ldots]), [\,]\rangle \\
\rightharpoonup_{Sched} & \{(p, m_3)\}; & \langle \mathrm{p}, [\mathsf{spawn}(\_,\_,c_1), \mathsf{spawn}(\_,\_,c_2), \mathsf{self}(\_,\_), \mathsf{send}(\_,\_,c_1,m_1), \\
 & & \mathsf{send}(\_,\_,c_2,m_2)], (\_, C[\mathsf{receive}\ldots], [\,]\rangle \\
 & & \mid \langle \mathrm{c}_1, [\mathsf{rec}(\_,\_,m_1,[m_1]), \mathsf{send}(\_,\_,p,m_3)], (\_, C[5]), [\,]\rangle \\
 & & \mid \langle \mathrm{c}_2, [\,], (id, C[\underline{\mathsf{receive}\ldots}]), [m_2]\rangle \\
\rightharpoonup_{Receive} & \{(p, m_3)\}; & \langle \mathrm{p}, [\mathsf{spawn}(\_,\_,c_1), \mathsf{spawn}(\_,\_,c_2), \mathsf{self}(\_,\_), \mathsf{send}(\_,\_,c_1,m_1), \\
 & & \mathsf{send}(\_,\_,c_2,m_2)], (\_, C[\mathsf{receive}\ldots], [\,]\rangle \\
 & & \mid \langle \mathrm{c}_1, [\mathsf{rec}(\_,\_,m_1,[m_1]), \mathsf{send}(\_,\_,p,m_3)], (\_, C[5]), [\,]\rangle \\
 & & \mid \langle \mathrm{c}_2, [\mathsf{rec}(\_,\_,m_2,[m_2])], (\_, C[\underline{p\ !\ 2}]), [\,]\rangle \\
\rightharpoonup_{Send} & \{(p, m_3), (p, m_4)\}; & \langle \mathrm{p}, [\mathsf{spawn}(\_,\_,c_1), \mathsf{spawn}(\_,\_,c_2), \mathsf{self}(\_,\_), \mathsf{send}(\_,\_,c_1,m_1), \\
 & & \mathsf{send}(\_,\_,c_2,m_2)], (\_, C[\mathsf{receive}\ldots], [\,]\rangle \\
 & & \mid \langle \mathrm{c}_1, [\mathsf{rec}(\_,\_,m_1,[m_1]), \mathsf{send}(\_,\_,p,m_3)], (\_, C[5]), [\,]\rangle \\
 & & \mid \langle \mathrm{c}_2, [\mathsf{rec}(\_,\_,m_2,[m_2]), \mathsf{send}(\_,\_,m_4)], (\_, C[2]), [\,]\rangle
\end{array}
$$

Figure 2.13: A derivation under the forward reversible semantics, with $m_1 = \{1, \{p, "ok?"\}\}$, $m_2 = \{2, \{c, "ok!"\}\}$, $m_3 = \{3, \{5\}\}$, and $m_4 = \{4, \{2\}\}$

as it was when the message had been received. This is required to ensure that rule *Receive* does not commute with eventual applications of rule *Sched*.

- Rules $\overline{Sched}$ is the one with the highest degree of freedom, indeed, there are several cases where one can apply rule $\overline{Sched}$ without interfering with the other rules, the only rules that do not commute with $\overline{Sched}$ are another $\overline{Sched}$ or the rule $\overline{Receive}$. The fact that two rules $\overline{Sched}$ do not commute is ensured by the fact that they always apply to the most recent message of the queue and that message, by definition, is unique. The fact that $\overline{Sched}$ does not commute with $\overline{Receive}$ is ensured thanks to the side condition of the rule $\overline{Sched}$.

Example 2.5. Let us focus on Fig. 2.14, where a possible backward derivation for the final state of the computation depicted in Fig. 2.13 is shown. As one can see, the (backward) derivation does not strictly follows the inverse order of the original forward derivation. Of course, a derivation which follow the exact inverse order exists but it is not the only one. In this example we have used the same rules and conventions of Example 2.4.

2.3.1 Properties of the uncontrolled reversible semantics

Given two systems, s_1 and s_2, by means of $s_1 \rightharpoonup^* s_2$ we denote a forward derivation from system s_1 to system s_2, by means of $s_1 \leftharpoondown^* s_2$ we denote a backward derivation from system s_1 to system s_2. If a derivation contains both forward and backward transitions we will denote it by $s_1 \rightleftharpoons^* s_2$. Let us now consider the labelled transition $t = s_1 \rightleftharpoons_{p,r,k} s_2$ and let us analyse the labels:

- p and r represent, respectively, the pid of the process and the rule applied
- k represents the history item, if the selected rule is different from $\overline{Sched}$ or *Sched* otherwise it represents the history item.
- $k = \mathsf{sched}(\{\lambda, v\})$ when the selected rule is either $\overline{Sched}$ or *Sched*.

The notion of init, final, composable, coinitial and cofinal, from Sec. 2.2.1, are extended, to the reversible semantics in the natural way. Given generic label r we indicate with $\overline{r}$ the correspondent inverse rule, i.e., if $r = Send$ then $\overline{r} = \overline{Send}$ (note that if $r = \overline{Receive}$ then $\overline{r} = Receive$). We use the same notation for transitions, if $t = s_1 \rightharpoonup_{p,r,k} s_2$ then $\overline{t} = s_1 \leftharpoondown_{p,\overline{r},k} s_2$, consequently if $t = s_1 \leftharpoondown_{p,\overline{r},k} s_2$ then $\overline{t} = s_1 \rightharpoonup_{p,r,k} s_2$. In other words, $\overline{t}$ is the inverse of t. The definition of inverse is naturally extended to derivations. The zero steps derivations ϵ_s is indicated by $s \rightleftharpoons^* s$.

$\{(p, m_3), (p, m_4)\}$; $\langle \mathrm{p}, [\mathsf{spawn}(\_,\_,c_1), \mathsf{spawn}(\_,\_,c_2), \mathsf{self}(\_,\_), \mathsf{send}(\_,\_,c_1,m_1),$
$\mathsf{send}(\_,\_,c_2,m_2)], (\_, C[\mathsf{receive}\ldots], [\,]\rangle$
$\mid \langle \mathrm{c}_1, [\mathsf{rec}(\_,\_,m_1,[m_1]), \mathsf{send}(\_,\_,p,m_3)], (\_, C[5]), [\,]\rangle$
$\mid \langle \mathrm{c}_2, [\mathsf{rec}(\_,\_,m_2,[m_2]), \underline{\mathsf{send}(\_,\_,m_4)}], (\_, C[2]), [\,]\rangle$

$\leftharpoondown_{\overline{Send}}$ $\{(p, m_3)\}$; $\langle \mathrm{p}, [\mathsf{spawn}(\_,\_,c_1), \mathsf{spawn}(\_,\_,c_2), \mathsf{self}(\_,\_), \mathsf{send}(\_,\_,c_1,m_1),$
$\mathsf{send}(\_,\_,c_2,m_2)], (\_, C[\mathsf{receive}\ldots], [\,]\rangle$
$\mid \langle \mathrm{c}_1, [\mathsf{rec}(\_,\_,m_1,[m_1]), \mathsf{send}(\_,\_,p,m_3)], (\_, C[5]), [\,]\rangle$
$\mid \langle \mathrm{c}_2, [\underline{\mathsf{rec}(\_,\_,m_2,[m_2])}], (\_, C[p\ !\ 2]), [\,]\rangle$

$\leftharpoondown_{\overline{Receive}}\{(p, m_3)\}$; $\langle \mathrm{p}, [\mathsf{spawn}(\_,\_,c_1), \mathsf{spawn}(\_,\_,c_2), \mathsf{self}(\_,\_), \mathsf{send}(\_,\_,c_1,m_1),$
$\mathsf{send}(\_,\_,c_2,m_2)], (\_, C[\mathsf{receive}\ldots], [\,]\rangle$
$\mid \langle \mathrm{c}_1, [\mathsf{rec}(\_,\_,m_1,[m_1]), \mathsf{send}(\_,\_,p,m_3)], (\_, C[5]), [\,]\rangle$
$\mid \langle \mathrm{c}_2, [\,], (id, C[\mathsf{receive}\ldots]), [m_2]\rangle$

$\leftharpoondown_{\overline{Sched}}$ $\{(c_2, m_2), (p, m_3)\}$; $\langle \mathrm{p}, [\mathsf{spawn}(\_,\_,c_1), \mathsf{spawn}(\_,\_,c_2), \mathsf{self}(\_,\_), \underline{\mathsf{send}(\_,\_,c_1,m_1)},$
$\mathsf{send}(\_,\_,c_2,m_2)], (\_, C[\mathsf{receive}\ldots], [\,]\rangle$
$\mid \langle \mathrm{c}_1, [\mathsf{rec}(\_,\_,m_1,[m_1]), \mathsf{send}(\_,\_,p,m_3)], (\_, C[5]), [\,]\rangle$
$\mid \langle \mathrm{c}_2, [\,], (id, C[\mathsf{receive}\ldots]), [\,]\rangle$

$\leftharpoondown_{\overline{Send}}$ $\{(p, m_3)\}$; $\langle \mathrm{p}, [\mathsf{spawn}(\_,\_,c_1), \mathsf{spawn}(\_,\_,c_2), \mathsf{self}(\_,\_), \mathsf{send}(\_,\_,c_1,m_1)],$
$(\_, C[c_2\ !\ \{p, \{divide, 6, 3\}\}], [\,]\rangle$
$\mid \langle \mathrm{c}_1, [\mathsf{rec}(\_,\_,m_1,[m_1]), \underline{\mathsf{send}(\_,\_,p,m_3)}], (\_, C[5]), [\,]\rangle$
$\mid \langle \mathrm{c}_2, [\,], (id, C[\mathsf{receive}\ldots]), [\,]\rangle$

$\leftharpoondown_{\overline{Send}}$ $\{\,\}$; $\langle \mathrm{p}, [\mathsf{spawn}(\_,\_,c_1), \mathsf{spawn}(\_,\_,c_2), \mathsf{self}(\_,\_), \mathsf{send}(\_,\_,c_1,m_1)],$
$(\_, C[c_2\ !\ \{p, \{sum, 3, 2\}\}], [\,]\rangle$
$\mid \langle \mathrm{c}_1, [\underline{\mathsf{rec}(\_,\_,m_1,[m_1])}], (\_, C[p\ !\ 5]), [\,]\rangle \mid \langle \mathrm{c}_2, [\,], (id, C[\mathsf{receive}\ldots]), [\,]\rangle$

$\leftharpoondown_{\overline{Receive}}\{\,\}$; $\langle \mathrm{p}, [\mathsf{spawn}(\_,\_,c_1), \mathsf{spawn}(\_,\_,c_2), \mathsf{self}(\_,\_), \mathsf{send}(\_,\_,c_1,m_1)],$
$(\_, C[c_2\ !\ \{p, \{sum, 3, 2\}\}], [\,]\rangle$
$\mid \langle \mathrm{c}_1, [\,], (id, C[\mathsf{receive}\ldots]), [m_1]\rangle \mid \langle \mathrm{c}_2, [\,], (id, C[\mathsf{receive}\ldots]), [\,]\rangle$

$\leftharpoondown_{\overline{Sched}}$ $\{(c_1, m_1)\}$; $\langle \mathrm{p}, [\mathsf{spawn}(\_,\_,c_1), \mathsf{spawn}(\_,\_,c_2), \mathsf{self}(\_,\_), \underline{\mathsf{send}(\_,\_,c_1,m_1)}],$
$(\_, C[c_2\ !\ \{p, \{sum, 3, 2\}\}], [\,]\rangle$
$\mid \langle \mathrm{c}_1, [\,], (id, C[\mathsf{receive}\ldots]), [\,]\rangle \mid \langle \mathrm{c}_2, [\,], (id, C[\mathsf{receive}\ldots]), [\,]\rangle$

$\leftharpoondown_{\overline{Send}}$ $\{\,\}$; $\langle \mathrm{p}, [\mathsf{spawn}(\_,\_,c_1), \mathsf{spawn}(\_,\_,c_2), \underline{\mathsf{self}(\_,\_)}], (\_, C[\mathsf{let} S = \mathsf{self}()], [\,]\rangle$
$\mid \langle \mathrm{c}_1, [\,], (id, C[\mathsf{receive}\ldots]), [\,]\rangle \mid \langle \mathrm{c}_2, [\,], (id, C[\mathsf{receive}\ldots]), [\,]\rangle$

$\leftharpoondown_{\overline{Self}}$ $\{\,\}$; $\langle \mathrm{p}, [\mathsf{spawn}(\_,\_,c_1), \underline{\mathsf{spawn}(\_,\_,c_2)}], (\_, C[\mathsf{spawn}(child/0, [\,])], [\,]\rangle$
$\mid \langle \mathrm{c}_1, [\,], (id, C[\mathsf{receive}\ldots]), [\,]\rangle \mid \langle \mathrm{c}_2, [\,], (id, C[\mathsf{receive}\ldots]), [\,]\rangle$

$\leftharpoondown_{\overline{Spawn}}$ $\{\,\}$; $\langle \mathrm{p}, [\underline{\mathsf{spawn}(\_,\_,c_1)}], (\_, C[\mathsf{spawn}(\mathrm{child}/0, [\,])], [\,]\rangle$
$\mid \langle \mathrm{c}_1, [\,], (id, C[\mathsf{receive}\ldots]), [\,]\rangle$

$\leftharpoondown_{\overline{Spawn}}$ $\{\,\}$; $\langle \mathrm{p}, [\,], (\_, C[\mathsf{spawn}(\mathrm{child}/0, [\,])], [\,]\rangle$

$\leftharpoondown^*$ $\{\,\}$; $\langle \mathrm{p}, [\,], (\_, C[\mathsf{apply}\ \mathrm{parent}, [\,])], [\,]\rangle$

Figure 2.14: A backward derivation under the reversible semantics, with $m_1 = \{1, \{p, \text{"}ok?\text{"}\}\}$, $m_2 = \{2, \{c, \text{"}ok!\text{"}\}\}$, $m_3 = \{3, \{5\}\}$, and $m_4 = \{4, \{2\}\}$

Definition 2.4. A system is *initial* if it is composed by a single process, and this process has an empty history and an empty queue; furthermore the global mailbox is empty. A system s is reachable if there exists an initial system s_0 and a derivation $s_0 \rightleftharpoons^* s$ using the rules corresponding to a given program.

Now the previous definition of concurrency for the standard semantics will be extended to the reversible semantics.

Definition 2.5 (Concurrent transitions)**.** Given two coinitial transitions, $t_1 = (s \rightleftharpoons_{p_1,r_1,k_1} s_1)$ and $t_2 = (s \rightleftharpoons_{p_2,r_2,k_2} s_2)$, we say that they are in conflict if at least one of the following conditions holds:

- **both transition are forward**, they consider the same process, i.e., $p_1 = p_2$, and either $r_1 = r_2 = \mathit{Sched}$ or one transition applies rule *Sched* and the other transition applies rule *Receive*.
- one is a **forward** transition that applies to a process p, say $p_1 = p$, and the other one is a **backward** transition that undoes the creation of p, i.e., $p_2 = p' \neq p, r_2 = \mathit{Spawn}$ and $k_2 = \mathsf{spawn}(\theta, e, p)$ for some control (θ, e);
- one is a **forward** transition that delivers a message $\{\lambda, v\}$ to a process p, say $p_1 = p$, $r_1 = \mathit{Sched}$ and $k_1 = \mathsf{sched}(\{\lambda, v\})$, and the other one is a **backward** transition that undoes the sending $\{\lambda, v\}$ to p, i.e., $p_2 = p'$, $r_2 = \mathit{Send}$ and $k_2 = \mathsf{send}(\theta, e, p, \{\lambda, v\})$ for some control (θ, e);
- one is a **forward** transition and the other one is a **backward** transition such that $p_1 = p_2$ and either i) both applied rules are different from both *Sched* and $\overline{\mathit{Sched}}$, i.e., $\{r_1, r_2\} \cap \{\mathit{Sched}, \overline{\mathit{Sched}}\} = \emptyset$; ii) one rule is *Sched* and the other one is $\overline{\mathit{Sched}}$; iii) one rule is *Sched* and the other one is $\overline{\mathit{Receive}}$; or iv) one rule is $\overline{\mathit{Sched}}$ and the other one is *Receive*.

Two transitions are *concurrent* if they are *not* in conflict, two backward transitions are always concurrent.

In [1] it is possible to find more results, among which causal consistency. Those results will not be reported here, this because we intend to prove causal-consistency, and the other properties in a different manner, i.e., relying on the work done in [12].

In [12] causal consistency, and other properties are proved on an abstract model (i.e., labelled transition system) by building these properties upon a small set of axioms. Hence, if we prove that those axioms also hold for this (concrete) model, automatically we prove all the properties which follow from these axioms, among which causal-consistency.

$$(\overline{Seq}) \quad \Gamma; \lfloor \langle p, \tau(\theta, e) : h, (\theta', e')q \rangle \rfloor_\Psi \mid \Pi \rightsquigarrow \Gamma; \lfloor \langle p, h, (\theta, e), q \rangle \rfloor_\Psi \mid \Pi$$

$$(\overline{Send1}) \quad \begin{array}{c} \Gamma \cup \{(p'', \{\lambda, v\})\}; \lfloor \langle p, \mathsf{send}(\theta, e, p'', \{\lambda, v\}) : h, (\theta', e'), q \rangle \rfloor_\Psi \mid \Pi \\ \rightsquigarrow \Gamma; \lfloor \langle p, h, (\theta, e), q \rangle \rfloor_\Psi \mid \Pi \end{array}$$

$$(\overline{Send1}) \quad \begin{array}{c} \Gamma; \lfloor \langle p, \mathsf{send}(\theta, e, p'', \{\lambda, v\}) : h, (\theta', e'), q \rangle \rfloor_\Psi \mid \lfloor \langle p'', h'', (\theta'', e''), q' \rangle \rfloor_{\Psi'} \mid \Pi \\ \rightsquigarrow \Gamma; \lfloor \langle p, \mathsf{send}(\theta, e, p'', \{\lambda, v\}) : h, (\theta', e'), q \rangle \rfloor_\Psi \mid \lfloor \langle p'', h'', (\theta'', e''), q' \rangle \rfloor_{\Psi' \cup \{\#^\lambda_{\mathsf{sch}}\}} \mid \Pi \\ \text{if } \{(p, \{\lambda, v\})\} \text{ does not occur in } \Gamma \text{ and } \#^\lambda_{\mathsf{sch}} \notin \Psi' \end{array}$$

$$(\overline{Receive}) \quad \Gamma; \lfloor \langle p, \mathsf{rec}(\theta, e, \{\lambda, v\}, q) : h, (\theta', e'), q \backslash\backslash \{\lambda, v\} \rangle \rfloor_\Psi \mid \Pi \rightsquigarrow \Gamma; \lfloor \langle p, h, (\theta, e), q \rangle \rfloor_\Psi \mid \Pi$$

$$(\overline{Spawn1}) \quad \begin{array}{c} \Gamma; \lfloor \langle p, \mathsf{spawn}(\theta, e, p') : h, (\theta', e'), q \rangle \rfloor_\Psi \mid \lfloor \langle p', [\,], (id, e''), [\,] \rangle \rfloor_{\Psi'} \mid \Pi \\ \rightsquigarrow \Gamma; \lfloor \langle p, h, (\theta, e), q \rangle \rfloor_\Psi \mid \Pi \end{array}$$

$$(\overline{Spawn2}) \quad \begin{array}{c} \Gamma; \lfloor \langle p, \mathsf{spawn}(\theta, e, p') : h, (\theta', e'), q \rangle \rfloor_\Psi \mid \lfloor \langle p', h'', (\theta'', e''), q' \rangle \rfloor_{\Psi'} \mid \Pi \\ \rightsquigarrow \Gamma; \lfloor \langle p, \mathsf{spawn}(\theta, e, p') : h, (\theta', e'), q \rangle \rfloor_\Psi \mid \lfloor \langle p', h'', (\theta'', e''), q' \rangle \rfloor_{\Psi' \cup \{\#_{\mathsf{sp}}\}} \mid \Pi \\ \text{if } h'' \neq [\,] \vee q' \neq [\,] \wedge \#_{\mathsf{sp}} \notin \Psi \end{array}$$

$$(\overline{Self}) \quad \Gamma; \lfloor \langle p, \mathsf{self}(\theta, e) : h, (\theta', e'), q \rangle \rfloor_\Psi \mid \Pi \rightsquigarrow \Gamma; \lfloor \langle p, h, (\theta, e), q \rangle \rfloor_\Psi \mid \Pi$$

$$(\overline{Sched}) \quad \begin{array}{c} \Gamma; \lfloor \langle p, h, (\theta', e'), \{\lambda, v\} : q \rangle \rfloor_\Psi \mid \Pi \rightsquigarrow \Gamma \cup \{(p, \{\lambda, v\})\}; \lfloor \langle p, h, (\theta, e), q \rangle \rfloor_{\Psi \setminus \{\#^\lambda_{\mathsf{sch}}\}} \mid \Pi \\ \text{if the topmost } \mathsf{rec}(...) \text{ item in h (if any) has the} \\ \text{form } \mathsf{rec}(\theta', e', \{\lambda', v'\}, q') \text{ with } q' \backslash\backslash \{\lambda', v'\} \neq \{\lambda, v\} : q \end{array}$$

Figure 2.15: Rollback semantics.

2.4 Rollback semantics

In this section we will describe (a simplified version of) the rollback semantics, following, as usual, [1].

The approach used to define the rollback semantics that we will present here is different from the one that we will use later to define the rollback semantics of distributed CauDEr. Indeed, the rollback semantics that we will present here is *uncontrolled* while the one that we will provide later for distributed CauDEr is *controlled*. Nonetheless, even if the two approaches are different we believe that reporting here the uncontrolled rollback semantics will provide the reader with a full overview of how the first version of CauDEr has been developed. Moreover, the reader by comparing the two (rollback) semantics will have a deeper understanding of how the approach has improved overtime, indeed the approach that we will use later in Section 3.4 has been inspired by [13] and makes much easier both the definition of the semantics and the proof of desirable properties.

Here, a process is said in *rollback* mode when it is annotated with $\lfloor\ \rfloor_\Psi$, where Ψ

represents the set of requests to be undone. We identify two kinds of request:

- $\#_{\mathsf{sp}}$ where sp stands for "spawn", a rollback to undo all the actions of a process, finally deleting it from the system;
- $\#^{\lambda}_{\mathsf{sch}}$ where sch stands for sched: a rollback to undo the actions of a process until the delivery of a message $\{\lambda, v\}$ is undone.

In the following, to simplify the reduction rules we assume that the semantics satisfies the following *structural equivalence*:

$$(SC) \quad \Gamma \mid \lfloor \langle p, h, (\theta, e), q \rangle \rfloor_{\emptyset} \mid \Pi \equiv \Gamma \mid \langle p, h, (\theta, e), q \rangle \mid \Pi$$

The rollback semantics has been introduced to automatically reach previous states of the computation, while maintaining causal-consistency. Indeed, while the backward semantics is able to undo the computation, of a specified process, step by step, with the rollback semantics is possible to specify a target (e.g., the spawn of a process) and the rollback semantics will automatically *roll-back* the system, until the target that we specified is satisfied. The rules for the rollback semantics are shown in Fig. 2.15. Here, we assume that $\Psi \neq \emptyset$, but Ψ' might be empty. Let us analyse the most interesting cases:

- As one can notice, we have two rules for $\overline{Send}$, one considers the scenario where the message can be found in the global mailbox and the other considers the scenario where the message is not in the global mailbox. If we consider the first case, i.e., $\overline{Send1}$, it is possible to proceed and undo the send, the fact that the message is in the global mailbox imply that all the consequences of the send are already undone. On the contrary, if the message is not there it means that before undoing the send we need to undo its consequences. In order to reach this goal, we add to the set of requests of the receiver, the request $\#^{\lambda}_{\mathsf{sch}}$, in this way the rollback semantics will undo all the actions which have a dependency on the send, then it will proceed to undo the actual send.
- A similar scenario can also be found for the rule $\overline{Spawn}$. Similarly to rule $\overline{Send}$, before undoing the spawn of a process, it is necessary to ensure that every action performed by such process has been undone, this can be verified by checking its history and mailbox. On the one hand, we have the case where both history and mailbox are empty, therefore we can proceed and safely undo the spawn. On the other hand, the history of the process is not empty, and in this case we need to add to the set of requests of this process also $\#_{\mathsf{sp}}$. Hence, the rollback semantics will undo every action of the process, and in the end, when both the history will be empty, it will be possible to undo the spawn.
- Finally, rule $\overline{Sched}$ requires the same side-conditions as in the backward semantics, this is required in order to avoid commutations of rules $\overline{Sched}$ and $\overline{Receive}$.

Chapter 3

Distributed CauDEr

In the following of this chapter, we will present the syntax of the extended language supported, followed by the system semantics, and finally by the reversible semantics. In contrast to the language already presented in Section 2.1, the one that will be presented here will include built-in functions meant for distributed programming.

$$
\begin{array}{rcl}
\mathit{module} & ::= & \mathit{fun}_1 \ \ldots \ \mathit{fun}_n \\
\mathit{fun} & ::= & \mathit{fname} = \mathsf{fun}\ (X_1, \ldots, X_n) \rightarrow \mathit{expr} \\
\mathit{fname} & ::= & \mathit{Atom}/\mathit{Integer} \\
\mathit{lit} & ::= & \mathit{Atom} \mid \mathit{Integer} \mid \mathit{Float} \mid [\,] \\
\mathit{expr} & ::= & \mathit{Var} \mid \mathit{lit} \mid \mathit{fname} \mid [\mathit{expr}_1 | \mathit{expr}_2] \mid \{\mathit{expr}_1, \ldots, \mathit{expr}_n\} \\
 & \mid & \mathsf{call}\ \mathit{expr}\ (\mathit{expr}_1, \ldots, \mathit{expr}_n) \mid \mathsf{apply}\ \mathit{expr}\ (\mathit{expr}_1, \ldots, \mathit{expr}_n) \\
 & \mid & \mathsf{case}\ \mathit{expr}\ \mathsf{of}\ \mathit{clause}_1; \ldots; \mathit{clause}_m\ \mathsf{end} \\
 & \mid & \mathsf{let}\ \mathit{Var} = \mathit{expr}_1\ \mathsf{in}\ \mathit{expr}_2 \mid \mathsf{receive}\ \mathit{clause}_1; \ldots; \mathit{clause}_n\ \mathsf{end} \\
 & \mid & \mathsf{spawn}(\mathit{expr}, \mathit{expr}, [\mathit{expr}_1, \ldots, \mathit{expr}_n]) \mid \mathit{expr}_1\,!\,\mathit{expr}_2 \mid \mathsf{self}() \\
 & \mid & \mathsf{start}(\mathit{expr}) \mid \mathsf{node}() \mid \mathsf{nodes}() \\
\mathit{clause} & ::= & \mathit{pat}\ \mathsf{when}\ \mathit{expr}_1 \rightarrow \mathit{expr}_2 \\
\mathit{pat} & ::= & \mathit{Var} \mid \mathit{lit} \mid [\mathit{pat}_1 | \mathit{pat}_2] \mid \{\mathit{pat}_1, \ldots, \mathit{pat}_n\}
\end{array}
$$

Figure 3.1: Extended language syntax rules

3.1 Extended language: syntax

The syntax for the distributed version of CauDEr can be found in Fig. 3.1. As one can notice, the syntax is similar to the one presented in [1], with the addition of three new functions: start, node, nodes. The extended syntax follows the same rules and conventions already presented in Section 2.1. Before proceeding and discussing the new syntax we will introduce the notions of *node* and *network*.

Definition 3.1 (Node)**.** A node is a pool of processes, each node is uniquely identified by an atom of the form *name*@*host*; there is no limit on the number of nodes that a single machine can host, although, the convention is that there is only one node per physical machine.

Definition 3.2 (Network)**.** A network is a set of nodes cooperating together, each node in the network is uniquely identified by its name, in the following we will often refer to the network with the symbol Ω.

Let us now analyse the differences w.r.t. the syntax of Fig.2.1.
First, start allows one to start a new node and add it to the network of nodes cooperating together, if the node is already part of the network an error will be returned. Then, node returns the name of the node where the process which called it is running, and, ultimately, nodes returns the list of nodes of the network (minus the node of the process which invoked the nodes, as it happens in Erlang). Moreover, now the function spawn takes in input three arguments instead of two, the additional argument is used to specify on which node the spawn has to be performed. Indeed, a process is able to spawn processes also in a node different from its own, this can be accomplished just by specifying the node as first argument of the new *spawn*/3.

In a real Erlang environment, in order for a *remote* spawn to succeed, it is necessary that the module, where the function is defined, is loaded in the targeted node. Usually, to load a module in all the nodes of the network, the built-in function *nl*/1 is used, where the argument represents which module has to be loaded. Here, we assume that every node is already provided with the module that contains the function's body, therefore, a spawn can never fail because the module is not loaded.

Now we extend the definition of system already provided in [1]. Previously, a system was defined as a pool of processes with a global mailbox, now a system is defined as a pool of nodes with a global mailbox. Each process belonging to the system is still uniquely identified by its pid, even across different nodes, and they can communicate with each other through asynchronous message passing. The syntax for sending a message from one process to another is still $v_1 \; ! \; v_2$, where v_1 is the pid of the receiver and v_2 is the message, regardless of the fact that the two processes are running in the same node or not.

3.2 The extended language semantics

In this section we will formalise the semantics of the extended language, the definitions included here are an extension of the ones already presented in Section 2.2.

Definition 3.3 (Process)**.** A process is a tuple $\langle node, p, (\theta, e), q \rangle$, where *node* represents the node where the process is running, p represents the process' pid, θ is the environment

(i.e., a substitution), e is the next expression that needs to be evaluated and q is the process' *mailbox*, a FIFO queue of messages that have been sent to the process. The following operations are permitted on the mailbox. Given a message v and a local mailbox q, we identify $v : q$ as a new local mailbox with the message v on top. On the other hand, by means of $q \backslash\backslash v$ we indicate the queue resulting by removing v from q, it is important to notice that v not necessarily is the oldest message in the queue.

Definition 3.4 (Distributed system)**.** A distributed system is denoted by $\Gamma; \Pi; \Omega$, where Γ, the global mailbox, is a multiset of the form *(target_process_pid,message)*, Π is a pool of running processes, denoted by an expression of the form

$$\langle node_1, p_1, (\theta_1, e_1), q_1 \rangle \mid \ldots \mid \langle node_n, p_n, (\theta_n, e_n), q_n \rangle$$

where "|" represents a commutative and associative operator. Given a global mailbox Γ we denote $\Gamma' = \Gamma \cup \{(p, v)\}$ as the new global mailbox which also stores the pair (p, v). Often, in the rest of this work we will denote a system with the expression

$$\Gamma; \langle node, p, (\theta, e), q \rangle \mid \Pi; \Omega$$

to indicate the process p, which is an arbitrary process of the pool (this is possible thanks to the associativity and commutativity of the operator "|"). Ultimately, Ω represents the set of nodes connected to the network, two operations are permitted on Ω, add a new node by means of start, we will indicate it by $\{node\} \cup \Omega$, where *node* is the new node, and reading the nodes connected together through the built-in function nodes.

As already done previously, we define the semantics by means of two relations: $\longrightarrow$ for expressions and $\hookrightarrow$ for systems. The relation $\longrightarrow$ is defined as follow:

$$\longrightarrow : \{Env, Exp\} \times \{Label\} \times \{Env, Exp\}$$

where Env is the domain of environments (i.e. substitutions) and Exp is the domain of expressions, and $Label$ represents an element of the following set:

$$\{\tau, \mathsf{send}(v_1, v_2), \mathsf{rec}(\kappa, \overline{cl_n}), \mathsf{spawn}(\kappa, a', a/n, [\overline{v_n}]), \mathsf{self}(\kappa), \mathsf{start}(\kappa, a), \mathsf{node}(\kappa), \mathsf{nodes}(\kappa)\}$$

The letter ℓ is used to range over the labels, and as one can notice, $\longrightarrow$ is now able to range also on $\mathsf{start}(\kappa, a), \mathsf{node}(\kappa), \mathsf{nodes}(\kappa)$. The set of rules that will be used to evaluate sequential expressions is still the one defined in Fig 2.4, indeed, using the same abuse of notation that has been used in [1], we moved in to the set of rules for evaluating concurrent expressions also node, nodes, and start. Fig. 3.3 shows the set of rules for evaluating concurrent expressions. Now, we will analyse the distributed aspects of these rules, the reader interested in other aspects of the rules will find a detailed explanation in Section 2.2.

$$\textit{Send1}\ \frac{\theta, e_1 \xrightarrow{\ell} \theta', e_1'}{\theta, e_1\ !\ e_2 \xrightarrow{\ell} \theta', e_1'\ !\ e_2} \qquad \textit{Send2}\ \frac{\theta, e_2 \xrightarrow{\ell} \theta', e_2'}{\theta, v_1\ !\ e_2 \xrightarrow{\ell} \theta', v_1\ !\ e_2'}$$

$$\textit{Send3}\ \frac{}{\theta, v_1\ !\ v_2 \xrightarrow{\mathsf{send}(v_1,v_2)} \theta, v_2}$$

$$\textit{Receive}\ \frac{}{\theta, \mathsf{receive}\ cl_1; ...; cl_n\ \mathsf{end} \xrightarrow{\mathsf{rec}(\kappa,\overline{cl_n})} \theta, \kappa}$$

$$\textit{Spawn1}\ \frac{\theta, e \xrightarrow{\ell} \theta', e'}{\theta, \mathsf{spawn}(e, a/n, [\overline{args_n}]) \xrightarrow{\ell} \theta', \mathsf{spawn}(e', a/n, [\overline{args_n}])}$$

$$\textit{Spawn2}\ \frac{\theta, e_i \xrightarrow{\ell} \theta', e_i' \quad i \in \{1, ..., n\}}{\theta, \mathsf{spawn}(a', a/n, [\overline{v_{1,i-1}}, e_i, \overline{e_{i+1,n}}]) \xrightarrow{\ell} \theta', \mathsf{spawn}(a', a/n, [\overline{v_{1,i-1}}, e_i', \overline{e_{i+1,n}}])}$$

$$\textit{Spawn3}\ \frac{}{\theta, \mathsf{spawn}(a', a/n, [\overline{v_n}]) \xrightarrow{\mathsf{spawn}(\kappa,a',a/n,[\overline{v_n}])} \theta, \kappa}$$

$$\textit{Self}\ \frac{}{\theta, \mathsf{self}() \xrightarrow{\mathsf{self}(\kappa)} \theta, \kappa}$$

$$\textit{Node}\ \frac{}{\theta, \mathsf{node}() \xrightarrow{\mathsf{node}(\kappa)} \theta, \kappa}$$

$$\textit{Nodes}\ \frac{}{\theta, \mathsf{nodes}() \xrightarrow{\mathsf{nodes}(\kappa)} \theta, \kappa}$$

$$\textit{Start1}\ \frac{\theta, e \xrightarrow{\ell} \theta', e'}{\theta, \mathsf{start}(e) \xrightarrow{\ell} \theta', \mathit{start}(e')}$$

$$\textit{Start2}\ \frac{}{\theta, \mathsf{start}(a) \xrightarrow{\mathsf{start}(\kappa,a)} \theta, \kappa}$$

Figure 3.2: Extended standard semantics: evaluations of concurrent expression

$$(Seq)\frac{\theta, e \xrightarrow{\tau} \theta', e'}{\Gamma; \langle node, p, (\theta, e), q\rangle \mid \Pi; \Omega \hookrightarrow \Gamma; \langle node, p, (\theta', e'), q\rangle \mid \Pi; \Omega}$$

$$(Send)\frac{\theta, e \xrightarrow{\mathsf{send}(p'', v)} \theta', e'}{\Gamma; \langle node, p, (\theta, e), q\rangle \mid \Pi; \Omega \hookrightarrow \Gamma \cup \{(p'', v)\}; \langle node, p, (\theta', e'), q\rangle \mid \Pi; \Omega}$$

$$(Receive)\frac{\theta, e \xrightarrow{\mathsf{rec}(\kappa, \overline{cl_n})} \theta', e' \quad \mathsf{matchrec}(\theta, \overline{cl_n}, q) = (\theta_i, e_i, v)}{\Gamma; \langle node, p, (\theta, e), q\rangle \mid \Pi; \Omega \hookrightarrow \Gamma; \langle node, p, (\theta'\theta_i, e'\{\kappa \mapsto e_i\}), q \backslash\backslash v\rangle \mid \Pi; \Omega}$$

$$(Spawn1)\frac{\theta, e \xrightarrow{\mathsf{spawn}(\kappa, node', a/n, [\overline{v_n}])} \theta', e' \quad p' \text{ is a fresh pid} \quad node' \in \Omega}{\begin{array}{c}\Gamma; \langle node, p, (\theta, e), q\rangle \mid \Pi; \Omega \hookrightarrow \Gamma; \langle node, p, (\theta', e'\{\kappa \mapsto p'\}), q\rangle \\ \mid \langle node', p', (id, \mathsf{apply}\ a/n(\overline{v_n}), [\,]\rangle \mid \Pi; \Omega\end{array}}$$

$$(Spawn2)\frac{\theta, e \xrightarrow{\mathsf{spawn}(\kappa, node', a/n, [\overline{v_n}])} \theta', e' \quad p' \text{ is a fresh pid} \quad node' \notin \Omega}{\Gamma; \langle node, p, (\theta, e), q\rangle \mid \Pi; \Omega \hookrightarrow \Gamma; \langle node, p, (\theta', e'\{\kappa \mapsto p'\}), q\rangle \mid \Pi; \Omega}$$

$$(Start1)\frac{\theta, e \xrightarrow{\mathsf{start}(\kappa, node')} \theta', e' \quad node' \notin \Omega}{\Gamma; \langle node, p, (\theta, e), q\rangle \mid \Pi; \Omega \hookrightarrow \Gamma; \langle node, p, (\theta', e'\{\kappa \mapsto node'\}), q\rangle \mid \Pi;\ \{node'\} \cup \Omega}$$

$$(Start2)\frac{\theta, e \xrightarrow{\mathsf{start}(\kappa, node')} \theta', e' \quad node' \in \Omega \quad err \text{ represents the error}}{\Gamma; \langle node, p, (\theta, e), q\rangle \mid \Pi; \Omega \hookrightarrow \Gamma; \langle node, p, (\theta', e'\{\kappa \mapsto err\}), q\rangle \mid \Pi; \Omega}$$

$$(Self)\frac{\theta, e \xrightarrow{\mathsf{self}(\kappa)} \theta', e'}{\Gamma; \langle node, p, (\theta, e), q\rangle \mid \Pi; \Omega \hookrightarrow \Gamma; \langle node, p, (\theta', e'\{\kappa \mapsto p\}), q\rangle \mid \Pi; \Omega}$$

$$(Node)\frac{\theta, e \xrightarrow{\mathsf{node}(\kappa)} \theta', e'}{\Gamma; \langle node, p, (\theta, e), q\rangle \mid \Pi; \Omega \hookrightarrow \Gamma; \langle node, p, (\theta', e'\{\kappa \mapsto node\}), q\rangle \mid \Pi; \Omega}$$

$$(Nodes)\frac{\theta, e \xrightarrow{\mathsf{nodes}(\kappa)} \theta', e'}{\Gamma; \langle node, p, (\theta, e), q\rangle \mid \Pi; \Omega \hookrightarrow \Gamma; \langle node, p, (\theta', e'\{\kappa \mapsto list(\Omega \setminus \{node\})\}), q\rangle \mid \Pi; \Omega}$$

$$(Sched)\frac{}{\Gamma \cup \{(p, v)\}; \langle node, p, (\theta, e), q\rangle \mid \Pi; \Omega \hookrightarrow \Gamma; \langle node, p, (\theta, e), v : q\rangle \mid \Pi; \Omega}$$

Figure 3.3: Extended standard semantics: system rules.

```
main/0 = fun () → let _ = start(counter, hot_dog_shop)
                  let C1 = spawn('counter@hot_dog_shop', cashier/0, [])
                  in let S = self()
                  in let _ = C1 ! {request, 3, hot_dog, S}
                  in receive
                      {3, hot_dog} → eat
                  end
cashier/0 = fun () → receive
                      {request, N, hot_dog, P} → P ! {N, hot_dog}
                      {request, N, chips, P} → P ! {N, chips}
                  end
```

Figure 3.4: An example of a program that starts a node

First of all, it is possible to notice that the function spawn now requires three arguments, as opposed to the spawn defined in Fig. 2.5, which had only two arguments. The additional extra-argument is required to specify in which node the spawn of the process will happen. Moreover, we can distinguish two categories of rules, rules belonging to the first category are those rules which we know *locally* to what value they will reduce, conversely, rules belonging to the second category are those rules which we do not know to what value they will reduce. Rules $Send1, Send2$, $Send3$ and $Start1$ belong to the first category, the remaining rules belong to the second one. For those rules which we do not know a value locally we return a fresh symbol κ, which will act as a future, then the system semantic will eventually bind κ to the right values.

The system rules are depicted in Fig. 3.3.
In most of the rules, we denote by $\Gamma; \langle node, p, (\theta, e), q\rangle \mid \Pi; \Omega$ an arbitrary system where we select a process, *node* indicates on which node the process is running, with the proviso that *node* always belong to Ω, p represents the unique pid of the process, (θ, e) represents the control environment, and, ultimately, q represents the queue of messages received by the process. Let us now discuss the distributed aspects of the system semantics.

Here, we have two rules $Spawn$, indeed, while before it was impossible for a spawn to fail, by adding the possibility to perform remote spawns we also have introduced the possibility for a spawn to fail. Rule $Spawn2$ represents a failed spawn, the spawn fails because the node feed as first argument does not belong to Ω, in other words it means that the node where the spawn was supposed to happen is not connected to the network.

On the contrary, rule $Spawn1$ depicts a successful spawn, and it is possible to observe that the freshly spawned process has, as a node attribute, the one that was specified to the spawn.

As one would expect, we also have two rules $Start$, one for when the start succeeds

(*Start*1), and one for when the start fails (*Start*2). In a real system, a start could fail for many reasons, to mention a few of them: the internet connection could fail, the daemon waiting for the connection could not work properly, or someone could accidentally unplug the remote machine, in our simulated environment the only reason why a start can fail is that the node is already part of the network. Indeed, since every node has to have a unique name, if someone tries to start a node that is already connected there is no viable solution but to return an error.

Finally, rules *Node* and *Nodes* respectively, bind κ to the process' node and to the network of connected nodes.
Now, we introduce the notion of concurrency for the new language which we have just introduced.

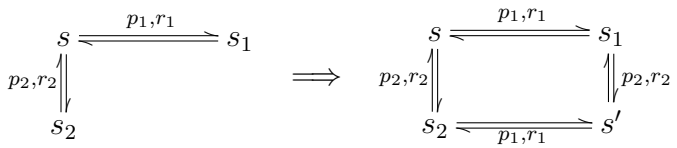
Definition 3.5 (Concurrent transitions)**.** Given two coinitial transitions, $t_1 = (s \rightleftharpoons_{p_1,r_1} s_1)$ and $t_2 = (s \rightleftharpoons_{p_2,r_2} s_2)$, we say that they are in conflict if at least one of the following conditions holds:

- they consider the same process, i.e., $p_1 = p_2$, and either $r_1 = r_2 = Sched$ or one transition applies rule *Sched* and the other transition applies rule *Receive*.
- they consider different processes, i.e., $p_1 \neq p_2$, and one rule is *Start*1, which starts a node, say n, that does not belong to the network, and the other process performs a spawn on n.
- they consider different processes, i.e., $p_1 \neq p_2$, and both start the same node that does not belong to the network yet.
- they consider different processes, i.e., $p_1 \neq p_2$, and one transition applies rule *Nodes* and the other applies rule *Start*1, to start a node that does not exists yet.

Two transitions are *concurrent* if they are *not* in conflict.

Now, we prove that this definition of concurrency makes sense.

Lemma 3.1 (Square lemma)**.** Given two coinitial concurrent transitions $t_1 = (s \rightleftharpoons_{p_1,r_1} s_1)$ and $t_2 = (s \rightleftharpoons_{p_2,r_2} s_2)$, there exist two cofinal transitions $t_2/t_1 = (s_1 \rightleftharpoons_{p_2,r_2} s')$ and $t_1/t_2 = (s_2 \rightleftharpoons_{p_1,r_1} s')$. Graphically,



Proof. We distinguish the following cases depending on the applied rules:

- Two transitions t_1 and t_2 where $r_1 \neq Sched$ and $r_2 \neq Sched$. Trivially, they apply to different processes, i.e., $p_1 \neq p_2$. There are few problematic cases, we will now analyse them:

  i) t_1 starts a node and t_2 spawns a process, if both rules consider the same node, and this one is not part of the network yet, by applying t_1 to p_2 and t_2 to p_1 we would end up in two different systems.

  ii) both transitions start a node, if both rules consider the same node by applying t_1 to p_2 and t_2 to p_1 we would end up in two different systems, one system would be the one where p_1 succeeded in the creation of the node and p_2 failed, and the other would the one where p_1 failed and p_2 succeeded.

  iii) $r_1 = Start1$ and $r_2 = Nodes$, by applying t_1 to p_2 and t_2 to p_1 we would end up in two different systems, indeed in one system the result of rule $Nodes$ applied by p_2 contains the node started by p_1, in the other, since rule $Start1$ is applied after $Nodes$, the result does not contains the node started by p_1.

  Nonetheless, i, ii, iii are not concurrent transitions, therefore, such situations cannot happen. Then, for the remaining cases, we can easily prove that by applying rule r_2 to p_1 in s_1 and rule r_1 to p_2 in s_2 we have two transitions t_1/t_2 and t_2/t_1 which produce the corresponding history items and are cofinal.

- One transition t_1 which applies rule $r_1 = Sched$ to deliver message v_1 to process $p_1 = p$, and another transition which applies a rule r_2 different from $Sched$. All cases but $r_2 = Receive$ with $p_2 = p$ are straightforward. This situation, though, cannot happen since transitions using rules $Sched$ and $Receive$ are not concurrent.

- Two transitions t_1 and t_2 with rules $r_1 = r_2 = Sched$ delivering messages v_1 and v_2, respectively. Since the transitions are concurrent, they should deliver the messages to different processes, i.e., $p_1 \neq p_2$. Therefore, we can easily prove that delivering v_2 from s_1 and v_1 from s_2 we get two cofinal transitions.

□

We remark the fact that this definition of concurrency is not unique, indeed many others exist, but we believe that this one is a good tradeoff between the complexity of the definition itself and the granularity of concurrent actions captured.

Example 3.1. In Fig. 3.5 one can observe a derivation from "apply $main/0$ ()" of the simple program described in Fig. 3.4.
In the figure is possible to observe one possible execution of the program. For clarity we label each step with the corresponding reduction rule, furthermore we underline the reduced expression. On the right one can see the value of Ω. For the sake of brevity we represented the processes' pid with m, for the processes executing main, and c, for the processes executing cashier.

3.3 A reversible semantics

In the following of this section we will introduce two new semantics, the forward reversible semantics, which will keep track of every step performed through an history item, and a backward semantics, which will tell us how and when it is possible to reach previous states of the computation, while being causally consistent. To be more precise, we will introduce two new relations, namely $\rightharpoonup$ and $\leftharpoondown$.

On the one hand, the first relation, $\rightharpoonup$, is the natural extension of the relation $\hookrightarrow$, introduced in Fig. 3.3, which follows a typical Landauer embedding, now each forward step records in the history which action has been performed and other useful informations in order to be able to undo the action in a causal-consistent way. We will refer to this relation as the *forward* (reversible) semantics.

On the other hand, we have $\leftharpoondown_{p,r,\Psi}$ which represents the *backward* reversible semantics, which is able to undo a step in a causal-consistent way. At present, we will ignore the labels p, r and Ψ, as they are there to implement a rollback operator, more details can be found in the next section. Lastly, in order to define the reversible semantics we will make use of auxiliary functions while defining the rules, each time an auxiliary function will be used an informal description will also be provided, while if the reader is interested in a more formal description (s)he can refer to Appendix A.

In Fig. 3.6 are depicted the rules of the forward semantics. As one can see, this semantics is a natural extension of the system semantics, presented in Fig.3.3, nonetheless, there are few interesting situations, due to the distributed functions that have been added, which we will now analyse.

Here, both rules *Spawn* map κ to a fresh pid, regardless of the fact that one has successfully created a new process and the other has failed, we opted for this solution because this is what actually happens in Erlang, also it is important to notice that the history kept is exactly the same in both situations.

On the contrary, rules *Start* produce two different history items, if the start succeeds ($Start1$) then an atom *succ* is included in the history, conversely, if the start fails ($Start2$) then an atom *fail* is included in the history.
Unfortunately, the choice of including such atom in the history item will narrow down our definition of concurrency, but it is necessary in order to be causally consistent. In fact, without the atom pointing out which process successfully created the node, it would be possible to broaden the definition of concurrency, but it would be impossible to be causally-consistent. As a matter of fact, we could reach a point in our system where several processes have a start item in the history with the same node and since we do not have a notion of time, it would be impossible to tell apart which one is the process which actually created the node from the rest.

Conversely, with the spawn we do not need to save such an atom because the failed spawn returns a unique pid, therefore when it comes to understanding if a spawn has

$$
\begin{array}{lll}
 & \{\,\}; & \langle n_1, \mathrm{m}, (id, \underline{\mathsf{apply}\ \mathrm{main}/0\ [\,]}), [\,]\rangle \quad ;\{n_1\} \\
\hookrightarrow_{Seq} & \{\,\}; & \langle n_1, \mathrm{m}, (id, \mathsf{let}\ \_\ = \underline{\mathsf{start}(counter, hot\_dog\_shop)}\ \mathsf{in}\ \ldots, [\,]\rangle \quad ;\{n_1\} \\
\hookrightarrow_{Start} & \{\,\}; & \langle n_1, \mathrm{m}, (id, \underline{\mathsf{let}\ \_\ = counter@hot\_dog\_shop}\ \mathsf{in}\ \ldots, [\,]\rangle \quad ;\{n_1, n_2\} \\
\hookrightarrow_{Seq} & \{\,\}; & \langle n_1, \mathrm{m}, (id, \mathsf{let}\ C\ = \underline{\mathsf{spawn}(n_2, \mathrm{cashier}/0, [\,])}\ \mathsf{in}\ \ldots, [\,]\rangle \quad ;\{n_1, n_2\} \\
\hookrightarrow_{Spawn} & \{\,\}; & \langle n_1, \mathrm{m}, (id, \mathsf{let}\ C\ = \underline{\mathsf{spawn}(n_2, \mathrm{cashier}/0, [\,])}\ \mathsf{in}\ \ldots, [\,]\rangle \\
 & & \mid \langle n_2, c, (id, \mathsf{apply}(\mathrm{cashier}/0\ [\,]), [\,]\rangle \quad ;\{n_1, n_2\} \\
\hookrightarrow_{Spawn} & \{\,\}; & \langle n_1, \mathrm{m}, (id, \mathsf{let}\ C\ = c\ \mathsf{in}\ \ldots, [\,]\rangle \\
 & & \mid \langle n_2, c, (id, \underline{\mathsf{apply}(\mathrm{cashier}/0\ [\,])}, [\,]\rangle \quad ;\{n_1, n_2\} \\
\hookrightarrow_{Seq} & \{\,\}; & \langle n_1, \mathrm{m}, (id, \underline{\mathsf{let}\ C\ = c}\ \mathsf{in}\ \ldots, [\,]\rangle \\
 & & \mid \langle n_2, c, (id, \mathsf{receive}\ \ldots), [\,]\rangle \quad ;\{n_1, n_2\} \\
\hookrightarrow_{Seq} & \{\,\}; & \langle n_1, \mathrm{m}, (\{C \mapsto c\}, \underline{\mathsf{let}\ \_\ = C\ !\ \{request, 3, hot\_dog\}}\ \mathsf{in}\ \ldots, [\,]\rangle \\
 & & \mid \langle n_2, c, (id, \mathsf{receive}\ \ldots), [\,]\rangle \quad ;\{n_1, n_2\} \\
\hookrightarrow_{Seq} & \{\,\}; & \langle n_1, \mathrm{m}, (\{C \mapsto c\}, \mathsf{let}\ \_\ = \underline{C}\ !\ \{request, 3, hot\_dog\}\ \mathsf{in}\ \ldots, [\,]\rangle \\
 & & \mid \langle n_2, c, (id, \mathsf{receive}\ \ldots), [\,]\rangle \quad ;\{n_1, n_2\} \\
\hookrightarrow_{Seq} & \{\,\}; & \langle n_1, \mathrm{m}, (\{C \mapsto c\}, \mathsf{let}\ \_\ = \underline{c\ !\ \{request, 3, hot\_dog\}}\ \mathsf{in}\ \ldots, [\,]\rangle \\
 & & \mid \langle n_2, c, (id, \mathsf{receive}\ \ldots), [\,]\rangle \quad ;\{n_1, n_2\} \\
\hookrightarrow_{Send} & \{m_1\}; & \langle n_1, \mathrm{m}, (\{C \mapsto c\}, \mathsf{let}\ \_\ = \{request, 3, hot\_dog\}\ \mathsf{in}\ \ldots, [\,]\rangle \\
 & & \mid \langle n_2, c, (id, \mathsf{receive}\ \ldots), [\,]\rangle \quad ;\{n_1, n_2\} \\
\hookrightarrow_{Sched} & \{\,\}; & \langle n_1, \mathrm{m}, (\{C \mapsto c\}, \underline{\mathsf{let}\ \_\ = \{request, 3, hot\_dog\}}\ \mathsf{in}\ \ldots, [\,]\rangle \\
 & & \mid \langle n_2, c, (id, \mathsf{receive}\ \ldots), [m_1]\rangle \quad ;\{n_1, n_2\} \\
\hookrightarrow_{Seq} & \{\,\}; & \langle n_1, \mathrm{m}, (\{C \mapsto c\}, \mathsf{receive}\ \{3, hot\_dog\}\ \rightarrow\ eat, [\,]\rangle \\
 & & \mid \langle n_2, c, (id, \underline{\mathsf{receive}\ \ldots}), [m_1]\rangle \quad ;\{n_1, n_2\} \\
\hookrightarrow_{Receive} & \{\,\}; & \langle n_1, \mathrm{m}, (\{C \mapsto c\}, \mathsf{receive}\ \{3, hot\_dog\}\ \rightarrow\ eat, [\,]\rangle \\
 & & \mid \langle n_2, c, (\{N \mapsto 3, P \mapsto m\}, \underline{P}\ !\ \{N, hot\_dog\}), [\,]\rangle \quad ;\{n_1, n_2\} \\
\hookrightarrow_{Receive} & \{\,\}; & \langle n_1, \mathrm{m}, (\{C \mapsto c\}, \mathsf{receive}\ \{3, hot\_dog\}\ \rightarrow\ eat, [\,]\rangle \\
 & & \mid \langle n_2, c, (\{N \mapsto 3, P \mapsto m\}, m\ !\ \{\underline{N}, hot\_dog\}), [\,]\rangle \quad ;\{n_1, n_2\} \\
\hookrightarrow_{Receive} & \{\,\}; & \langle n_1, \mathrm{m}, (\{C \mapsto c\}, \mathsf{receive}\ \{3, hot\_dog\}\ \rightarrow\ eat, [\,]\rangle \\
 & & \mid \langle n_2, c, (\{N \mapsto 3, P \mapsto m\}, \underline{m\ !\ \{3, hot\_dog\}}), [\,]\rangle \quad ;\{n_1, n_2\} \\
\hookrightarrow_{Send} & \{m_2\}; & \langle n_1, \mathrm{m}, (\{C \mapsto c\}, \mathsf{receive}\ \{3, hot\_dog\}\ \rightarrow\ eat, [\,]\rangle \\
 & & \mid \langle n_2, c, (\{N \mapsto 3, P \mapsto m\}, \{3, hot\_dog\}), [\,]\rangle \quad ;\{n_1, n_2\} \\
\hookrightarrow_{Sched} & \{\,\}; & \langle n_1, \mathrm{m}, (\{C \mapsto c\}, \underline{\mathsf{receive}\ \{3, hot\_dog\}\ \rightarrow\ eat}, [m_2]\rangle \\
 & & \mid \langle n_2, c, (\{N \mapsto 3, P \mapsto m\}, \{3, hot\_dog\}), [\,]\rangle \quad ;\{n_1, n_2\} \\
\hookrightarrow_{Receive} & \{\,\}; & \langle n_1, \mathrm{m}, (\{C \mapsto c\}, eat, [\,]\rangle \\
 & & \mid \langle n_2, c, (\{N \mapsto 3, P \mapsto m\}, \{3, hot\_dog\}), [\,]\rangle \quad ;\{n_1, n_2\}
\end{array}
$$

Figure 3.5: A derivation from "apply $main/0$ ()", where $m_1 = \{request, 3, hot\_dog\}$, $m_2 = \{3, hot\_dog\}$, $n_1 = cauder@debugger$ and $n_2 = counter@hot\_dog\_shop$

$$(Seq)\quad \frac{\theta, e \xrightarrow{\tau} \theta', e'}{\Gamma; \langle node, p, h, (\theta, e), q\rangle \mid \Pi; \Omega \rightharpoonup \Gamma; \langle node, p, \tau(\theta, e) : h, (\theta', e'), q\rangle \mid \Pi; \Omega}$$

$$(Send)\quad \frac{\theta, e \xrightarrow{\mathsf{send}(p'', v)} \theta', e' \quad \lambda \text{ is a fresh identifier}}{\begin{array}{c}\Gamma; \langle node, p, h, (\theta, e), q\rangle \mid \Pi; \Omega \\ \rightharpoonup \Gamma \cup \{(p'', \{\lambda, v\})\}; \langle node, p, \mathsf{send}(\theta, e, p'', \{\lambda, v\}) : h, (\theta', e'), q\rangle \mid \Pi; \Omega\end{array}}$$

$$(Receive)\quad \frac{\theta, e \xrightarrow{\mathsf{rec}(\kappa, \overline{cl_n})} \theta', e' \quad \mathsf{matchrec}(\theta, \overline{cl_n}, q) = (\theta_i, e_i, \{\lambda, v\})}{\begin{array}{c}\Gamma; \langle node, p, h, (\theta, e), q\rangle \mid \Pi; \Omega \\ \rightharpoonup \Gamma; \langle node, p, \mathsf{rec}(\theta, e, \{\lambda, v\}, q) : h, (\theta'\theta_i, e'\{\kappa \mapsto e_i\}), q \backslash\backslash \{\lambda, v\}\rangle \mid \Pi; \Omega\end{array}}$$

$$(Spawn1)\quad \frac{\theta, e \xrightarrow{\mathsf{spawn}(\kappa, node', a/n, [\overline{v_n}])} \theta', e' \quad p' \text{ is a fresh pid} \quad node' \in \Omega}{\begin{array}{c}\Gamma; \langle node, p, h, (\theta, e), q\rangle \mid \Pi; \Omega \rightharpoonup \Gamma; \langle node, p, \mathsf{spawn}(\theta, e, node', p') : h, (\theta', e'\{\kappa \mapsto p'\}), q\rangle \\ \mid \langle node', p', [\,], (id, \mathsf{apply}\ a/n(\overline{v_n})), [\,]\rangle \mid \Pi; \Omega\end{array}}$$

$$(Spawn2)\quad \frac{\theta, e \xrightarrow{\mathsf{spawn}(\kappa, node', a/n, [\overline{v_n}])} \theta', e' \quad p' \text{ is a fresh pid} \quad node' \notin \Omega}{\Gamma; \langle node, p, h, (\theta, e), q\rangle \mid \Pi; \Omega \rightharpoonup \Gamma; \langle node, p, \mathsf{spawn}(\theta, e, node', p') : h, (\theta', e'\{\kappa \mapsto p'\}), q\rangle \mid \Pi; \Omega}$$

$$(Start1)\quad \frac{\theta, e \xrightarrow{\mathsf{start}(\kappa, node')} \theta', e' \quad node' \notin \Omega}{\begin{array}{c}\Gamma; \langle node, p, h, (\theta, e), q\rangle \mid \Pi; \Omega \rightharpoonup \Gamma; \langle node, p, \mathsf{start}(\theta, e, succ, node') : h, (\theta', e'\{\kappa \mapsto node'\}), q\rangle \\ \mid \Pi; \{node'\} \cup \Omega\end{array}}$$

$$(Start2)\quad \frac{\theta, e \xrightarrow{\mathsf{start}(\kappa, node')} \theta', e' \quad node' \in \Omega \quad err \text{ represents the error}}{\Gamma; \langle node, p, h, (\theta, e), q\rangle \mid \Pi; \Omega \rightharpoonup \Gamma; \langle node, p, \mathsf{start}(\theta, e, fail, node') : h, (\theta', e'\{\kappa \mapsto err\}), q\rangle \mid \Pi; \Omega}$$

$$(Self)\quad \frac{\theta, e \xrightarrow{\mathsf{self}(\kappa)} \theta', e'}{\Gamma; \langle node, p, h, (\theta, e), q\rangle \mid \Pi; \Omega \rightharpoonup \Gamma; \langle node, p, \mathsf{self}(\theta, e) : h, (\theta', e'\{\kappa \mapsto p\}), q\rangle \mid \Pi; \Omega}$$

$$(Node)\quad \frac{\theta, e \xrightarrow{\mathsf{node}(\kappa)} \theta', e'}{\Gamma; \langle node, p, h, (\theta, e), q\rangle \mid \Pi; \Omega \rightharpoonup \Gamma; \langle node, p, \mathsf{node}(\theta, e) : h, (\theta', e'\{\kappa \mapsto node\}), q\rangle \mid \Pi; \Omega}$$

$$(Nodes)\quad \frac{\theta, e \xrightarrow{\mathsf{nodes}(\kappa)} \theta', e'}{\begin{array}{c}\Gamma; \langle node, p, h, (\theta, e), q\rangle \mid \Pi; \Omega \\ \rightharpoonup \Gamma; \langle node, p, \mathsf{nodes}(\theta, e, \Omega) : h, (\theta', e'\{\kappa \mapsto list(\Omega \setminus \{node\})\}), q\rangle \mid \Pi; \Omega\end{array}}$$

$$(Sched)\quad \frac{}{\Gamma \cup \{(p, \{\lambda, v\})\}; \langle node, p, h, (\theta, e), q\rangle \mid \Pi; \Omega \rightharpoonup \Gamma; \langle node, p, h, (\theta, e), \{\lambda, v\} : q\rangle \mid \Pi; \Omega}$$

Figure 3.6: Extended forward reversible semantics

failed or not, it is enough to check if the pid in the history item matches any of the pid in Π, if not it was a failed spawn, otherwise it was a successful spawn.

Let us now move our focus on the rules of the backward semantics, which are depicted in Fig. 3.7. As usual, we will now discuss some interesting details and tricky situations. First of all, just by observing the rules one could notice that some of them make use of auxiliary functions and some do not. The reason why is that, for those rules which use auxiliary functions, before undoing the step some criteria need to be met, otherwise we would end up in a *non-consistent* state.

Now, let us start analysing the first rule which makes use of auxiliary functions, namely rule $\overline{Spawn2}$, by introducing an example.

Example 3.2. When a process, say p_1, tries to spawn another process, say p_2, and fails in our system that can only mean that the node, say *node*, where p_1 tried to spawn p_2 was not part of Ω. Now, if another process, say p_3, starts *node*, and we undo the failed spawn, ignoring the auxiliary function, and then we redo it this time it will not fail, because now *node* is part of Ω, breaking the loop lemma (Lemma 3.2).

Here, Example 3.2 gives us an idea of why it is necessary, in order to maintain causal consistency, to make sure that the node which caused the fail is still not part of Ω. The other condition, $exists(p, \Pi) = false$, is necessary in order to distinguish if the spawn failed or not, indeed, as already discussed, thanks to the unicity of the pids it is not necessary to save in the history the result of the spawn.

Again on the one hand, this implies that we will have a broader definition of concurrency, indeed, the definition of concurrency depends on the definition of the history, but on the other hand, it is more computationally expensive. In fact, adding an atom to the history would imply a cost $O(1)$ to determine if the spawn has failed or not, without this information we have to scan every process of the system and check if the pid in the history tag matches one of them, therefore we have a computational cost of $O(n)$, where n represents the number of process in Π.

During this work, every time we had to face such a decision we have always opted for a broader definition of concurrency against lower computational costs.

Now, let us move our focus to rule $\overline{Start1}$; this rule is the one undoing the successful start of a node, therefore before undoing this step it is crucial to make sure that every action which depends on this one has been undone beforehand. Here, three auxiliary functions have been used, the first one makes sure that there are no more processes running on the node, the second one makes sure that no one has performed a *read* of Ω by means of rule *Nodes*, and, finally, the third function makes sure that no other processes tried a start of the same node. It is important to notice that the first check could have been avoided with the assumption that the only way to introduce in a program a node name is through the evaluation of a start, however, that would have implied that there would be no way to hard-code the name of a node in a program, therefore we decided

$$(\overline{Seq}) \quad \Gamma; \langle node, p, \tau(\theta, e) : h, (\theta', e'), q\rangle \mid \Pi; \Omega \leftharpoondown_{p,\mathsf{seq},\{s\}} \Gamma; \langle node, p, h, (\theta, e), q\rangle \mid \Pi; \Omega$$

$$(\overline{Send}) \quad \begin{array}{c} \Gamma \cup \{(p'', \{\lambda, v\})\}; \langle node, p, \mathsf{send}(\theta, e, p'', \{\lambda, v\}) : h, (\theta', e'), q\rangle \mid \Pi; \Omega \\ \leftharpoondown_{p,\mathsf{send}(\lambda),\{s,\lambda^{\Uparrow}\}} \Gamma; \langle node, p, h, (\theta, e), q\rangle \mid \Pi; \Omega \end{array}$$

$$(\overline{Receive}) \quad \begin{array}{c} \Gamma; \langle node, p, \mathsf{rec}(\theta, e, \{\lambda, v\}, q) : h, (\theta', e'), q \backslash\backslash \{\lambda, v\}\rangle \mid \Pi; \Omega \\ \leftharpoondown_{p,\mathsf{rec}(\lambda),\{s,\lambda^{\Downarrow}\}} \Gamma; \langle node, p, h, (\theta, e), q\rangle \mid \Pi; \Omega \end{array}$$

$$(\overline{Spawn1}) \quad \begin{array}{c} \Gamma; \langle node, p, \mathsf{spawn}(\theta, e, node', p') : h, (\theta', e'), q\rangle \mid \langle node', p, [\,], (id, e''), [\,]\rangle \mid \Pi; \Omega \\ \leftharpoondown_{p,\mathsf{spawn}(p'),\{s,sp_{p'}\}} \Gamma; \langle node, p, h, (\theta, e), q\rangle \mid \Pi; \Omega \end{array}$$

$$(\overline{Spawn2}) \quad \begin{array}{c} \Gamma; \langle node, p, \mathsf{spawn}(\theta, e, node', p') : h, (\theta', e'), q\rangle \mid \Pi; \Omega \\ \leftharpoondown_{p,\mathsf{spawn}(p'),\{s,sp_{p'}\}} \Gamma; \langle node, p, h, (\theta, e), q\rangle \mid \Pi; \Omega \\ \text{if } exists(p', \Pi) = false \ \wedge \ node' \notin \Omega \end{array}$$

$$(\overline{Start1}) \quad \begin{array}{c} \Gamma; \langle node, p, \mathsf{start}(\theta, e, succ, node') : h, (\theta', e'), q\rangle \mid \Pi; \Omega \cup \{node'\} \\ \leftharpoondown_{p,\mathsf{start}(node'),\{s,st_{node'}\}} \Gamma; \langle node, p, h, (\theta, e), q\rangle \mid \Pi; \Omega \\ \text{if } procs(node', \Pi) = [\,] \wedge reads(node', \Pi) = [\,] \wedge tried\_starts(node', \Pi) = [\,] \end{array}$$

$$(\overline{Start2}) \quad \begin{array}{c} \Gamma; \langle node, p, \mathsf{start}(\theta, e, fail, node') : h, (\theta', e'), q\rangle \mid \Pi; \Omega \\ \leftharpoondown_{p,\mathsf{start}(node'),\{s\}} \Gamma; \langle node, p, h, (\theta, e), q\rangle \mid \Pi; \Omega \end{array}$$

$$(\overline{Self}) \quad \Gamma; \langle node, p, \mathsf{self}(\theta, e) : h, (\theta', e'), q\rangle \mid \Pi; \Omega \leftharpoondown_{p,\mathsf{self},\{s\}} \Gamma; \langle node, p, h, (\theta, e), q\rangle \mid \Pi; \Omega$$

$$(\overline{Node}) \quad \Gamma; \langle node, p, \mathsf{node}(\theta, e) : h, (\theta', e'), q\rangle \mid \Pi; \Omega \leftharpoondown_{p,\mathsf{node},\{s\}} \Gamma; \langle node, p, h, (\theta, e), q\rangle \mid \Pi; \Omega$$

$$(\overline{Nodes}) \quad \begin{array}{c} \Gamma; \langle node, p, \mathsf{nodes}(\theta, e, \Omega') : h, (\theta', e'), q\rangle \mid \Pi; \Omega \leftharpoondown_{p,\mathsf{nodes},\{s\}} \Gamma; \langle node, p, h, (\theta, e), q\rangle \mid \Pi; \Omega \\ \text{if } \Omega = \Omega' \end{array}$$

$$(\overline{Sched}) \quad \begin{array}{c} \Gamma; \langle node, p, h, (\theta, e), \{\lambda, v\} : q\rangle \mid \Pi; \Omega \\ \leftharpoondown_{p,\mathsf{sched}(\lambda),\{s,\lambda^{sched}\}} \Gamma \cup \{(p, \{\lambda, v\})\}; \langle node, p, h, (\theta, e), q\rangle \mid \Pi; \Omega \\ \text{if the topmost } \mathsf{rec}(...) \text{ item in h (if any) has the} \\ \text{form } \mathsf{rec}(\theta', e', \{\lambda', v'\}, q') \text{ with } q' \backslash\backslash \{\lambda', v'\} \neq \{\lambda, v\} : q \end{array}$$

Figure 3.7: Extended backward reversible semantics

$$
\begin{aligned}
\mathrm{main}/0 = \mathsf{fun}\ () \rightarrow\ & \mathsf{let}\ R1 = \mathsf{spawn}(n_1, \mathrm{racer}/0, [\,]) \\
& \mathsf{in\ let}\ R2 = \mathsf{spawn}(n_1, \mathrm{racer}/0, [\,]) \\
& \mathsf{in\ let}\ \_ = R1\ !\ go \\
& \mathsf{in}\ R2\ !\ go \\
\mathrm{racer}/0 = \mathsf{fun}\ () \rightarrow\ & \mathsf{let}\ R = \mathsf{receive} \\
& \quad go \rightarrow \mathsf{start}(another, node) \\
& \mathsf{end} \\
& \mathsf{in\ case}\ R\ \mathsf{of} \\
& \quad \{ok, \_\} \rightarrow winner \\
& \quad \{error, \_\} \rightarrow loser \\
& \mathsf{end}
\end{aligned}
$$

Figure 3.8: An example of a program where two processes race to start a node

again for the solution that would bring us as close as possible to a real Erlang system. Now, through the following example we will clarify why we need also to make use of the second auxiliary function.

Example 3.3. Let us consider a system composed by two processes, p_1 and p_2, running on the same node, now $p1$ performs a start, then p_2 performs a nodes. In this situation, if we undo the start performed by p_1 and then we also undo the nodes made by p_2 and then we redo the nodes we would enter a new state, this because now only one node is part of Ω and not two.

For this reason, before undoing a start we also need to ensure that no process has read the node that we are about to undo, this also explains why while performing *Nodes* we save a copy of the current Ω.

Similarly, the third function checks that no other process tried to perform a start and failed because the node was already part of Ω, because, again, if we undo the successful start, then we undo the failed start and then we redo it will succeed instead of failing, as one would expect if the system was causally-consistent. A clever way of performing these three checks would cost $O(nm)$, where n represents the number of processes in Π and m represents the length of the longest process' history.

Lastly, we have rule $\overline{Nodes}$, where, dually to rule $\overline{Start1}$, we have to check that new nodes have not been added to Ω before undoing a *Nodes*. This check can be easily accomplished by controlling that the Ω' available in the history item and the Ω of the system are exactly the same.

Example 3.4. In Fig. 3.9 it is possible to observe a possible forward derivation of the program depicted in Fig. 3.8. For the sake of brevity we omitted non-relevant arguments in the histories by denoting them with the anonymous variable "_", the same has been done for the environment. Then, instead of showing the full expression for each process

we show $C[e]$, where C represents the context and e represents the redex which need to be evaluated, also we omitted every applications of rule Seq.

Then, in Fig. 3.10 it is possible to observe a possible backward derivation of the final system shown in Fig. 3.9. The derivation follows the same conventions and rules as above.

Finally, we stress that the backward derivation does not undo the steps performed in the same order as they were originally performed in the forward derivation, nonetheless causal consistency is maintained.

3.3.1 Properties of the extended uncontrolled reversible semantics

In the following of this section we will prove several properties of our extended reversible semantics, including its *causal consistency*. In this part we will use the same rules and conventions that we have used in Sub-Section 2.3.1.

We begin by providing the definition of *initial* distributed system.

Definition 3.6. A distributed system is *initial* if it is composed by a single process, and this process has an empty history and an empty queue; furthermore the global mailbox is empty and Ω contains only one node, i.e., the node of the only existent process. A system s is reachable if there exists an initial system s_0 and a derivation $s_0 \rightleftharpoons^* s$ using the rules corresponding to a given program.

The next lemma proves that every forward (resp. backward) transition can be undone by a correspondent backward (resp. forward) transition.

Lemma 3.2 (Loop Lemma)**.** For every pair of systems, s_1 and s_2, we have $s_1 \rightharpoonup_{p,r,k} s_2$ iff $s_2 \rightharpoonup_{p,\overline{r},k} s_1$.

Proof. The proof that a forward transition can be undone follows by rule inspection. The other direction relies on the restriction to reachable systems: consider the process undoing the action. Since the system is reachable, restoring the memory item would put us back in a state where the undone action can be performed again (if the system would not be reachable the memory item would be arbitrary, hence there would not be such a guarantee), as desired. Again, this can be proved by rule inspection. □

Now, we introduce the definition of concurrency, which tell us what actions can be switched without changing the semantics of the computation.

Definition 3.7 (Concurrent transitions)**.** Given two coinitial transitions, $t_1 = (s \rightleftharpoons_{p_1,r_1,k_1} s_1)$ and $t_2 = (s \rightleftharpoons_{p_2,r_2,k_2} s_2)$, we say that they are in conflict if at least one of the following conditions holds:

$$
\begin{array}{lll}
 & \{\,\}; & \langle n_1, \mathrm{p}, [\,], (id, C[\underline{\mathsf{apply\ main/0}\ [\,]}]), [\,]\rangle \quad ; \{n_1\} \\
\rightharpoonup_{*} & \{\,\}; & \langle n_1, \mathrm{p}, [\,], (id, C[\underline{\mathsf{spawn}(n_1, \mathrm{main}/0, [\,])}], [\,]\rangle \quad ; \{n_1\} \\
\rightharpoonup_{Spawn} & \{\,\}; & \langle n_1, \mathrm{p}, [\mathsf{spawn}(\_, \_, n_1, \mathrm{r}_1)], (id, C[\underline{\mathsf{spawn}(n_1, \mathrm{racer}/0, [\,])}], [\,]\rangle \\
 & & \mid \langle n_1, \mathrm{r}_1, [\,], (id, C[\mathsf{receive}\ go \rightarrow \ldots]), [\,]\rangle \quad ; \{n_1\} \\
\rightharpoonup_{Spawn} & \{\,\}; & \langle n_1, \mathrm{p}, [\mathsf{spawn}(\_, \_, n_1, \mathrm{r}_1), \mathsf{spawn}(\_, \_, n_1, \mathrm{r}_2)], (\_, C[\underline{\mathrm{r}_1\ !\ go}], [\,]\rangle \\
 & & \mid \langle n_1, \mathrm{r}_1, [\,], (id, C[\mathsf{receive}\ go \rightarrow \ldots]), [\,]\rangle \\
 & & \mid \langle n_1, \mathrm{r}_2, [\,], (id, C[\mathsf{receive}\ go \rightarrow \ldots]), [\,]\rangle \quad ; \{n_1\} \\
\rightharpoonup_{Send} & \{(\mathrm{r}_1, m_1)\}; & \langle n_1, \mathrm{p}, [\mathsf{spawn}(\_, \_, n_1, \mathrm{r}_1), \mathsf{spawn}(\_, \_, n_1, \mathrm{r}_2), \mathsf{send}(\_, \_, \mathrm{r}_1, m_1)], \\
 & & (\_, C[\mathrm{r}_2\ !\ go], [\,]\rangle \mid \langle n_1, \mathrm{r}_1, [\,], (id, C[\mathsf{receive}\ go \rightarrow \ldots]), [\,]\rangle \\
 & & \mid \langle n_1, \mathrm{r}_2, [\,], (id, C[\mathsf{receive}\ go \rightarrow \ldots]), [\,]\rangle \quad ; \{n_1\} \\
\rightharpoonup_{Sched} & \{\,\}; & \langle n_1, \mathrm{p}, [\mathsf{spawn}(\_, \_, n_1, \mathrm{r}_1), \mathsf{spawn}(\_, \_, n_1, \mathrm{r}_2), \mathsf{send}(\_, \_, \mathrm{r}_1, m_1)], \\
 & & (\_, C[\underline{\mathrm{r}_2\ !\ go}], [\,]\rangle \mid \langle n_1, \mathrm{r}_1, [\,], (id, C[\mathsf{receive}\ go \rightarrow \ldots]), [m_1]\rangle \\
 & & \mid \langle n_1, \mathrm{r}_2, [\,], (id, C[\mathsf{receive}\ go \rightarrow \ldots]), [\,]\rangle \quad ; \{n_1\} \\
\rightharpoonup_{Send} & \{(\mathrm{r}_2, m_2)\}; & \langle n_1, \mathrm{p}, [\mathsf{spawn}(\_, \_, n_1, \mathrm{r}_1), \mathsf{spawn}(\_, \_, n_1, \mathrm{r}_2), \mathsf{send}(\_, \_, \mathrm{r}_1, m_1), \\
 & & \mathsf{send}(\_, \_, \mathrm{r}_2, m_2)], (\_, C[go], [\,]\rangle \mid \langle n_1, \mathrm{r}_1, [\,], (id, C[\mathsf{receive}\ go \rightarrow \ldots]), \\
 & & [m_1]\rangle \mid \langle n_1, \mathrm{r}_2, [\,], (id, C[\mathsf{receive}\ go \rightarrow \ldots]), [\,]\rangle \quad ; \{n_1\} \\
\rightharpoonup_{Sched} & \{\,\}; & \langle n_1, \mathrm{p}, [\mathsf{spawn}(\_, \_, n_1, \mathrm{r}_1), \mathsf{spawn}(\_, \_, n_1, \mathrm{r}_2), \mathsf{send}(\_, \_, \mathrm{r}_1, m_1), \\
 & & \mathsf{send}(\_, \_, \mathrm{r}_2, m_2)], (\_, C[go], [\,]\rangle \mid \langle n_1, \mathrm{r}_1, [\,], (id, C[\mathsf{receive}\ go \rightarrow \ldots]), \\
 & & [m_1]\rangle \mid \langle n_1, \mathrm{r}_2, [\,], (id, C[\underline{\mathsf{receive}\ go \rightarrow \ldots}]), [m_2]\rangle \quad ; \{n_1\} \\
\rightharpoonup_{Receive} & \{\,\}; & \langle n_1, \mathrm{p}, [\mathsf{spawn}(\_, \_, n_1, \mathrm{r}_1), \mathsf{spawn}(\_, \_, n_1, \mathrm{r}_2), \mathsf{send}(\_, \_, \mathrm{r}_1, m_1), \\
 & & \mathsf{send}(\_, \_, \mathrm{r}_2, m_2)], (\_, C[go], [\,]\rangle \mid \langle n_1, \mathrm{r}_1, [\,], (id, C[\mathsf{receive}\ go \rightarrow \ldots]), \\
 & & [m_1]\rangle \mid \langle n_1, \mathrm{r}_2, [\mathsf{rec}(\_, \_, m_1, [m_1])], (id, C[\underline{\mathsf{start}(another, node)}]), [\,]\rangle \quad ; \{n_1\} \\
\rightharpoonup_{Start1} & \{\,\}; & \langle n_1, \mathrm{p}, [\mathsf{spawn}(\_, \_, n_1, \mathrm{r}_1), \mathsf{spawn}(\_, \_, n_1, \mathrm{r}_2), \mathsf{send}(\_, \_, \mathrm{r}_1, m_1), \\
 & & \mathsf{send}(\_, \_, \mathrm{r}_2, m_2)], (\_, C[go], [\,]\rangle \mid \langle n_1, \mathrm{r}_1, [\,], (id, C[\underline{\mathsf{receive}\ go \rightarrow \ldots}]), \\
 & & [m_1]\rangle \mid \langle n_1, \mathrm{r}_2, [\mathsf{rec}(\_, \_, m_1, [m_1]), \mathsf{start}(\_, \_, succ, n_2)], \\
 & & (id, C[winner]), [\,]\rangle \quad ; \{n_1, n_2\} \\
\rightharpoonup_{Receive} & \{\,\}; & \langle n_1, \mathrm{p}, [\mathsf{spawn}(\_, \_, n_1, \mathrm{r}_1), \mathsf{spawn}(\_, \_, n_1, \mathrm{r}_2), \mathsf{send}(\_, \_, \mathrm{r}_1, m_1), \\
 & & \mathsf{send}(\_, \_, \mathrm{r}_2, m_2)], (\_, C[go], [\,]\rangle \mid \langle n_1, \mathrm{r}_1, [\mathsf{rec}(\_, \_, m_2, [m_2])], \\
 & & (id, C[\underline{\mathsf{start}(another, node)}]), [\,]\rangle \mid \langle n_1, \mathrm{r}_2, [\mathsf{rec}(\_, \_, m_1, [m_1]), \\
 & & \mathsf{start}(\_, \_, succ, n_2)], (id, C[winner]), [\,]\rangle \quad ; \{n_1, n_2\} \\
\rightharpoonup_{Start2} & \{\,\}; & \langle n_1, \mathrm{p}, [\mathsf{spawn}(\_, \_, n_1, \mathrm{r}_1), \mathsf{spawn}(\_, \_, n_1, \mathrm{r}_2), \mathsf{send}(\_, \_, \mathrm{r}_1, m_1), \\
 & & \mathsf{send}(\_, \_, \mathrm{r}_2, m_2)], (\_, C[go], [\,]\rangle \mid \langle n_1, \mathrm{r}_1, [\mathsf{rec}(\_, \_, m_2, [m_2]), \\
 & & \mathsf{start}(\_, \_, fail, n_2)], (id, C[loser]), [\,]\rangle \mid \langle n_1, \mathrm{r}_2, [\mathsf{rec}(\_, \_, m_1, [m_1]), \\
 & & \mathsf{start}(\_, \_, succ, n_2)], (id, C[winner]), [\,]\rangle \quad ; \{n_1, n_2\}
\end{array}
$$

Figure 3.9: A possible forward derivation for the program depicted in Fig. 3.8, where $n_1 = cauder@debugger$, $n_2 = another@node$, $m_1 = \{1, \{go\}\}$ and $m_2 = \{2, \{go\}\}$

$$
\begin{array}{lll}
 & \{\,\}; & \langle n_1, \mathrm{p}, [\mathsf{spawn}(\_,\_,n_1,\mathrm{r}_1), \mathsf{spawn}(\_,\_,n_1,\mathrm{r}_2),\\
 & & \mathsf{send}(\_,\_,\mathrm{r}_1,m_1), \mathsf{send}(\_,\_,\mathrm{r}_2,m_2)], (\_, C[go], [\,])\rangle\\
 & & \mid \langle n_1, \mathrm{r}_1, [\mathsf{rec}(\_,\_,m_2,[m_2]), \underline{\mathsf{start}(\_,\_,\mathit{fail},n_2)}],\\
 & & (id, C[loser]), [\,])\rangle \mid \langle n_1, \mathrm{r}_2, [\mathsf{rec}(\_,\_,m_1,[m_1]),\\
 & & (\mathsf{start}(\_,\_,succ,n_2)], (id, C[winner]), [\,])\rangle & ;\{n_1,n_2\}\\
\leftharpoondown\overline{Start2} & \{\,\}; & \langle n_1, \mathrm{p}, [\mathsf{spawn}(\_,\_,n_1,\mathrm{r}_1), \mathsf{spawn}(\_,\_,n_1,\mathrm{r}_2),\\
 & & \mathsf{send}(\_,\_,\mathrm{r}_1,m_1), \mathsf{send}(\_,\_,\mathrm{r}_2,m_2)], (\_, C[go], [\,])\rangle\\
 & & \mid \langle n_1, \mathrm{r}_1, [\underline{\mathsf{rec}(\_,\_,m_2,[m_2])}], (id, C[\mathsf{start}(another,node)]), [\,])\rangle\\
 & & \mid \langle n_1, \mathrm{r}_2, [\underline{\mathsf{rec}(\_,\_,m_1,[m_1])},\\
 & & \mathsf{start}(\_,\_,succ,n_2)], (id, C[winner]), [\,])\rangle & ;\{n_1,n_2\}\\
\leftharpoondown\overline{Receive} & \{\,\}; & \langle n_1, \mathrm{p}, [\mathsf{spawn}(\_,\_,n_1,\mathrm{r}_1), \mathsf{spawn}(\_,\_,n_1,\mathrm{r}_2),\\
 & & \mathsf{send}(\_,\_,\mathrm{r}_1,m_1), \mathsf{send}(\_,\_,\mathrm{r}_2,m_2)], (\_, C[go], [\,])\rangle\\
 & & \mid \langle n_1, \mathrm{r}_2, [\mathsf{rec}(\_,\_,m_1,[m_1]), \underline{\mathsf{start}(\_,\_,succ,n_2)}],\\
 & & (id, C[winner]), [\,])\rangle & ;\{n_1,n_2\}\\
\leftharpoondown\overline{Start1} & \{\,\}; & \langle n_1, \mathrm{p}, [\mathsf{spawn}(\_,\_,n_1,\mathrm{r}_1), \mathsf{spawn}(\_,\_,n_1,\mathrm{r}_2),\\
 & & \mathsf{send}(\_,\_,\mathrm{r}_1,m_1), \mathsf{send}(\_,\_,\mathrm{r}_2,m_2)], (\_, C[go], [\,])\rangle\\
 & & \mid \langle n_1, \mathrm{r}_1, [\,], (id, C[\mathsf{receive}\ go \to \ldots]), [m_1]\rangle\\
 & & \mid \langle n_1, \mathrm{r}_2, [\underline{\mathsf{rec}(\_,\_,m_1,[m_1])}], (id, C[\mathsf{start}(another,node)]), [\,])\rangle & ;\{n_1\}\\
\leftharpoondown\overline{Receive} & \{\,\}; & \langle n_1, \mathrm{p}, [\mathsf{spawn}(\_,\_,n_1,\mathrm{r}_1), \mathsf{spawn}(\_,\_,n_1,\mathrm{r}_2),\\
 & & \mathsf{send}(\_,\_,\mathrm{r}_1,m_1), \mathsf{send}(\_,\_,\mathrm{r}_2,m_2)], (\_, C[go], [\,])\rangle\\
 & & \mid \langle n_1, \mathrm{r}_2, [\,], (id, C[\mathsf{receive}\ go \to \ldots]), [m_2]\rangle & ;\{n_1\}\\
\leftharpoondown\overline{Sched} & \{(\mathrm{r}_2, m_2)\}; & \langle n_1, \mathrm{p}, \mathsf{send}(\_,\_,\mathrm{r}_1,m_1), \mathsf{send}(\_,\_,\mathrm{r}_2,m_2)], (\_, C[go], [\,])\rangle\\
 & & \mid \langle n_1, \mathrm{r}_1, [\,], (id, C[\mathsf{receive}\ go \to \ldots]), [m_1]\rangle\\
 & & \mid \langle n_1, \mathrm{r}_2, [\,], (id, C[\mathsf{receive}\ go \to \ldots]), [\,])\rangle & ;\{n_1\}\\
\leftharpoondown\overline{Sched} & \{(\mathrm{r}_1, m_1), (\mathrm{r}_2, m_2)\}; & \langle n_1, \mathrm{p}, [\mathsf{spawn}(\_,\_,n_1,\mathrm{r}_1), \mathsf{spawn}(\_,\_,n_1,\mathrm{r}_2),\\
 & & \mathsf{send}(\_,\_,\mathrm{r}_1,m_1), \underline{\mathsf{send}(\_,\_,\mathrm{r}_2,m_2)}], (\_, C[go], [\,])\rangle\\
 & & \mid \langle n_1, \mathrm{r}_1, [\,], (id, C[\mathsf{receive}\ go \to \ldots]), [m_1]\rangle\\
 & & \mid \langle n_1, \mathrm{r}_2, [\,], (id, C[\mathsf{receive}\ go \to \ldots]), [\,])\rangle & ;\{n_1\}\\
\leftharpoondown\overline{Send} & \{(\mathrm{r}_1, m_1)\}; & \langle n_1, \mathrm{p}, [\mathsf{spawn}(\_,\_,n_1,\mathrm{r}_1), \mathsf{spawn}(\_,\_,n_1,\mathrm{r}_2),\\
 & & \underline{\mathsf{send}(\_,\_,\mathrm{r}_1,m_1)}], (\_, C[\mathrm{r}_2\ !\ go], [\,])\rangle\\
 & & \mid \langle n_1, \mathrm{r}_1, [\,], (id, C[\mathsf{receive}\ go \to \ldots]), [m_1]\rangle\\
 & & \mid \langle n_1, \mathrm{r}_2, [\,], (id, C[\mathsf{receive}\ go \to \ldots]), [\,])\rangle & ;\{n_1\}\\
\leftharpoondown\overline{Send} & \{\,\}; & \langle n_1, \mathrm{p}, [\mathsf{spawn}(\_,\_,n_1,\mathrm{r}_1), \underline{\mathsf{spawn}(\_,\_,n_1,\mathrm{r}_2)}],\\
 & & (\_, C[\mathrm{r}_1\ !\ go], [\,])\rangle \mid \langle n_1, \mathrm{r}_1, [\,], (id, C[\mathsf{receive}\ go \to \ldots]), [m_1]\rangle\\
 & & \mid \langle n_1, \mathrm{r}_2, [\,], (id, C[\mathsf{receive}\ go \to \ldots]), [\,])\rangle & ;\{n_1\}\\
\leftharpoondown\overline{Spawn} & \{\,\}; & \langle n_1, \mathrm{p}, [\underline{\mathsf{spawn}(\_,\_,n_1,\mathrm{r}_1)}], (\_, C[\mathsf{spawn}(n_1, \mathrm{racer}/0, [\,])], [\,])\rangle\\
 & & \mid \langle n_1, \mathrm{r}_1, [\,], (id, C[\mathsf{receive}\ go \to \ldots]), [m_1]\rangle\\
\leftharpoondown\overline{Spawn} & \{\,\}; & \langle n_1, \mathrm{p}, [\,], (\_, C[\mathsf{spawn}(n_1, \mathrm{racer}/0, [\,])], [\,])\rangle
\end{array}
$$

Figure 3.10: A possible backward derivation for the program depicted in Fig. 3.8, where $n_1 = cauder@debugger$, $n_2 = another@node$, $m_1 = \{1, \{go\}\}$ and $m_2 = \{2, \{go\}\}$

(a) **Both transitions are forward**

- they consider the same process, i.e., $p_1 = p_2$, and either $r_1 = r_2 = Sched$ or one transition applies rule $Sched$ and the other transition applies rule $Receive$.
- they consider different processes, i.e., $p_1 \neq p_2$, and one rule is $Start1$, which starts a node, say n, that does not belong to the network, and the other process performs a spawn on n.
- they consider different processes, i.e., $p_1 \neq p_2$, and both start the same node that does not belong to the network yet.
- they consider different processes, i.e., $p_1 \neq p_2$, and one transition applies rule $Nodes$ and the other applies rule $Start1$, to start a node that does not exists yet.

(b) **One transition is forward and the other is backward**

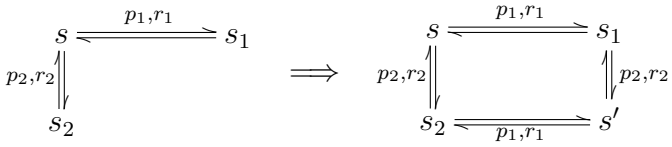
- one is a **forward** transition that applies to a process p, say $p_1 = p$, and the other one is a **backward** transition that undoes the creation of p, i.e., $p_2 \neq p, r_2 = \overline{Spawn1}$ and $k_2 = \mathsf{spawn}(\theta, e, p)$ for some control (θ, e);
- one is a **forward** transition that delivers a message $\{\lambda, v\}$ to a process p, say $p_1 = p$, $r_1 = Sched$ and $k_1 = \mathsf{sched}(\{\lambda, v\})$, and the other one is a **backward** transition that undoes the sending $\{\lambda, v\}$ to p, i.e., $r_2 = Send$ and $k_2 = \mathsf{send}(\theta, e, p, \{\lambda, v\})$ for some control (θ, e);
- one is a **forward** transition and the other one is a **backward** transition such that $p_1 = p_2$ and either
  - i) both applied rules are different from both $Sched$ and $\overline{Sched}$, i.e., $\{r_1, r_2\} \cap \{Sched, \overline{Sched}\} = \emptyset$
  - ii) one rule is $Sched$ and the other one is $\overline{Sched}$
  - iii) one rule is $Sched$ and the other one is $\overline{Receive}$
  - iv) one rule is $\overline{Sched}$ and the other one is $Receive$
- one is a **forward** transition that spawns a process on $node$, and the other one is a backward transition that undoes the creation of $node$, i.e., $p_2 \neq p, r_2 = \overline{Start1}$ and $k_2 = \mathsf{start}(\theta, e, succ, node)$, for some control (θ, e)
- one is a **forward** transition that starts $node$, i.e., $r_1 = Start1$ and the other is a **backward** transition that undoes a (failed) spawn of a process in $node$, i.e., $r_2 = \overline{Spawn2}$ and $k_2 = \mathsf{spawn}(\theta, e, node, p'')$, for some control (θ, e).
- one is a **forward** transition that applies rule $Nodes$, i.e., reads the value of Ω, and the other is a **backward** transition that applies rule $\overline{Start1}$, i.e., starts a new node, with $p_1 = p \neq p_2$.

- one is a **forward** transition that applies rule $Start1$, i.e., starts a new node, and the other is a **backward** transition that applies rule $\overline{Nodes}$, i.e., reads the value of Ω.
- one is a **forward** transition that tried to start $node$, and the other is a **backward** transition that applies rule $\overline{Start1}$ and undoes the creation of $node$, i.e., $p_2 \neq p, r_2 = \overline{Start1}$, and $k_2 = \mathsf{start}(\theta, e, succ, node)$, for some control (θ, e).

Two coinitial transition are *concurrent* if they are not in conflict.

Below we prove that this definition of concurrency makes sense by proving the square lemma, a fundamental result to prove causal consistency.

Lemma 3.3 (Square lemma)**.** Given two coinitial concurrent transitions $t_1 = (s \rightleftharpoons_{p_1,r_1,k_1} s_1)$ and $t_2 = (s \rightleftharpoons_{p_2,r_2,k_2} s_2)$, there exist two cofinal transitions $t_2/t_1 = (s_1 \rightleftharpoons_{p_2,r_2} s')$ and $t_1/t_2 = (s_2 \rightleftharpoons_{p_1,r_1} s')$. Graphically,



Proof. We distinguish the following cases depending on the applied rules:

(a) Two **forward** transitions. Then we have the following cases:

- Two transitions t_1 and t_2 where $r_1 \neq Sched$ and $r_2 \neq Sched$. Trivially, they apply to different processes, i.e., $p_1 \neq p_2$. There are few problematic cases, we will now analyse them:
  - i) t_1 starts a node and t_2 spawns a process, if both rules consider the same node, and this one is not part of the network yet, by applying t_1 to p_2 and t_2 to p_1 we would end up in two different systems.
  - ii) both transitions start a node that is not part of Ω yet, if both rules consider the same node by applying t_1 to p_2 and t_2 to p_1 we would end up in two different systems, one system would be the one where p_1 succeeded in the creation of the node and p_2 failed, and the other would the one where p_1 failed and p_2 succeeded.
  - iii) $r_1 = Start1$ and $r_2 = Nodes$, by applying t_1 to p_2 and t_2 to p_1 we would end up in two different systems, indeed in one system the nodes history item of p_2 contains the node started by p_1, in the other, since rule $Start1$ is applied after $Nodes$, the history item does not contains the node started by p_1.

Nonetheless, i, ii, iii are not concurrent transitions, therefore, such situations cannot happen. Then, for the remaining cases, we can easily prove that by applying rule r_2 to p_1 in s_1 and rule r_1 to p_2 in s_2 we have two transitions t_1/t_2 and t_2/t_1 which produce the corresponding history items and are cofinal.

- One transition t_1 which applies rule $r_1 = Sched$ to deliver message $\{\lambda_1, v_1\}$ to process $p_1 = p$, and another transition which applies a rule r_2 different from *Sched*. All cases but $r_2 = Receive$ with $p_2 = p$ and $k_2 = \mathsf{rec}(\theta, e, \{\lambda_2, v_2\}, q)$ are straightforward. Note that $\lambda_1 \neq \lambda_2$ since these identifiers are unique. Here, by applying rule Receive to s_1 and rule *Sched* to s_2 we will end up with the same mailbox in p (since it is a FIFO queue). However, the history item $\mathsf{rec}(\theta, e, \{\lambda_2, v_2\}, q')$ will be necessarily different since $q \neq q'$ by the application of rule *Sched*. This situation, though, cannot happen since transitions using rules *Sched* and *Receive* are not concurrent.
- Two transitions t_1 and t_2 with rules $r_1 = r_2 = Sched$ delivering messages $\{\lambda_1, v1\}$ and $\{\lambda_2, v_2\}$, respectively. Since the transitions are concurrent, they should deliver the messages to different processes, i.e., $p_1 \neq p_2$. Therefore, we can easily prove that delivering $\{\lambda_2, v_2\}$ from s_1 and $\{\lambda_1, v_1\}$ from s_2 we get two cofinal transitions.

(b) One forward transition and one backward transition. Then, we distinguish the following cases:

- If the two transitions apply to the same process, i.e., $p_1 = p_2$, then, since they are concurrent, we can only have $r_1 = Sched$ and a rule different from both $\overline{Sched}$ and $\overline{Receive}$, or $r_1 = \overline{Sched}$ and a rule different from both *Sched* and *Receive*. In these cases, the claim follows easily by a case distinction on the applied rules.
- Let us now consider that the transitions apply to different processes, i.e., $p_1 \neq p_2$, and the applied rules are different from *Sched* and from $\overline{Sched}$. In this case, the claim follows easily except when:
  i) one transition considers a process p and the other one undoes the spawning of the same process p.
  ii) one transition considers the spawn of a process on a node and the other one undoes the start of the same node.
  iii) one transition applies a *Nodes* on a process, say p_1, and the other undoes the start of a node.
  iv) one transition applies a $Start1$ and the other undoes a *Nodes*.

  These cases, however, are not allowed since the transitions are concurrent.

- Finally, let us consider that the transitions apply to different processes, i.e., $p_1 \neq p_2$, and that one transition applies rule *Sched* to deliver a message $\{\lambda, v\}$ from sender p to receiver p', i.e., $p_1 = p', r_1 = Sched$ and $k_1 = \mathsf{sched}(\{\lambda, v\})$. In this case, the other transition should apply a rule r_2 different from $\overline{Send}$ with $k_2 = \mathsf{send}(\theta, e, p', \{\lambda, v\})$ for some control (θ, e) since, otherwise, the transitions would not be concurrent. In any other case, one can easily prove that by applying r_2 to s_1 and *Sched* to s_2 we get two cofinal transitions.

(c) Two backward transitions. We distinguish the following cases:

- If the two transitions apply to different processes, the claim follows easily, except when:
  i) one rule is $\overline{Start1}$, and the other is $\overline{Nodes}$, and the started node is not part of the nodes read by *Nodes*, however, this case is not allowed by the side condition of the backward rule $\overline{Nodes}$.
  ii) $r_1 = \overline{Start2}$, $r_2 = \overline{Start1}$ and $k_1 = \mathsf{start}(\theta, e, \mathit{fail}, \mathit{node})$, $k_2 = \mathsf{start}(\theta, e, \mathit{succ}, \mathit{node})$, however, this case is not allowed from rule $\overline{Start1}$.
  iii) one rule is $\overline{Start1}$, and the other is $\overline{Spawn1}$, and the spawned process is on the node started by *Start1*, however, this case is not allowed by the side condition of the backward rule $\overline{Start1}$.
  iv) one rule is $\overline{Start1}$, and the other is $\overline{Spawn2}$, and the spawned process does not exists because the node was not part of the network at the time of the spawn, however, this case is not allowed by the side condition of the backward rule $\overline{Spawn2}$.
- Let us now consider that they apply to the same process, i.e., $p_1 = p_2$ and that the applied rules are different from $\overline{Sched}$. This case is not possible since, given a system, only one backward transition rule different from $\overline{Sched}$ is applicable (i.e., the one that corresponds to the last item in the history).
- Let us consider that both transitions apply to the same process and that both are applications of rule $\overline{Sched}$. This case is not possible since rule $\overline{Sched}$ can only take the newest message from the local queue of the process, and thus only one rule $\overline{Sched}$ can be applied to a given process.
- Finally, consider that both transitions apply to the same process and only one of them applies rule $\overline{Sched}$. In this case, the only non-trivial case is when the other applied rule is $\overline{Receive}$, since both change the local queue of the process. However, this case is not allowed by the backward semantics, since the conditions to apply rule $\overline{Sched}$ and rule $\overline{Receive}$ are non-overlapping.

□

Now, in [1] more results have been proved in order to prove finally causal consistency, here we take a different approach and we will rely on the results shown in [12].

The main goal of the paper is to simplify the work required to demonstrate properties like causal-consistency. In order to obtain such result they consider an abstract model, i.e., a labelled transition system with independence equipped with reverse transitions, and use a small set of axioms to demonstrate desirable properties of the system.

Now, if we prove that those axioms hold for our reversible system we get for free the proofs of the most relevant properties.

More precisely, we have to show that five axioms are true: *Square Property* (SP), *Backward Transitions are Independent* (BTI), *Well-Foundedness* (WF), *Coinitial Propagation of Independence* (CPI), and *Coinitial Independence Respects Event* (CIRE).

Now let us discuss the validity of the axioms. First, SP is proved in Lemma 3.3, BTI is trivial from the definition of concurrency (Definition 3.7), and WF holds since the pair of integers (total number elements in histories, total number of message queued) ordered under lexicographical order are always positive and decrease at each backward step. Indeed, every reverse rule, but $\overline{Sched}$, remove an item from the history and each reverse *Sched* removes a message from a process' queue. Ultimately, CPI and CIRE hold since the notion of concurrency is given on the annotated labels, and by [12, Proposition 5.4] Consequently, since we have proved the fundamental set of axioms following the proof in [12] one can derive causal consistency and many other properties (an exhaustive list can be found in [12, Table 1]).

In the remaining of this section we will show other properties which will come into play in the next section to prove properties of the rollback semantics.

Lemma 3.4 (Switching lemma)**.** Given two composable transitions of the form $t_1 = (s_1 \rightleftharpoons_{p_1,r_1,k_1} s_2)$ and $t_2 = (s_2 \rightleftharpoons_{p_2,r_2,k_2} s_3)$ such that $\overline{t_1}$ and t_2 are concurrent, there exist a system s_4 and two composable transitions $t_{2\langle\langle t_1} = (s_1 \rightleftharpoons_{p_2,r_2,k_2} s_4)$ and $t_{1\rangle\rangle t_2} = (s_4 \rightleftharpoons_{p_1,r_1,k_1} s_3)$.

Proof. First, using the loop lemma (Lemma 3.2), we have $\overline{t_1} = (s_2 \rightleftharpoons_{p_1,\overline{r_1},k_1} s_1)$. Now, since $\overline{t_1}$ and t_2 are concurrent, by applying the square lemma (Lemma 3.3) to $\overline{t_1} = (s_2 \rightleftharpoons_{p_1,\overline{r_1},k_1} s_1)$ and $t_2 = (s_2 \rightleftharpoons_{p_2,r_2k_2} s_3)$, there exists a system s_4 such that $\overline{t_{1\rangle\rangle t_2}} = \overline{t_1}/t_2 = (s_3 \rightleftharpoons_{p_1,\overline{r_1},k_1} s_4)$ and $t_{2\langle\langle t_1} = t_2/\overline{t_1} = (s_1 \rightleftharpoons_{p_2,r_2,k_2} s_4)$. Using the loop lemma (Lemma 3.2) again, we have $t_{1\rangle\rangle t_2} = t_1/t_2 = (s_4 \rightleftharpoons_{p_1,r_1,k_1} s_3)$, which concludes the proof. □

The following auxiliary result will be crucial to prove several properties of the rollback semantic.

Lemma 3.5 (Shortening lemma)**.** Let d_1 and d_2 be coinitial and cofinal derivations, such that d_2 is a forward derivation while d_1 contains at least one backward transition.

Then, there exists a forward derivation d_1' of length strictly less than that of d_1 such that $d_1' \approx d_1$.

Proof. We prove this lemma by induction on the length of d_1. By the parabolic lemma ([12, Parabolic Lemma]) there exist a backward derivation d and a forward derivation d' such that $d_1 \approx d; d'$. Furthermore, $d; d'$ is not longer than d_1. Let $s_1 \prec\!\!-_{p_1,\overline{r_1},k_1} s_2 \rightarrow_{p_2,r_2,k_2} s_3$ be the only two successive transitions in $d; d'$ with opposite direction. We will show below that there is in d' a transition t which is the inverse of $s_1 \prec\!\!-_{p_1,\overline{r_1},k_1} s_2$. Moreover, we can swap t with all the transitions between t and $s_1 \prec\!\!-_{p_1,\overline{r_1},k_1} s_2$, in order to obtain a derivation in which $s_1 \prec\!\!-_{p_1,\overline{r_1},k_1} s_2$ and t are adjacent. To do so we apply the switching lemma (Lemma 3.4), since for all transitions t' in between, we have that $\overline{t'}$ and t are concurrent (this is proved below too). When $s_1 \prec\!\!-_{p_1,\overline{r_1},k_1} s_2$ and t are adjacent we can remove both of them using $\approx$. The resulting derivation is strictly shorter, thus the claim follows by inductive hypothesis.

Let us now prove the results used above. Thanks to the loop lemma (Lemma 3.2) we have the derivations above iff we have two forward derivation which are coinitial (with s_2 as initial state) and cofinal: $\overline{d}; d_2$ and d'. We first consider the case where $\overline{r_1} \neq \mathit{Sched}$. Since the first transition of $\overline{d}; d_2$, $\overline{(s_1 \prec\!\!-_{p,\overline{r_1},k_1} s_2)}$, adds item k_1 to the history of p_1 and such item is never removed (since the derivation is forward), then the same item k_1 has to be added also by a transition in d', otherwise the two derivation cannot be cofinal. The earliest transition in d' adding item k_1 is exactly t. Let us now justify that for each transition t' before t in d' we have that $\overline{t'}$ and t are concurrent. First, t' is a forward transition and it should be applied to a process which is different from p_1, otherwise the item k_1 would be added in the wrong position in the history of p_1. We consider the following cases:

- If t' applies rule $\mathit{Spawn}1$ to create process p, then t should not apply to process p since the process p_1, to which t applies, already existed before t'. Therefore, $\overline{t'}$ and t are concurrent.

- If t' applies rule Send to send a message to some process p, then t cannot deliver the same message since we know that t is not a Sched since it adds k_1 to the history. Thus, t and t' are concurrent.

- If t' applies rule $\mathit{Start}1$ to start a node, then t should not apply rule Nodes otherwise the two systems could not possibly be cofinal, indeed in one system the history item of rule Nodes would contain the node started by $\mathit{Start}1$ and the other would not. Thus, t and t' are concurrent.

- If t' applies rule $\mathit{Start}1$ to start a node, then t should not spawn a process on the node started by t', indeed if that was the case we would end up with two

different systems, one where the spawn was successful and one where the spawn failed. Thus, t and t' are concurrent.

- If t' tries to spawn a process and fails, i.e., applies rule $Spawn2$, then t should not start the node which made the spawn fail, otherwise the two systems would not be cofinal. Thus, t and t' are concurrent.
- If t' applies rule $Nodes$, then t should not apply rule $Start1$, otherwise the history item created by the application of rule $Nodes$ would be different in the two systems and consequently they would not be cofinal. Thus, t and t' are concurrent.
- If t' applies some other rules, then t' and t are clearly concurrent.

Now we consider the case $\overline{r_1} = \overline{Sched}$ with $k_1 = \mathsf{sched}(\{\lambda, v\})$, so that $\overline{(s_1 \leftharpoondown_{p_1,\overline{Sched},k_1} s_2)}$ adds a message $\{\lambda, v\}$ to the queue of p_1. We now distinguish two cases according to whether there is in $\overline{d}; d_2$ an application of rule $Receive$ or not:

- If the forward derivation $\overline{d}; d_2$ contains no application of rule $Receive$ to p_1 then, in the final state, the queue of process p_1 contains the message. Hence, d' needs to contain the a $Sched$ for the same message. The earliest such $Sched$ transition in d' is exactly t.

  Let us now justify that for each transition t' before t in d' we have that $\overline{t'}$ and t are concurrent. Consider the case where t' applies rule $Sched$ to deliver a different message to the same process p_1. Since no receive would be performed on p_1 then the queue will stay different, and the two derivations could not be cofinal, hence this case can never happen. In all other cases the two transitions are concurrent.
- If the forward derivation $\overline{d}; d_2$ contains at least an application of $Receive$ to p_1, let us consider such first application. In order for the two derivations to be cofinal, the same history item needs to be created in d'. The queue stored in k_2 has a suffix $\{\lambda, v\} : q$, hence also in d' the first $Sched$ delivering a message to p_1 should deliver message $\{\lambda, v\} : q$. Since there are no other $Sched$ nor $Receive$ targeting p_1 then the $Sched$ delivering message $\{\lambda, v\}$ to p_1 is concurrent to all previous transitions as desired.

□

3.4 Rollback semantics

In this section, we introduce a rollback semantics which will allow the user to undo automatically several steps in a causal-consistent way, until the system reaches a specified

$$(\underline{U-Satisfy}) \quad \frac{\Gamma;\Pi;\Omega \rightharpoonup_{p,r,\Psi'} \Gamma';\Pi';\Omega' \ \wedge \ \psi \in \Psi'}{\lceil\lceil\Gamma;\Pi;\Omega\rceil\rceil_{\{p,\psi\}+\Psi} \rightsquigarrow \lceil\lceil\Gamma';\Pi';\Omega'\rceil\rceil_{\Psi}}$$

$$(\underline{U-Sched1}) \quad \frac{\Gamma;\Pi;\Omega \rightharpoonup_{p,r,\{s,\lambda'^{sched}\}} \Gamma';\Pi';\Omega' \ \wedge \lambda'^{sched} \neq \lambda^{sched}}{\lceil\lceil\Gamma;\Pi;\Omega\rceil\rceil_{\{p,\lambda^{sched}\}+\Psi} \rightsquigarrow \lceil\lceil\Gamma';\Pi';\Omega'\rceil\rceil_{\{p,\lambda^{sched}\}+\Psi}}$$

$$(\underline{U-Sched2}) \quad \frac{\forall\lambda' \in \mathbb{N} \ \ \Gamma;\Pi;\Omega \not\rightharpoonup_{p,r,\{s,\lambda'^{sched}\}} \wedge \ \Gamma;\Pi;\Omega \rightharpoonup_{p,r,\Psi} \Gamma';\Pi';\Omega'}{\lceil\lceil\Gamma;\Pi;\Omega\rceil\rceil_{\{p,\lambda^{sched}\}+\Psi} \rightsquigarrow \lceil\lceil\Gamma';\Pi';\Omega'\rceil\rceil_{\{p,\lambda^{sched}\}+\Psi}}$$

$$(\underline{U-Act1}) \quad \frac{\Gamma;\Pi;\Omega \rightharpoonup_{p,r,\Psi'} \Gamma';\Pi';\Omega' \ \wedge \ \psi \notin \Psi' \ \wedge \ \lambda^{sched} \notin \Psi' \ \wedge \ \psi \neq \lambda^{sched} \ \forall\lambda \in \mathbb{N}}{\lceil\lceil\Gamma;\Pi;\Omega\rceil\rceil_{\{p,\psi\}+\Psi} \rightsquigarrow \lceil\lceil\Gamma';\Pi';\Omega'\rceil\rceil_{\{p,\psi\}+\Psi}}$$

$$(\underline{U-Act2}) \quad \frac{\Gamma;\Pi;\Omega \not\rightharpoonup_{p,r,\Psi'} \wedge \ \psi \notin \Psi' \wedge \ \Gamma;\Pi;\Omega \rightharpoonup_{p,r,\{s,\lambda^{sched}\}} \Gamma';\Pi';\Omega' \ \wedge \ \psi \neq \lambda^{sched} \ \forall\lambda \in \mathbb{N}}{\lceil\lceil\Gamma;\Pi;\Omega\rceil\rceil_{\{p,\psi\}+\Psi} \rightsquigarrow \lceil\lceil\Gamma';\Pi';\Omega'\rceil\rceil_{\{p,\psi\}+\Psi}}$$

$$(\underline{Send}) \quad \frac{\Gamma;\langle node,p,\mathsf{send}(\theta,e,p',\{\lambda,v\}):h,(\theta',e'),q\rangle \mid \Pi;\Omega \not\rightharpoonup_{p,r,\Psi'}}{\begin{array}{c}\lceil\lceil\Gamma;\langle node,p,\mathsf{send}(\theta,e,p',\{\lambda,v\}):h,(\theta,e),q\rangle \mid \Pi;\Omega\rceil\rceil_{\{p,\psi\}+\Psi} \\ \rightsquigarrow \lceil\lceil\Gamma;\langle node,p,\mathsf{send}(\theta,e,p',\{\lambda,v\}):h,(\theta',e'),q\rangle \mid \Pi;\Omega\rceil\rceil_{(\{p',\lambda^{sched}\},\{p,\psi\})+\Psi}\end{array}}$$

$$(\underline{Spawn1}) \quad \frac{\Gamma;\langle node,p,\mathsf{spawn}(\theta,e,node',p'):h,(\theta',e'),q\rangle \mid \Pi;\Omega \not\rightharpoonup_{p,r,\Psi'} \ \ exists(p',\Pi)=true}{\begin{array}{c}\lceil\lceil\Gamma;\langle node,p,\mathsf{spawn}(\theta,e,node',p'):h,(\theta,e),q\rangle \mid \Pi;\Omega\rceil\rceil_{\{p,\psi\}+\Psi} \\ \rightsquigarrow \lceil\lceil\Gamma;\langle node,p,\mathsf{spawn}(\theta,e,node',p'):h,(\theta,e),q\rangle \mid \Pi;\Omega\rceil\rceil_{(\{p',s\},\{p,\psi\})+\Psi}\end{array}}$$

$$(\underline{Spawn2}) \quad \frac{\Gamma;\langle node,p,\mathsf{spawn}(\theta,e,node',p'):h,(\theta',e'),q\rangle \mid \Pi;\Omega \not\rightharpoonup_{p,r,\Psi'} \quad exists(p',\Pi)=false}{\begin{array}{c}\lceil\lceil\Gamma;\langle node,p,\mathsf{spawn}(\theta,e,node',p'):h,(\theta,e),q\rangle \mid \Pi;\Omega\rceil\rceil_{\{p,\psi\}+\Psi} \\ \rightsquigarrow \lceil\lceil\Gamma;\langle node,p,\mathsf{spawn}(\theta,e,node',p'):h,(\theta,e),q\rangle \mid \Pi;\Omega\rceil\rceil_{(\{p'',st_{node'}\},\{p,\psi\})+\Psi} \\ \text{where } p''=node\_parent(node',\Pi)\end{array}}$$

$$(\underline{Nodes}) \quad \frac{\Gamma;\langle node,p,\mathsf{nodes}(\theta,e,\Omega):h,(\theta',e'),q\rangle \mid \Pi;\Omega' \not\rightharpoonup_{p,r,\Psi}}{\begin{array}{c}\lceil\lceil\Gamma;\langle node,p,\mathsf{nodes}(\theta,e,\Omega):h,(\theta,e),q\rangle \mid \Pi;\Omega'\rceil\rceil_{\{p,\psi\}+\Psi'} \\ \rightsquigarrow \lceil\lceil\Gamma;\langle node,p,\mathsf{nodes}(\theta,e,\Omega):h,(\theta,e),q\rangle \mid \Pi;\Omega'\rceil\rceil_{(\{p',st_{node'}\},\{p,\psi\})+\Psi} \\ \text{where } node'=fst(list(\Omega'\setminus\Omega)) \ \wedge \ p'=node\_parent(node',\Pi)\end{array}}$$

$$(\underline{Start}) \quad \frac{\Gamma;\langle node,p,\mathsf{start}(\theta,e,succ,node'):h,(\theta',e'),q\rangle \mid \Pi;\Omega' \not\rightharpoonup_{p,r,\Psi}}{\begin{array}{c}\lceil\lceil\Gamma;\langle node,p,\mathsf{start}(\theta,e,succ,node'):h,(\theta,e),q\rangle \mid \Pi;\Omega'\rceil\rceil_{\{p,\psi\}+\Psi'} \\ \rightsquigarrow \lceil\lceil\Gamma;\langle node,p,\mathsf{start}(\theta,e,succ,node'):h,(\theta,e),q\rangle \mid \Pi;\Omega'\rceil\rceil_{(\{p',\psi'\},\{p,\psi\})+\Psi} \\ \text{where } p'=fst(reads(node,\Pi)) \ \wedge \ \psi'=\{s\} \text{ if } reads(node,\Pi)\neq[\,] \\ \text{or } p'=fst(tried\_starts(node,\Pi)) \ \wedge \ \psi'=\{s\} \text{ if } tried\_starts(node,\Pi)\neq[\,] \\ \text{or } p''=fst(procs(node',\Pi)) \wedge \ p'=proc\_parent(p'',\Pi) \ \wedge \ \psi'=\{sp_{p''}\}\end{array}}$$

Figure 3.11: Rollback semantics

state. The rollback operator that we are about to introduce has been strongly inspired by the one introduced in [13].

Similarly to what we have done for the reversible semantics, also here we make use of auxiliary functions, the reader will find an informal description of the function in this section every time one is used, instead for a formal description the user can refer to Appendix A.

Usually, when it comes to debugging the user has an idea where the bug is located and if the computation proceeds too much it might be tedious for him/her to undo it step by step, therefore we introduce the rollback operator.

We denote a system in rollback mode by $[\![s]\!]_{\{p,\psi\}}$, where the subscript means that we wish to undo the action ψ performed by process p, and, of course, every action which depends on it. More in general, the subscript of $[\![\,]\!]$, often depicted with Ψ or Ψ', can be seen as a stack of requests that need to be undone, once this stack is empty it means that the system has reached the state desired by the user. In this work, we consider the following kinds of requests:

- $\{p, s\}$: a step back for the process p
- $\{p, \lambda^{\Downarrow}\}$: a backward derivation of process p up to the receiving of the message uniquely identified by λ
- $\{p, \lambda^{\Uparrow}\}$: a backward derivation of process p up to the sending of the message uniquely identified by λ
- $\{p, \lambda^{sched}\}$: a backward derivation until the scheduling of the message uniquely identified by λ
- $\{p, st_{node}\}$: a backward derivation of process p up to the start of *node*
- $\{p, sp_{p'}\}$: a backward derivation of process p up to the spawn of p'

Moreover, this is where the labels introduced in the backward semantic of Fig.3.7 come into play. Formerly, we defined the backward semantics in terms of the relation $\leftharpoondown_{p,r,\Psi}$, where:

- p represents the pid of the process which is performing the backward transition
- r represents the rule applied
- Ψ represents the requests satisfied by the backward transition

The controlled rules for the rollback semantics are shown in Fig. 3.11, and we will now proceed to analyse them.

Here, we have two kind of rules, the ones belonging to the first kind are those rules prefixed by $\underline{U}$, these five rules are the ones actually performing a step back, then, the ones belonging to the second kind are the remaining rules, and those are the rules which push another request on the top of the stack because a consequence of the action that we desire to undo has to be undone beforehand.

Example 3.5. In Fig. 3.12 and Fig. 3.13 one can see the rollback semantics in action. In order to have a more readable and coincise derivation we used $C[e]$ to represents the context, where e is the next expression which needs to be evaluated. Where possible we underline the element of the history which influence the application of the next rule, unfortunately that is not always possible because sometimes we need to undo τ-actions that are not show in the history.

As one can see, to satisfy the request the semantics starts to undo steps on the targeted process, then once it founds itself stuck it starts to push new requests on top of the stack in order to undo the action which has a causal dependency on the primary request. Then, once all the consequences of the start have been undone the semantics proceed to satisfy the user's request.

Let us begin with the rules of the first kind. The rollback semantics has to undo only the actions that are in the stack, and every action which depends on one of them, the remaining actions must not be undone.

During the process of rolling back, it might happen that more than one backward rule could be applied to a process, i.e., a $\overline{Sched}$ and another one, therefore we need to ensure that the minimum amount of backward steps are performed.

Intuitively, since rules *Sched* do not commute, i.e., they have to be undone in the same order as they have been performed during the forward computation in order to ensure causal-consistency, if our goal is to undo a *Sched* and more than one backward rule can be applied we have to choose the $\overline{Sched}$, regardless of the fact that the $\overline{Sched}$ undoes the scheduling of the "right" message or not (the right message is the one identified by the λ in the request on top of the stack). Similarly, if we aim to undo every action different from *Sched* and we have more than one option available we need to go for the action which is not a $\overline{Sched}$. In both situations, if we have only one backward rule available, obviously, we have to undo that one. Rules prefixed by $\underline{U}$ ensure exactly this property.

Now, let us now discuss the rules of the second kind by making a premise: every time that we apply one of these rules it means that not all the criteria required to perform a backward step, on the desired process, are met. Therefore, in order to meet all the criteria and to ensure causal-consistency we need to push onto the stack of requests new requests in order to undo all the consequences of the desired action.

Let us start with rule $\underline{Send}$. In this scenario rule *Send* cannot be undone because the message is not in Γ, the global mailbox, therefore that means that the message has

been delivered to the receiver, and rule *Sched* is the only one able to remove a message from Γ. Hence, in order to undo the send we have first to undo the *Sched*, to that end we push on top of the stack of request $\{p', \lambda^{sched}\}$, in other words we first need to undo the *Sched* which has delivered the message.

Rule *Spawn1* is the one that we apply when we intend to undo the successful spawn of a process, say p_1, but p_1 has not an empty history. For this reason, we push the request of a step back for p_1, then once the step back has been performed we check if it is possible to undo the spawn or not, if yes we undo it, if no we push again the request for a step back of p_1. Since there is no infinite backward computation, sooner or later, the criteria required to undo the spawn, i.e., an empty history and an empty queue of messages, will be met and the spawn will be undone.

Then, we have rule *Spawn2*, which takes care of undoing a failed spawn; a failed spawn cannot be undone if someone started the node which was not part of Ω when the spawn failed, therefore if this node is now part of Ω we have to undo its start first, and we do so by pushing the request $\{p'', st_{node}\}$, where p'' represents the pid of the process which started the node. Here, the auxiliary function *node_parent* scans the system searching for the process responsible for the start of *node'*.

Moving on, we have rule *Nodes*. This rule cannot be undone if the Ω saved in the history item is not identical to the one of the system, because otherwise if we undo the nodes and then we redo it we would obtain a different result, entering thus a new state. To this end, if this condition is not met we push, one by one, the request of undoing the nodes contained in the difference of the two sets. The decision of pushing these requests one by one is due to the fact that by adding them all together we could face a tricky situation. Indeed, one of the requests, say $\{p, \psi\}$, in Ψ could have depended on one of the deepest requests, and then while undoing $\{p, \psi\}$ we would have had to undo also the dependency, i.e., pushing on the stack a request which is already present, having then two identical requests in the stack. Let us clarify this scenario with an example.

Example 3.6. Let say that we have a system with two processes, p_1 and p_2, running in the same node, i.e., $node_1$, now p_1 perform a nodes, then it starts a new node, $node_2$, and finally it sends a message to p_2. Now, p_2 receives the message from p_1 and then starts another node, $node_3$. In this scenario, in order to undo the nodes performed by p_1 we first need to undo the start of $node_2$ and $node_3$, but if we push all the requests together and we push $\{p_1, st_{node_2}\}$ after $\{p_2, st_{node_3}\}$, i.e., we have a stack of the form $\{p_1, st_{node_2}\}, \{p_2, st_{node_3}\}, \{p_1, \psi\}$. Then, in order to undo the start of $node_2$, we need to undo the send, in order to undo the send we have to undo the sched, to undo the sched we need to undo the receive, and ultimately before undoing the receive we have to undo the start of $node_3$.

A possible solution to Example 3.6 would be to perform a deep analysis of the stack when a start is removed, to check if there are other occurrences of the same start, but

this solution is intricate and to keep things as simple as possible we decided to push the requests one by one.

At the end, we have rule *Start*, which pushes requests on the stack when it is not possible to undo the successful start of the node identified by the history item (we will refer to it as $node'$). A successful start cannot be undone if there are processes still running on the node, or if someone read the node by means of rule *Nodes*, or if someone tried to start the same node and failed because the node was already part of Ω. Since here as well some of the consequences, that we have to undo, could have some dependencies we decided to push one request at the time, again to avoid the deep check on the stack and to keep things simple. Here, the auxiliary function *reads* scans the system searching for the pids of the processes which have applied rule *Nodes* when $node'$ was part of Ω, similarly *tried_start* scans the system searching for processes who tried to start $node'$ and failed, then we have *procs* which returns the list of processes currently running on $node'$, and lastly *proc_parent* given the pid of a process returns the pid of its parent.

Now let us move our focus on the relation $\rightsquigarrow$; one could say that $\rightsquigarrow$ is a controlled version of the uncontrolled reversible semantics presented in Section 3.3. Moreover, we can observe that for each derivation of the controlled rollback semantics there is an equivalent derivation of the uncontrolled reversible semantics, while the opposite is not generally true. In the following of this section we will formalise this claim, but before proceeding we need some notation. The notions of transition and derivation are easily extended to controlled derivation. Now we provide a notion of projection from controlled systems to uncontrolled ones:

$$uctrl(\llbracket \Gamma; \Pi; \Omega \rrbracket_{\Psi}) = \Gamma; \Pi; \Omega$$

The notion of projection trivially extends to derivations.

Proof. Trivially by inspection of the controlled rules, noting that rules prefixed by $\underline{U}$ execute an uncontrolled backward step, and the remaining ones do some bookkeeping which is removed by function *uctrl*. □

Moreover, our controlled semantics is not only causal-consistent but also *minimal*, this means that in order to satisfy the first request on Ψ, and consequently every other request in Ψ, we undo the least amount of steps which allow us to satisfy such requests.

In order to prove this we need to restrict our attention to those requests which require us to rollback transitions that are in the past of the process.

Definition 3.8. A controlled system $c = \llbracket s \rrbracket_{(\{p,\psi\})}$ is well initialised iff there exist a derivation d under the reversible semantics, a system $s_0 = init(d)$, an uncontrolled derivation $s_0 \rightleftharpoons^* s$, and an uncontrolled backward derivation from s satisfying $\{p, \psi\}$. A controlled derivation d is well initialised iff $init(d)$ is well initialised.

The existence of a derivation satisfying the request can be easily checked in $O(n)$ time, where n is the length of the process' history. We proceed now to prove that controlled derivations are finite.

Lemma 3.6. Let d a well initialised derivation. Then d is finite.

Proof. First note that $uctrl(d)$ is finite. Indeed, for rollback request, the total length is bounded by the length of histories and by the number of messages exchanged.
In addition to lifting the uncontrolled steps, the controlled semantics also takes some administrative steps. If we show that between each pair of uncontrolled steps there is a finite amount of administrative steps then the thesis follow. Let us consider the rollback semantics. Rule *Sched* is bounded by the number of messages exchanged. Rule *Spawn*1 can be applied only a limited amount of times since the history of the targeted process is finite, rule *Spawn*2 can be applied only one time for each process, rule *Nodes* is bounded by the number of nodes of the system. Ultimately, rule *Start* is bounded because the number of reads that other process have performed, which is finite because their history is finite, plus the number of failed start, again bounded for the finiteness of the history, plus the number of processes still running in the node is a finite value. The thesis follow. □

We can now show that all the uncontrolled transitions executed as a consequence of a rollback request depend on the action that needs to be undone.

Theorem 3.1. For each well-initialised controlled system $c = [\![\Gamma; \Pi; \Omega]\!]_{(\{p,\psi\})}$ consider a maximal derivation d with $init(d) = c$. Let us call t the last transition in $uctrl(d)$. We have that t satisfies $\{p, \psi\}$, and for each transition t' in $uctrl(d)$, $t' \rightsquigarrow t$.

Proof. We can prove that t satisfies the request $\{p, \psi\}$ by inspections of the rules, since a derivations only terminates when the request at the bottom of the stack is removed, and this is always the original request $\{p, \psi\}$. In order to show this we need to show that the controlled semantics never gets stuck, otherwise namely that if the uncontrolled semantics gets stuck a new request is generated. This can be shown by contrasting uncontrolled and controlled rules.
Now let us move to the second part of the thesis. We will show two invariants of the derivation. First, consider transitions t_1 and t_2 satisfying two rollback requests $\{p_1, \psi_1\}$ and $\{p_2, \psi_2\}$ on the stack, such that $\{p_1, \psi_1\}$ is on top of $\{p_2, \psi_2\}$. Then $t_1 \rightsquigarrow t_2$. Second if t_1 satisfies the request on top of the stack, and transition t_3 is performed, then $t_1 \rightsquigarrow t_3$. Both invariants can be proved by inspection of the rules. The thesis then follow by transitivity of $\rightsquigarrow$. □

We conclude the section by proving that we undo the least amount of step for each request, but before doing so we need to prove another result.

Proposition 3.1 (Confluence)**.** Let s be a system in the uncontrolled reversible semantics. If $s \rightleftharpoons^* s_1$ and $s \rightleftharpoons^* s_2$ then there exists s_3 such that $s_1 \backharpoonup^* s_3$ and $s_1 \backharpoonup^* s_3$.

Proof. Let s_0 be the system obtained from s by undoing all the actions. Consider the derivation $s_0 \rightharpoonup^* s \rightleftharpoons^* s_1$, where the first part exists from the loop lemma (Lemma 3.2). From the parabolic lemma ([12, Parabolic Lemma]) we have $s_0 \backharpoonup^* \rightharpoonup^* s_1$. Since there is no possible backward transition from s_0 we have $s_0 \rightharpoonup^* s_1$. Similarly, we get $s_0 \rightharpoonup s_2$. Backward confluence follows from the loop lemma. □

Theorem 3.2 (Minimality)**.** Let d be a well-initialised controlled derivation such that $init(d) = [\![s]\!]_{(\{p,\psi\})}$. Derivation $uctrl(d)$ has minimal length among all uncontrolled derivations d' with $init(d') = s$ including at least one transition satisfying $\{p, \psi\}$.

Proof. Take an uncontrolled derivation d' satisfying the premises. By definition d and d' are coinitial. We can assume that there is in d' a unique transition satisfying the request and that is the last transition in d'. For backward derivations, by backward confluence (Prop. 3.1) we can extend the derivation to cofinal derivation $d'; d''$ and $uctrl(d); d'''$ with d'' and d''' backward. Thanks to the shortening lemma (Lemma 3.5) we can assume $d'; d''$ to be backward too. By causal consistency the two derivation are causally equivalent, and since they are backward they differ only for swap of concurrent actions. Note that for each request there is a unique action satisfying it, hence there is a sequence of swaps of independent transitions transforming $d'; d''$ into $uctrl(d); d'''$. Assume towards a contradiction that $length(d') < length(d)$. Then t in d' must be swapped with some of the transitions following t in $uctrl(d)$, but this is impossible thanks to Theorem 3.1. □

$$
\begin{array}{ll}
 & \llbracket\{\,\};\ \langle n_1, \mathrm{p}, [\mathsf{spawn}(\_,\_,n_1,\mathrm{r}_1), \mathsf{spawn}(\_,\_,n_1,\mathrm{r}_2), \mathsf{send}(\_,\_,\mathrm{r}_1,m_1),\\
 & \mathsf{send}(\_,\_,\mathrm{r}_2,m_2)], (\_, C[go], [\,]\rangle \mid \langle n_1, \mathrm{r}_1, [\mathsf{rec}(\_,\_,m_2,[m_2]),\\
 & \mathsf{start}(\_,\_,\mathit{fail},n_2)], (id, C[loser]), [\,]\rangle \mid \langle n_1, \mathrm{r}_2, [\mathsf{rec}(\_,\_,m_1,[m_1]),\\
 & \mathsf{start}(\_,\_,\mathit{succ},n_2)], (id, C[winner]), [\,]\rangle & ;\{n_1,n_2\}\rrbracket_{(\{\mathrm{r}_2, st_{n_2}\})}\\
\rightsquigarrow_{\underline{U-Act1}} & \llbracket\{\,\};\ \langle n_1, \mathrm{p}, [\mathsf{spawn}(\_,\_,n_1,\mathrm{r}_1), \mathsf{spawn}(\_,\_,n_1,\mathrm{r}_2), \mathsf{send}(\_,\_,\mathrm{r}_1,m_1),\\
 & \mathsf{send}(\_,\_,\mathrm{r}_2,m_2)], (\_, C[go], [\,]\rangle \mid \langle n_1, \mathrm{r}_1, [\mathsf{rec}(\_,\_,m_2,[m_2]),\\
 & \mathsf{start}(\_,\_,\mathit{fail},n_2)], (id, C[loser]), [\,]\rangle \mid \langle n_1, \mathrm{r}_2, [\mathsf{rec}(\_,\_,m_1,[m_1]),\\
 & \mathsf{start}(\_,\_,\mathit{succ},n_2)], (id, C[\mathsf{case}\ ok\ \mathsf{of}\ \ldots]), [\,]\rangle & ;\{n_1,n_2\}\rrbracket_{(\{\mathrm{r}_2, st_{n_2}\})}\\
\rightsquigarrow_{\underline{U-Act1}} & \llbracket\{\,\};\ \langle n_1, \mathrm{p}, [\mathsf{spawn}(\_,\_,n_1,\mathrm{r}_1), \mathsf{spawn}(\_,\_,n_1,\mathrm{r}_2), \mathsf{send}(\_,\_,\mathrm{r}_1,m_1),\\
 & \mathsf{send}(\_,\_,\mathrm{r}_2,m_2)], (\_, C[go], [\,]\rangle \mid \langle n_1, \mathrm{r}_1, [\mathsf{rec}(\_,\_,m_2,[m_2]),\\
 & \mathsf{start}(\_,\_,\mathit{fail},n_2)], (id, C[loser]), [\,]\rangle \mid \langle n_1, \mathrm{r}_2, [\mathsf{rec}(\_,\_,m_1,[m_1]),\\
 & \underline{\mathsf{start}(\_,\_,\mathit{succ},n_2)}], (id, C[\mathsf{case}\ R\ \mathsf{of}\ \ldots]), [\,]\rangle & ;\{n_1,n_2\}\rrbracket_{(\{\mathrm{r}_2, st_{n_2}\})}\\
\rightsquigarrow_{\underline{Start}} & \llbracket\{\,\};\ \langle n_1, \mathrm{p}, [\mathsf{spawn}(\_,\_,n_1,\mathrm{r}_1), \mathsf{spawn}(\_,\_,n_1,\mathrm{r}_2), \mathsf{send}(\_,\_,\mathrm{r}_1,m_1),\\
 & \mathsf{send}(\_,\_,\mathrm{r}_2,m_2)], (\_, C[go], [\,]\rangle \mid \langle n_1, \mathrm{r}_1, [\mathsf{rec}(\_,\_,m_2,[m_2]),\\
 & \mathsf{start}(\_,\_,\mathit{fail},n_2)], (id, C[loser]), [\,]\rangle \mid \langle n_1, \mathrm{r}_2, [\mathsf{rec}(\_,\_,m_1,[m_1]),\\
 & \mathsf{start}(\_,\_,\mathit{succ},n_2)], (id, C[winner]), [\,]\rangle & ;\{n_1,n_2\}\rrbracket_{(\{\mathrm{r}_1, s\})+\Psi}\\
\rightsquigarrow_{\underline{U-Act1}} & \llbracket\{\,\};\ \langle n_1, \mathrm{p}, [\mathsf{spawn}(\_,\_,n_1,\mathrm{r}_1), \mathsf{spawn}(\_,\_,n_1,\mathrm{r}_2), \mathsf{send}(\_,\_,\mathrm{r}_1,m_1),\\
 & \mathsf{send}(\_,\_,\mathrm{r}_2,m_2)], (\_, C[go], [\,]\rangle \mid \langle n_1, \mathrm{r}_1, [\mathsf{rec}(\_,\_,m_2,[m_2]),\\
 & \underline{\mathsf{start}(\_,\_,\mathit{fail},n_2)}], (id, C[\mathsf{case}\ error\ \mathsf{of}\ \ldots]), [\,]\rangle\\
 & \mid \langle n_1, \mathrm{r}_2, [\mathsf{rec}(\_,\_,m_1,[m_1]), \mathsf{start}(\_,\_,\mathit{succ},n_2)],\\
 & (id, C[winner]), [\,]\rangle & ;\{n_1,n_2\}\rrbracket_{(\{\mathrm{r}_2, st_{n_2}\})}\\
\rightsquigarrow_{\underline{Start}} & \llbracket\{\,\};\ \langle n_1, \mathrm{p}, [\mathsf{spawn}(\_,\_,n_1,\mathrm{r}_1), \mathsf{spawn}(\_,\_,n_1,\mathrm{r}_2), \mathsf{send}(\_,\_,\mathrm{r}_1,m_1),\\
 & \mathsf{send}(\_,\_,\mathrm{r}_2,m_2)], (\_, C[go], [\,]\rangle \mid \langle n_1, \mathrm{r}_1, [\mathsf{rec}(\_,\_,m_2,[m_2]),\\
 & \mathsf{start}(\_,\_,\mathit{fail},n_2)], (id, C[\mathsf{case}\ error\ \mathsf{of}\ \ldots]), [\,]\rangle\\
 & \mid \langle n_1, \mathrm{r}_2, [\mathsf{rec}(\_,\_,m_1,[m_1]), \mathsf{start}(\_,\_,\mathit{succ},n_2)],\\
 & (id, C[winner]), [\,]\rangle & ;\{n_1,n_2\}\rrbracket_{(\{\mathrm{r}_1, s\})+\Psi}\\
\rightsquigarrow_{\underline{U-Act1}} & \llbracket\{\,\};\ \langle n_1, \mathrm{p}, [\mathsf{spawn}(\_,\_,n_1,\mathrm{r}_1), \mathsf{spawn}(\_,\_,n_1,\mathrm{r}_2), \mathsf{send}(\_,\_,\mathrm{r}_1,m_1),\\
 & \mathsf{send}(\_,\_,\mathrm{r}_2,m_2)], (\_, C[go], [\,]\rangle \mid \langle n_1, \mathrm{r}_1, [\mathsf{rec}(\_,\_,m_2,[m_2]),\\
 & \underline{\mathsf{start}(\_,\_,\mathit{fail},n_2)}], (id, C[\mathsf{case}\ R\ \mathsf{of}\ \ldots]), [\,]\rangle\\
 & \mid \langle n_1, \mathrm{r}_2, [\mathsf{rec}(\_,\_,m_1,[m_1]), \mathsf{start}(\_,\_,\mathit{succ},n_2)], (id, C[winner]),\\
 & [\,]\rangle\\
 & & ;\{n_1,n_2\}\rrbracket_{(\{\mathrm{r}_2, st_{n_2}\})}
\end{array}
$$

Figure 3.12: The rollback semantics applied on the final state of the program depicted in Fig. 3.8, where Ψ represents $\{r_2, st_{n_2}\}$, n_1 represents the main node and n_2 represents $'another@node'$. (part 1/2)

$\rightsquigarrow_{\underline{Start}}$ $[\!|\{\,\};\ \langle n_1, \mathrm{p}, [\mathsf{spawn}(\_,\_,n_1,\mathrm{r}_1), \mathsf{spawn}(\_,\_,n_1,\mathrm{r}_2), \mathsf{send}(\_,\_,\mathrm{r}_1,m_1),$
$\mathsf{send}(\_,\_,\mathrm{r}_2,m_2)], (\_, C[go], [\,]\rangle \mid \langle n_1, \mathrm{r}_1, [\mathsf{rec}(\_,\_,m_2,[m_2]),$
$\underline{\mathsf{start}(\_,\_,\mathit{fail},n_2)}], (id, C[\mathsf{case}\ R\ \mathsf{of}\ \ldots]), [\,]\rangle$
$\mid \langle n_1, \mathrm{r}_2, [\mathsf{rec}(\_,\_,m_1,[m_1]), \mathsf{start}(\_,\_,succ,n_2)],$
$(id, C[winner]), [\,]\rangle$ $;\{n_1,n_2\}|\!]_{(\{\mathrm{r}_1,s\})+\Psi}$

$\rightsquigarrow_{\underline{U-Act1}}$ $[\!|\{\,\};\ \langle n_1, \mathrm{p}, [\mathsf{spawn}(\_,\_,n_1,\mathrm{r}_1), \mathsf{spawn}(\_,\_,n_1,\mathrm{r}_2), \mathsf{send}(\_,\_,\mathrm{r}_1,m_1),$
$\mathsf{send}(\_,\_,\mathrm{r}_2,m_2)], (\_, C[go], [\,]\rangle$
$\mid \langle n_1, \mathrm{r}_1, [\mathsf{rec}(\_,\_,m_2,[m_2])], (id, C[\mathsf{start}(another, node)]), [\,]\rangle$
$\mid \langle n_1, \mathrm{r}_2, [\mathsf{rec}(\_,\_,m_1,[m_1]), \mathsf{start}(\_,\_,succ,n_2)],$
$(id, C[winner]), [\,]\rangle$ $;\{n_1,n_2\}|\!]_{(\{\mathrm{r}_2,st_{n_2}\})}$

$\rightsquigarrow_{\underline{U-Act2}}$ $[\!|\{\,\};\ \langle n_1, \mathrm{p}, [\mathsf{spawn}(\_,\_,n_1,\mathrm{r}_1), \mathsf{spawn}(\_,\_,n_1,\mathrm{r}_2), \mathsf{send}(\_,\_,\mathrm{r}_1,m_1),$
$\mathsf{send}(\_,\_,\mathrm{r}_2,m_2)], (\_, C[go], [\,]\rangle$
$\mid \langle n_1, \mathrm{r}_1, [\mathsf{rec}(\_,\_,m_2,[m_2])], (id, C[\mathsf{start}(another, node)]), [\,]\rangle$
$\mid \langle n_1, \mathrm{r}_2, [\mathsf{rec}(\_,\_,m_1,[m_1]), \mathsf{start}(\_,\_,succ,n_2)],$
$(id, C[\mathsf{case}\ ok\ \mathsf{of}\ \ldots]), [\,]\rangle$ $;\{n_1,n_2\}|\!]_{(\{\mathrm{r}_2,st_{n_2}\})}$

$\rightsquigarrow_{\underline{U-Act2}}$ $[\!|\{\,\};\ \langle n_1, \mathrm{p}, [\mathsf{spawn}(\_,\_,n_1,\mathrm{r}_1), \mathsf{spawn}(\_,\_,n_1,\mathrm{r}_2), \mathsf{send}(\_,\_,\mathrm{r}_1,m_1),$
$\mathsf{send}(\_,\_,\mathrm{r}_2,m_2)], (\_, C[go], [\,]\rangle$
$\mid \langle n_1, \mathrm{r}_1, [\mathsf{rec}(\_,\_,m_2,[m_2])], (id, C[\mathsf{start}(another, node)]), [\,]\rangle$
$\mid \langle n_1, \mathrm{r}_2, [\mathsf{rec}(\_,\_,m_1,[m_1]), \mathsf{start}(\_,\_,succ,n_2)],$
$(id, C[\mathsf{case}\ R\ \mathsf{of}\ \ldots]), [\,]\rangle$ $;\{n_1,n_2\}|\!]_{(\{\mathrm{r}_2,st_{n_2}\})}$

$\rightsquigarrow_{\underline{U-Act1}}$ $[\!|\{\,\};\ \langle n_1, \mathrm{p}, [\mathsf{spawn}(\_,\_,n_1,\mathrm{r}_1), \mathsf{spawn}(\_,\_,n_1,\mathrm{r}_2), \mathsf{send}(\_,\_,\mathrm{r}_1,m_1),$
$\mathsf{send}(\_,\_,\mathrm{r}_2,m_2)], (\_, C[go], [\,]\rangle$
$\mid \langle n_1, \mathrm{r}_1, [\mathsf{rec}(\_,\_,m_2,[m_2])], (id, C[\mathsf{start}(another, node)]), [\,]\rangle$
$\mid \langle n_1, \mathrm{r}_2, [\mathsf{rec}(\_,\_,m_1,[m_1])], (id, C[\mathsf{start}(another, node)]), [\,]\rangle$ $;\{n_1,n_2\}|\!]_{()}$

Figure 3.13: The rollback semantics applied on the final state of the program depicted in Fig. 3.8, where Ψ represents $\{r_2, st_{n_2}\}$, n_1 represents the main node and n_2 represents $'another@node'$. (part 2/2)

Chapter 4

Distributed CauDEr

In this chapter we will discuss the implementation of the debugger and how it works from the user perspective. The first section will describe the already existent CauDEr, then the second section will describe how it has been expanded in order to support the distributed functions and how it changed. Furthermore, in the second section we will provide examples of bugged programs, which display typical problems of concurrent and distributed programming and we will show how CauDEr can be used to detect those bugs.

4.1 CauDEr

In [1] CauDEr has been presented, here we remind the reader that CauDEr is a reversible debugger designed for Core Erlang, a much simpler version of Erlang, indeed Core Erlang works as an intermediate representation of a program during its compilation. However, since it would not be practical for the user to write programs directly in Core Erlang the debugger automatically translates the Erlang source code into Core Erlang, having then the advantage of dealing with a simple language while offering the user the ability to write Erlang programs.

The debugger works as follow: first the user has to select the Erlang source file, then the file is compiled into Core Erlang, if the compilation succeeds CauDEr will show the user the Core Erlang source code and the user can choose which function will be the entry point of the program and feed to it its arguments, if the compilation fails then an error is returned. Hence, the user can execute the program both in a forward and a backward manner, searching for the misbehaviour.

Fig. 4.1 shows a screenshot of CauDEr after loading and executing a program.

On one hand, by selecting the Code tab one can see the Core Erlang source code of the program loaded by the user. On the other hand, by selecting the tab State it will be shown the state of the program, where GM represents the global mailbox, i.e., Γ. Then,

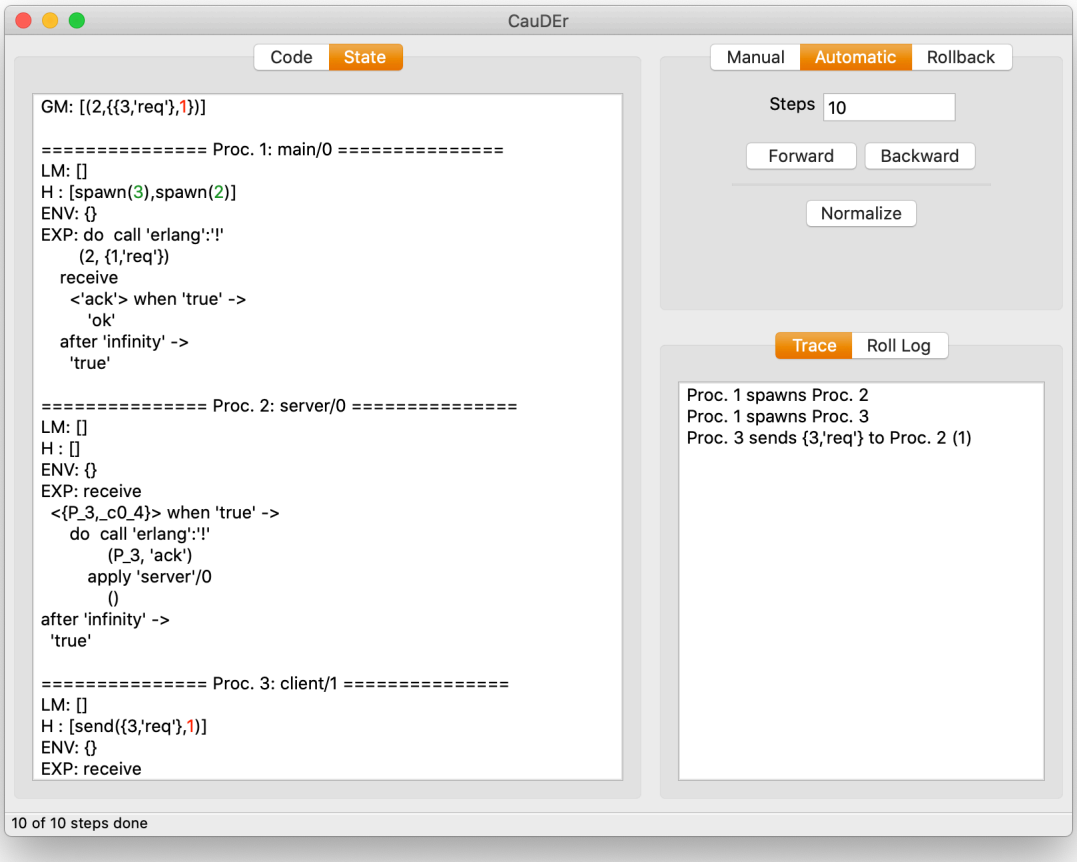


Figure 4.1: CauDEr after starting a program

below GM we have listed all the processes of the system, i.e., Π.

Each process is introduced by its pid and the name of the function followed by the number of arguments required.

For each process we have the local mailbox, represented by LM, the history, represented by H, the environment, represented by ENV and finally EXP, which represents the next expression that needs to be evaluated. For both history and environment we have two modalities, *full* and *concise*, as the names of the modalities suggest one shows every single piece of information about the process, while the other focus only on the interesting facts. More precisely, if we choose the "concise mode" for the history then CauDEr will show us only the application of rules receive, send, and spawn, conversely if we choose the "full history" CauDEr will show us every rule which has been applied. The same goes for the environment, if we choose "concise mode" CauDEr will show us

only the environment for those variables occurring in the current expression, conversely if we choose "full environment" CauDEr will show us every variable of the process.

Now, to execute the program the user can choose between three modes: Manual, Automatic and Rollback. If the user chooses manual then (s)he can select which process has to perform the step, by means of its pid, and if the step has to be forward or backward. The granularity offered by this mode is the finest possible. Here, it is important to stress that the button which performs a step back will be active only when all the consequences of the action, if any, have been undone beforehand.

Of course, executing a program only with the manual mode would be tedious and unproductive, that is why the automatic mode has been implemented as well. While in automatic mode the user can choose the number of steps that (s)he wants to perform and in which direction, then the debugger will perform the steps halting himself once it has performed all of them or in case the system has reached a final state. Here, there are two modalities for the scheduler which will decide who is performing every single step. The first modality is called *Random* mode, i.e., CauDEr will randomly choose an available move of the system and will perform it, the second one is called *Random (Prio. Proc.)* mode, while in this mode CauDEr, if available, will always prefer to perform a step on one of the processes of the system, if no one of the processes can move then a rule *Sched*, if available, will be applied.

The third mode is the rollback mode, which allows the user to perform steps back until a particular state of the program is reached, to do so the user can express which state he wishes to reach, for instance the spawning of a particular process, and then hit the Roll button. After doing so, the debugger will automatically roll the system up to that point, by undoing every action which was a consequence of the original request of the user and finally it will undo the user's request itself, reaching the desired state.

Moreover, in the right bottom corner of the program we have more useful information about the computation of the system. If we choose the Trace tab we will see the trace of the program, i.e., which messages have been sent and to whom, which messages have been received and finally which processes have been spawned and by who. Conversely, by selecting the Roll Log tab, we can choose to see the rollback log which will tell us what relevant actions have been undone while performing the rollback.

Lastly, let us briefly discuss the implementation. CauDEr has been implemented in a modular way, so that it is possible to extract the source files with the semantics and reuse them in different projects. Indeed, each semantics has its own module, then we have a module named cauder.hrl which is the module that contains the descriptions of the tuples used to model the system, the processes, the messages and the traces, then we have modules with utility functions, and modules for the graphic interface.

4.2 Distributed CauDEr

Here, we will introduce the distributed version of CauDEr, which is an extension of the one already presented in the previous section. In Fig. 4.2 one can see a screenshot of Distributed CauDEr while executing a program.

Some differences w.r.t. CauDEr are immediately noticeable: first of all now the state box begins by showing the variable Nodes, which shows which nodes are part of the network, then we can observe that each process is introduced by the information on which node is running, followed by the function's name and the number of arguments required.

Moving on, we can see some changes also in the Trace box, which now contains information about the nodes, for instance when a spawn is performed now the trace will tell us not only which process has performed the spawn but also on which node, also it will tell us which nodes has been started and by whom.

The Roll Log box is changed similarly to the Trace one. The only update that is not visible from the screenshot is under the Rollback mode, indeed following the rollback semantics presented in Section 3.4 now it is also possible to roll back up to the point where a node has been started.

The structure of the debugger is unaltered, indeed since it was designed in a modular way it has been sufficient to modify the modules containing the semantics and the one containing the description of the system (cauder.hrl) to make possible for the debugger to handle the distributed constructs (minor changes have been necessary also in other modules, for instance to update the graphic interface).

4.2.1 Workflow

Usually, a session with CauDEr works as follow: the user loads the Erlang source file, which is compiled into Core Erlang, subsequently the source code is shown to the user in the tab Code, then the user can choose which function will act as an entry point and which ones are its arguments. Successively, after pressing Start, the debugger will switch to the State tab and will show the user the state of the system. From now on, the user is free to execute the program, either by taking advantage of the manual mode or the automatic mode, until the misbehaviour shows up.

When the user detects the misbehaviour then (s)he can start analysing which ones could be the possible causes. To do so, the user can go through the State tab, or through the Trace panel, or even through the console output (which can be found in the same console where CauDEr has been launched). Here, the user is free to take advantage of every feature offered by CauDEr, for instance (s)he could rollback the system up to the point where a faulty message has been exchanged and then proceed in a forward manner again to see what went wrong.

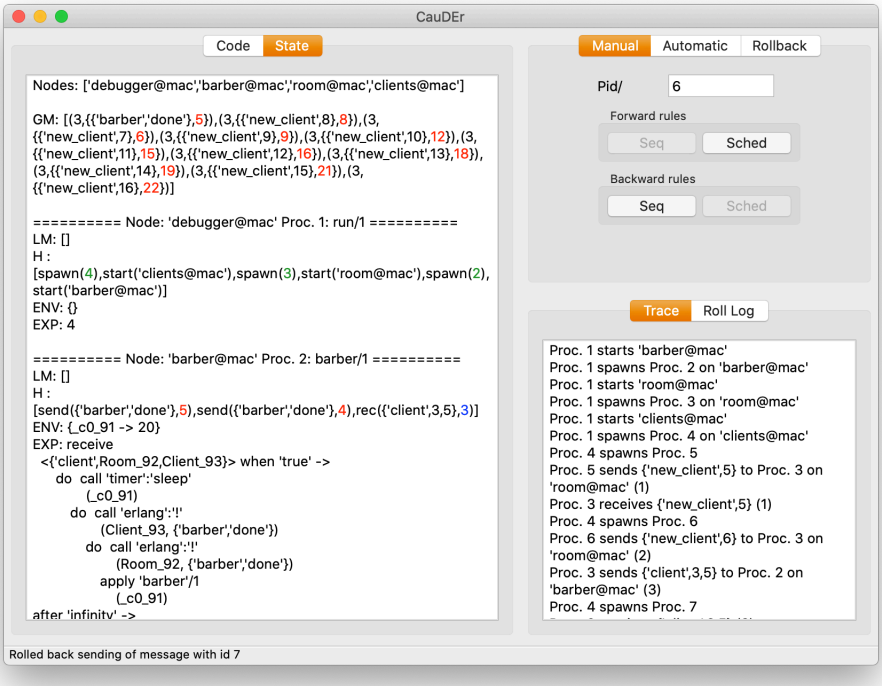


Figure 4.2: Screenshot of distributed CauDEr while executing the sleeping barber program

Unfortunately, there is not a unique solution which works perfectly with every scenario, indeed every program is unique and so are its bugs, therefore the user needs to analyse the situation, correctly understand the logic of the program and eventually do some tests to precisely locate the problem. Experience with concurrent and distributed programming will help during the process of debugging because it means that the user has already been exposed to a wide variety of problems and (s)he should also know how to deal with them.

4.2.2 Finding concurrent bugs with CauDEr

Here, by showing different implementations of the sleeping barber, firstly introduced in [14], we will show some scenarios where bugs typical of concurrent computation arise and how to use CauDEr to detect and solve them. Actually, the version of the sleeping barber that we propose here is slightly different from the canonical one, indeed we made use also of the distributed functions, although they do not play a fundamental role.

The problem is simple, there is a barber shop with one barber who can serve one

client at the time, the shop has a waiting room which can host at most n clients. When a client arrives if the barber is free then he gets served, or else if the barber is busy and if there is enough space left in the waiting room he takes a seat and waits for his turn, otherwise he simply leaves the shop. Once done with a customer the barber checks if there is someone waiting in the waiting room, if yes he picks up the next client and starts to serve him, otherwise he starts to sleep in the chair where he cuts his client's hair, waiting for a new customer to come and wake him up.

Here, the barber, the waiting room and the clients all exist in different nodes, the init function starts each node and then proceeds to spawn the barber process, the waiting room process and the process which creates the clients.

Livelock scenario

Let us consider now the program "sleeping_barber_livelock.erl", which can be found in the GitHub repository. The entry point function takes as argument the number of seats available in the waiting room, after firing the system and performing about 1000 forward steps one can start to see, from the output in the console, that something went wrong, indeed every time that a new customer shows up, he is refused because the barber is busy and there are no available seats for waiting. This scenario should not occur, indeed the barber takes a finite amount of time to serve a client, and we are guaranteed of this because by analysing the trace box we can see that the barber is sending the message $\{\mathsf{barber}, \mathsf{done}\}$ to both the customer and the waiting room. Now, some new client can be refused, we could face the situation of having a full waiting room and the barber busy with a client, but if this happens systematically for every new customer something is wrong.

One can see then, by observing the history of the barber, that the same client keeps appearing on his seat, while this should not happen, indeed once that a customer has received a haircut we should not see him again (it is unlikely that a customer needs a haircut immediately after receiving one). Then, the user can rollback up to the point where the client has been served twice and try to analyse the system in order to understand what went wrong. Indeed, after serving a customer this one should be removed from the waiting room, but by executing the program again in a forward manner it is easy to see that this does not happen, and the list of clients in the waiting room is always the same. Therefore, the problem must be in the line of code which takes care of updating such list, in fact if one checks such line it is possible to notice that instead of updating the list by removing the customer served the previous list is kept.

Bad order messaging

Let us now move to a different scenario, i.e., bad order messaging. According to [15], we can define a bad order messaging bug as a violation of the order of the messages defined

by the protocol. Again, this kind of error is a logical error, therefore we will not observe any crash of the program.

An example of this bug can be found in sleeping_barber_bad_order.erl. After loading and executing about 500 forward steps by observing the state of the second client it is possible to notice that something went wrong. In fact, by observing its state we can see that the process sent a request to the room, then the room answered with the message {room, no_space}, and this seems already weird because unless we set the room capacity to zero then the second client should have found an available seat, but this could be justified by a particular interleaving, i.e., the request of the client got stuck in the global mailbox and it has been scheduled after other requests, although unlikely it is an admissible scenario.

Nonetheless, we receive the confirmation that something went wrong when we observe the local mailbox of the process and we discover the message {barber, done}. Clearly here something went wrong, indeed the client has been told by the room that there was not enough space available, then the client terminated its computation and then it received another message from the barber stating that he has performed the cut. First of all, one could roll back up to the point where the message {room, no_space} has been sent and then check if the room has enough space available or not. By doing so one can see that the room is empty, therefore unless we set its capacity to zero the room should be able to host the client, then by performing some steps forward it is possible to see that the room sends again a message stating that there is not enough space.

With this acquired knowledge, one can start analysing the program, in particular the code managing new clients when the room has still available space and the barber is busy, by doing so it is possible to see that, at line 50, there is an instruction which is clearly misplaced, i.e., the instruction that informs the client that the room is busy, which instead should be placed in the case below.

4.2.3 Finding distributed bugs with CauDEr

Here, we will show a faulty program which shows a problem typical of distributed computation.

The scenario is the following: we have a client which produces data every n milliseconds, the client wants to store the data remotely on a server since it does not have enough space to keep all of them locally. There is a server which acts as a hub, it is able to receive data from the client, then it forwards them to a storage node, receives a confirmation that the storage node has saved a copy of the data locally and then sends an acknowledgement to the client. Each storage node should host only one process and that process is able to store at most m packets from the client, once the limit is reached it proceeds to inform the server that its capacity has been reached (this constraint could be imposed for various reason e.g., performance, lack of space, balancing of the load,

etc). The server holds a list of domains, each domain is followed by a counter, i.e., an integer, and the server also holds an index of the list. Each time it receives a notification from a storage server stating that this last one has reached its capacity it proceeds as follow: if the counter of the domain pointed by the index is a multiple of five it selects the next domain in the list, increases its counter and starts a new storage node on the new domain, if the counter of the domain is not a multiple of five then the server increases it and then starts a storage node on the new domain. Each domain should host at most one node.

Violation of the protocol

Here, we will refer to the program named distributed_storage_node.erl, available in the GitHub repository. The program shows a wrong implementation of the program described above, as always the bug is at the application level. In order to check the program one has to load it and then start the system, consequently it is sufficient to execute about 2000 steps forward to notice that something went wrong, indeed by checking the Trace box one can see a warning: a start has failed because the node was already part of the network. At first glance, the computation has proceeded correctly, but if one then starts to analyse carefully the state of the system it will be evident that the specifics have not been respected, indeed we have two storage processes running in the same node.

This is a violation of the protocol described above, which could cause a slow down of the whole system or, even worse, could obstacle the correct storing of the data. To investigate why this happened one could roll back to the reception of the message {store, full} right before the failed start. After rolling back, since the misbehaviour happened in the server one could proceed with the computation in the manual mode by performing forward steps on the server. After few steps it is possible to observe that after receiving the message the server enters the case where the index of the domain is a multiple of 5, which is correct because we have 5 storage nodes on such domain so far. Now, one should expect the server to select the next domain in the list, increase its counter and perform the start there, instead the server proceeds to perform the start on the selected domain and then it selects the new domain and increases the counter. This malfunction has occurred because few lines of code have been swapped, the server should have first selected a new domain, increased its counter and then proceeded to start a new storage node there.

Chapter 5

Related work

First, the work presented here is an extension of the one presented in [1], therefore many elements are in common. The first element shared with [1] is the approach used to define the semantics: that is the modular approach, indeed here as well our relation is split into expression-level rules and system-level rules. This division allows us to simplify the work of defining the reversible semantics, indeed since the reversible semantics affects only the system semantics if this last one is easier consequently the reversible semantics will be easier. The same modular approach for the semantics of Core Erlang can be found in [13, 16, 17].

In contrast to the modular approach, we have the monolithic approach used in [18]. The monolithic semantics does not split the relation in two levels but there is a single relation which define the behaviour of both the expression-level and the system-level.

Although a different approach for defining the semantics has been used, our work presents also similarities with [18], i.e., the presence of a global mailbox which allows every possible interleaving of the messages even within the same node, which we remind is not the case for real Erlang. However, thanks to this choice extending the semantics to a distributed system has been incredibly natural.

Another work describing the semantics of a distributed Erlang system is [17], where, conversely to what we have done, they have provided every node with its own local mailbox. Moreover, the semantics for message exchanging within the same node that they use does not allow every possible interleaving, as it happens in real Erlang, indeed the preservation of the order is guaranteed. As a consequence of this approach, they needed rules for exchanging messages within the same node and also rules to exchange messages between different nodes, adding a layer of complexity to the definition of the semantics.

Regarding the reversibility part, it has also been studied in the context of CCS, by [19, 20]. The first one is closer to our work, indeed in order to reach previous states of the computation it makes use of memories, while in the second they propose a slightly different syntax for CCS and then they rely on that to retrieve past states of the computation,

without using external devices (like memories).

Another interesting work is [21], in which Actoverse, a causal-consistent debugger for Akka, has been presented. Among the features offered by Actoverse we have: message-oriented breakpoints which allow the user to set breakpoints according to some conditions specified on the messages, rollback, state inspection, message timeline and session replay.

The last one is particularly interesting, session replay allows one to replay the execution of a program given the log of a computation. Such a feature is particularly helpful when one detects a bug, fix it, and then wants to verify that such a situation does not happen again.

Another interesting work which studied session replay, in the context of Erlang, is [13], in such work the "session replay" is extended with the dual notion of causal consistency, which allows one to replay the execution of the system up to an action (conversely to what happens in rollback mode), including *all and only* its *causes*. The possibility to replay only action in which we are interested, and their causes, allows us to focus only in the part of the computation of the system that we are interested in, leaving out actions which are not related, therefore increasing the chances that the user will detect the misbehaviour. Moreover, with [13], as already mentioned, we share the approach used to define the rollback semantics.

Chapter 6

Conclusions and future work

In this work we have extended the reversible semantics for Erlang presented in [1] by adding functions meant for distributed computing. Adding these functions has required to introduce new rules in the reversible semantics and to re-think some of the already existent ones, while for the rollback semantics we followed the approach proposed in [13]. Moreover, we extended the notion of system by adding a variable to keep track of which nodes are part of the network and we also extended the tuple defining a process, adding an atom which represents on which node the process is running.

Regarding the message exchanging although we were moving from a non-distributed environment to a distributed one no changes were required, indeed the global mailbox used in [1] was already able to model every possible interleaving (including the distributed ones). Then, after introducing the semantics we have proved that they enjoy desirable properties, like the loop lemma, the square lemma, and causal consistency for the reversible semantics and then we also proved desirable properties for the rollback semantics, like minimality. As final step we have extended the already existent implementation of CauDEr by adding such constructs and updating the graphic interface accordingly.

As for future developments, plenty of directions still have to be explored, first of all there are still a plethora of functions that one could add to CauDEr, then as the number of supported functions increase it will increase also the number of dependencies that each action will have, therefore it would be helpful to have a tool which shows what actions need to be undone before undoing a targeted step. Already by adding these three functions meant for distributed programming we have examples of subtle dependencies, for instance in order to undo a `nodes` performed in the early stages of a system it might be necessary to undo the creation of several nodes and processes running on them, sometimes such relationship might not be obvious. As of now, CauDEr only provides the user with a piece of binary information, if the action can be undone then the backward button in the manual mode is active, otherwise it is not. Therefore, the assistance of a graphical tool which shows what actions need to be undone in the form

of a tree would help the user to have a deeper understanding of the system.

Another line of research could focus its efforts in investigating ways to reduce the amount of space required to save the histories, indeed if we want CauDEr to be exploited also for real programs it is essential that we improve its performances.

Appendix A

Auxiliary functions

Here, the interested reader will find both an informal and a formal description of the auxiliary functions used in Chapter 3 to describe the reversible semantics.

exists

Given a pid and a list of processes, returns true if the pid represents a real process.
$\mathsf{exists}(\mathsf{p}', [\,]) = \mathsf{false}$
$\mathsf{exists}(\mathsf{p}', \langle \mathsf{node}, \mathsf{p}, \mathsf{h}, (\theta, \mathsf{e}), \mathsf{q} \rangle \mid \Pi) = \mathsf{true} \;\; \mathsf{if} \;\; \mathsf{p} = \mathsf{p}'$
$\mathsf{exists}(\mathsf{p}', \langle \mathsf{node}, \mathsf{p}, \mathsf{h}, (\theta, \mathsf{e}), \mathsf{q} \rangle \mid \Pi) = \mathsf{exists}(\mathsf{p}', \Pi) \;\; \mathsf{otherwise}$

reads

Given a node and a list of processes, returns the processes that have performed a nodes when node was part of the network.
$\mathsf{reads}(\mathsf{node}, [\,]) = [\,]$
$\mathsf{reads}(\mathsf{node}, \langle \mathsf{node}', \mathsf{p}, \mathsf{h}, (\theta, \mathsf{e}), \mathsf{q} \rangle \mid \Pi) = \mathsf{p} + \mathsf{reads}(\mathsf{node}, \Pi) \;\; \mathsf{if} \;\; \mathsf{has\_read}(\mathsf{node}, \mathsf{h})$
$\mathsf{reads}(\mathsf{node}, \langle \mathsf{node}', \mathsf{p}, \mathsf{h}, (\theta, \mathsf{e}), \mathsf{q} \rangle \mid \Pi) = \mathsf{reads}(\mathsf{node}, \Pi) \;\; \mathsf{otherwise}$

has_read

Given a process' history and a node, returns the pid of the process if node has been read by one of the nodes performed.
$\mathsf{has\_read}(\mathsf{node}, [\,]) = \mathsf{false}$
$\mathsf{has\_read}(\mathsf{node}, \mathsf{nodes}(\theta, \mathsf{e}, \Omega) : \mathsf{h}) = \mathsf{true} \;\; \mathsf{if} \;\; \mathsf{node} \in \Omega$
$\mathsf{has\_read}(\mathsf{node}, \mathsf{nodes}(\theta, \mathsf{e}, \Omega) : \mathsf{h}) = \mathsf{has\_read}(\mathsf{node}, \mathsf{h}) \;\; \mathsf{if} \;\; \mathsf{node} \notin \Omega$
$\mathsf{has\_read}(\mathsf{node}, \mathsf{op}(...) : \mathsf{h}) = \mathsf{has\_read}(\mathsf{node}, \mathsf{h})$

procs

Given a node and a list of processes, returns the list of processes running in node.
$\mathsf{procs}(\mathsf{node}, [\,]) = [\,]$
$\mathsf{procs}(\mathsf{node}, \langle \mathsf{node}', \mathsf{p}, \mathsf{h}, (\theta, \mathsf{e}), \mathsf{q} \rangle \mid \Pi) = \mathsf{p} + \mathsf{procs}(\mathsf{node}, \Pi) \;\; \text{if } \mathsf{node} = \mathsf{node}'$
$\mathsf{procs}(\mathsf{node}, \langle \mathsf{node}', \mathsf{p}, \mathsf{h}, (\theta, \mathsf{e}), \mathsf{q} \rangle \mid \Pi) = \mathsf{procs}(\mathsf{node}, \Pi) \;\; \text{otherwise}$

proc_parent

Returns the parent of a given process.
$\mathsf{proc\_parent}(\mathsf{p}, [\,]) = \mathsf{error}$
$\mathsf{proc\_parent}(\mathsf{p}, \langle \mathsf{node}, \mathsf{p}', \mathsf{h}, (\theta, \mathsf{e}), \mathsf{q} \rangle \mid \Pi) = \mathsf{p}' \;\; \text{if } \mathsf{has\_spawned}(\mathsf{p}, \mathsf{h})$
$\mathsf{proc\_parent}(\mathsf{p}, \langle \mathsf{node}, \mathsf{p}', \mathsf{h}, (\theta, \mathsf{e}), \mathsf{q} \rangle \mid \Pi) = \mathsf{proc\_parent}(\mathsf{p}, \Pi) \;\; \text{otherwise}$

has_spawned

Given a process' pid and the history of a process, returns true if in the history the spawn of the process is recorded, false otherwise.
$\mathsf{has\_spawned}(\mathsf{p}, [\,]) = \mathsf{false}$
$\mathsf{has\_spawned}(\mathsf{p}, \mathsf{spawn}(\theta, \mathsf{e}, \mathsf{node}, \mathsf{p}') : \mathsf{h}) = \mathsf{true} \;\; \text{if } \mathsf{p} = \mathsf{p}'$
$\mathsf{has\_spawned}(\mathsf{p}, \mathsf{spawn}(\theta, \mathsf{e}, \mathsf{node}, \mathsf{p}') : \mathsf{h}) = \mathsf{has\_spawned}(\mathsf{p}, \mathsf{h}) \;\; \text{if } \mathsf{p} \neq \mathsf{p}'$
$\mathsf{has\_spawned}(\mathsf{p}, \mathsf{op}(...) : \mathsf{h}) = \mathsf{has\_spawned}(\mathsf{p}, \mathsf{h})$

node_parent

Returns the parent of a given node.
$\mathsf{node\_parent}(\mathsf{node}, [\,]) = \mathsf{error}$
$\mathsf{node\_parent}(\mathsf{node}, \langle \mathsf{node}', \mathsf{p}', \mathsf{h}, (\theta, \mathsf{e}), \mathsf{q} \rangle \mid \Pi) = \mathsf{p}' \;\; \text{if } \mathsf{has\_started}(\mathsf{node}, \mathsf{h})$
$\mathsf{node\_parent}(\mathsf{node}, \langle \mathsf{node}', \mathsf{p}', \mathsf{h}, (\theta, \mathsf{e}), \mathsf{q} \rangle \mid \Pi) = \mathsf{node\_parent}(\mathsf{node}, \Pi) \;\; \text{otherwise}$

has_started

Given a node and the history of a process, returns true if in the history the start of node is recorded, false otherwise.
$\mathsf{has\_started}(\mathsf{node}, [\,]) = \mathsf{false}$
$\mathsf{has\_started}(\mathsf{node}, \mathsf{start}(\theta, \mathsf{e}, \mathsf{success}, \mathsf{node}') : \mathsf{h}) = \mathsf{true} \;\; \text{if } \mathsf{node} = \mathsf{node}'$
$\mathsf{has\_started}(\mathsf{node}, \mathsf{start}(\theta, \mathsf{e}, \mathsf{success}, \mathsf{node}') : \mathsf{h}) = \mathsf{has\_started}(\mathsf{node}, \mathsf{h}) \;\; \text{if } \mathsf{node} \neq \mathsf{node}'$

$\mathsf{has\_started}(\mathsf{node}, \mathsf{op}(...) : \mathsf{h}) = \mathsf{has\_spawned}(\mathsf{p}, \mathsf{h})$

tried_starts

Given a node and a list of processes, returns the list of processes that tried to start node and failed because node was already part of the network.
$\mathsf{tried\_starts}(\mathsf{node}, [\,]) = [\,]$
$\mathsf{tried\_starts}(\mathsf{node}, \langle \mathsf{node}, \mathsf{p}, \mathsf{h}, (\theta, \mathsf{e}), \mathsf{q} \rangle | \Pi) = \mathsf{p} + \mathsf{tried\_starts}(\mathsf{node}, \Pi) \;\; \text{if } \mathsf{has\_tried\_start}(\mathsf{node}, \mathsf{h})$
$\mathsf{tried\_starts}(\mathsf{node}, \langle \mathsf{node}, \mathsf{p}, \mathsf{h}, (\theta, \mathsf{e}), \mathsf{q} \rangle | \Pi) = \mathsf{tried\_starts}(\mathsf{node}, \Pi) \;\; \text{otherwise}$

has_tried_start

Given a node and a process' history, returns true if the process tried to start node and failed.
$\mathsf{has\_tried\_start}(\mathsf{node}, [\,]) = \mathsf{false}$
$\mathsf{has\_tried\_start}(\mathsf{node}, \mathsf{start}(\theta, \mathsf{e}, \mathsf{failure}, \mathsf{node}') : \mathsf{h}) = \mathsf{true} \;\; \text{if } \mathsf{node} = \mathsf{node}'$
$\mathsf{has\_tried\_start}(\mathsf{node}, \mathsf{start}(\theta, \mathsf{e}, \mathsf{failure}, \mathsf{node}') : \mathsf{h}) = \mathsf{has\_tried\_start}(\mathsf{node}, \mathsf{h}) \;\; \text{if } \mathsf{node} \neq \mathsf{node}'$
$\mathsf{has\_tried\_start}(\mathsf{node}, \mathsf{op}(...) : \mathsf{h}) = \mathsf{has\_tried\_start}(\mathsf{node}, \mathsf{h})$

Bibliography

[1] Ivan Lanese, Naoki Nishida, Adrián Palacios, and Germán Vidal. A theory of reversibility for Erlang. *Journal of Logical and Algebraic Methods in Programming*, 100:71–97, November 2018.

[2] Francesco Cesarini. *Erlang programming.* O'Reilly, Beijing Cambridge Mass, 2009.

[3] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, page 235245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[4] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs Journal*, 30(3):202–210, 2005.

[5] Ulf Wiger. Industrial-strength functional programming: Experiences with the ericsson AXD301 project. *IFL'00, Aachen*, 2000.

[6] Christopher Lutz. Janus: a time-reversible language. *Letter to R. Landauer.*, 1986.

[7] R. Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 5(3):183–191, July 1961.

[8] C. H. Bennett. Logical reversibility of computation. *IBM Journal of Research and Development*, 17(6):525–532, November 1973.

[9] W. Harrison and C. Cook. Insights on improving the maintenance process through software measurement. In *Proceedings. Conference on Software Maintenance 1990.* IEEE Comput. Soc. Press.

[10] Vincent Danos and Jean Krivine. Reversible communicating systems. In *CONCUR 2004 - Concurrency Theory*, pages 292–307. Springer Berlin Heidelberg, 2004.

[11] Richard Carlsson. An introduction to core Erlang. In *In Proceedings of the PLI01 Erlang Workshop*, 2001.

[12] Ivan Lanese, Iain Phillips, and Irek Ulidowski. An axiomatic approach to reversible computation. In Jean Goubault-Larrecq and Barbara König, editors, *Foundations of Software Science and Computation Structures*, pages 442–461, Cham, 2020. Springer International Publishing.

[13] Ivan Lanese, Adrián Palacios, and Germán Vidal. Causal-consistent replay debugging for message passing programs. In Jorge A. Pérez and Nobuko Yoshida, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 39th IFIP WG 6.1 International Conference, FORTE 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17-21, 2019, Proceedings*, volume 11535 of *Lecture Notes in Computer Science*, pages 167–184. Springer, 2019.

[14] Edsger W. Dijkstra. Cooperating sequential processes. In *The Origin of Concurrent Programming*, pages 65–138. Springer New York, 1968.

[15] Carmen Torres Lopez, Stefan Marr, Elisa Gonzalez Boix, and Hanspeter Mössenböck. A study of concurrency bugs and advanced development support for actor-based programs. In Alessandro Ricci and Philipp Haller, editors, *Programming with Actors*, volume 10789 of *Lecture Notes in Computer Science*, pages 155–185. Springer, 2018.

[16] Rafael Caballero, Enrique Martin-Martin, Adrián Riesco, and Salvador Tamarit. A declarative debugger for sequential erlang programs. In Margus Veanes and Luca Viganò, editors, *Tests and Proofs*, pages 96–114, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[17] Koen Claessen and Hans Svensson. A semantics for distributed Erlang. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Erlang*, ERLANG '05, page 7887, New York, NY, USA, 2005. Association for Computing Machinery.

[18] Hans Svensson, Lars-rAke Fredlund, and Clara Benac Earle. A unified semantics for future Erlang. In *Proceedings of the 9th ACM SIGPLAN Workshop on Erlang*, Erlang '10, page 2332, New York, NY, USA, 2010. Association for Computing Machinery.

[19] Vincent Danos and Jean Krivine. Reversible communicating systems. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR 2004 - Concurrency Theory*, pages 292–307, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[20] Iain Phillips and Irek Ulidowski. Reversing algebraic process calculi. In Luca Aceto and Anna Ingólfsdóttir, editors, *Foundations of Software Science and Computation Structures*, pages 246–260, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[21] Kazuhiro Shibanai and Takuo Watanabe. Actoverse: A reversible debugger for actors. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE 2017, page 5057, New York, NY, USA, 2017. Association for Computing Machinery.