Vrije Universiteit Brussel

Faculteit Wetenschappen en Bio-Ingenieurswetenschappen
Departement Computerwetenschappen
Software Languages Lab

Combining the Actor model with Software Transactional Memory

Proefschrift ingediend met het oog op het behalen van de titel Master in de Ingenieurswetenschappen: Computerwetenschappen, door

Christophe De Troyer

Promotor: Prof. Dr. Wolfgang De Meuter
Begeleiders: Dr. Joeri De Koster
Janwillem Swalens

Juni 2015

Vrije Universiteit Brussel

Faculty of Science and Bio-Egineering Sciences
Departement Of Computer Science
Software Languages Lab

Combining the Actor model with Software Transactional Memory

Dissertation submitted in partial fulfillment of the requirements for the degree of Master in Engineering: Computer Science, by

Christophe De Troyer

Promotor: Prof. Dr. Wolfgang De Meuter
Advisors: Dr. Joeri De Koster
Janwillem Swalens

June 2015

Abstract

Multi-core processors have become ubiquitous in modern hardware. However, efficiently exploiting the parallelism offered by these architectures still remains a challenge. Concurrency models allow the programmer to structure his code in logically separated concurrent tasks. How the different operations are structured into logically separated concurrent tasks and how these different tasks are synchronised heavily depends on the concurrency model that is used. Concurrency models come in a wide array of flavours and each of them is well suited for expressing a particular type of concurrency. The starting hypothesis of this thesis is that application developers can benefit from combining different concurrency models to parallelise the different parts of their application.

This thesis investigates the combination of two concurrency models, namely the actor model and software transactional memory. Both models offer certain semantic guarantees that allow programmers to write *lock-free* code. However, the former model does not allow shared mutable state by design which is one of its strengths but limits expresiveness. This thesis investigates whether it is possible to combine the actor model with software transactional memory into a conglomerate model that still satisfies the semantic properties that both models guarantee when used in isolation, and does not introduce additional liveness or safety issues.

We provide a unified list of semantic properties of both software transactional memory and actor languages which will form the correctness criteria for our result. The violation of these properties by the ad hoc combination is illustrated by means of several examples. For both models a library is implemented in Clojure as an experimental platform.

We compose a novel formal semantics that show that the actor model can be safely combined with software transactional memory while still ensuring the semantic properties that each model guarantees if used in isolation. As a proof of concept we provide an implementation of the formal semantics in Clojure under the name Actransors which is comprised of a modified version of the two aforementioned libraries.

We validate our approach by translating an existing Clojure application into a version that uses Actransors which results in an equivalent application with added expressivity. Finally we informally prove that the semantic properties of both models remain intact.

Samenvatting

Multikernprocessoren zijn alomtegenwoordig in moderne computersystemen. Het parallellisme van deze machines efficiënt uitbuiten blijft echter een hele uitdaging. Een concurrency model is een programmeermodel dat abstracties introduceert die de programmeur toelaten op een intuïtieve manier gebruik te maken van het potienteel van multikernprocessoren.

Elk programmeermodel heeft echter zijn beperkingen. Om ten volle gebruik te kunnen maken van een model is het essentieel om de zwaktes en sterktes ervan te kennen alsook de mogelijkheden van het model.

Programmeermodellen laten de programmeur toe om zijn code in logisch gescheiden onafhankelijke taken te verdelen. Hoe de verschillende operaties gestructureerd worden in logisch gescheiden taken en hoe deze taken gesynchroniseerd worden hangt af van het programmeermodel dat toegepast wordt. Er is de dag van vandaag een brede waaier aan programmeermodellen beschikbaar en elk van deze modellen is uitermate geschikt om een specifieke vorm van parallel programmeren uit te drukken. Het uitgangspunt van deze thesis is dat ontwikkelaars er baat bij hebben om niet een enkel programmeermodel toe te passen maar een combinatie van meerdere programmeermodellen tegelijk in eenzelfde programma.

Dit proefschrift zal de combinatie van twee specifieke programmeermodellen onderzoeken, namelijk het actor model en “Software Transactional Memory”. Beide modellen hebben aantrekkelijke eigenschappen die het interessante computationele modellen maken om multikernprocessoren uit te buiten. Beide modellen laten de programmeur ook toe om “lock-free” programma’s te schrijven. Het ontwerp van het actor model staat echter niet toe om geheugen te delen tussen actoren. Dit is een van de sterktes van het model maar dit beperkt ook de uitdrukkingskracht van het model. Dit proefschrift zal onderzoeken of het mogelijk is om een combinatie te maken van het actor model en “Software Transactional Memory” die nog steeds de semantische eigenschappen van beide modellen garandeert en geen nieuwe problemen introduceert die de correctheid schaden.

Dit proefschrift verzamelt een lijst van semantische eigenschappen gebaseerd op bestaande literatuur. Dez die een verzameling is van semantische eigenschappen van elk van beide modellene lijst zal als basis dienen om ons voorgesteld model te staven.We tonen via enkele voorbeelden aan dat een triviale combinatie tot problemen leidt. Om te experimenteren voorzien we een implementatie van beide modellen in Clojure, een programmeertaal voor de Java virtuele machine.

We geven een formele semantiek die weldegelijk aantoont dat een combinatie van beide modellen mogelijk is. De semantiek garandeert de beoogde semantische eigenschappen. Als proof of concept stellen we een implementatie voor van onze formele semantieken in Clojure onder de naam “Actransors”. Deze bibliotheek voor Clojure is een aangepaste versie van de voorgenoemde libraries.

Om onze bevindingen te valideren hebben we een bestaande applicatie vertaald zodat deze gebruik maakt van Actransors. Het resultaat is een equivalente applicatie die expressiever is. Als laatste tonen we aan dat de vooropgestelde semantische eigenschappen van beide modellen bewaard blijven.

Acknowledgements

First of all I would like to thank Wolfgang De Meuter for promoting this thesis. Furthermore I would like to express gratitude to Joeri De Koster and Janwillem Swalens for their guidance during this disseratation. I would like to thank them for never saying “no” if asked if they had a moment to spare and the invaluable advice they have given me throughout the year.

I would also like to thank all the members of the Software Languages Lab who have helped me out with various comments and support and the people of the `#infogroep` irc chat for the interesting conversations. Finally a big thank you goes out to the coffee maker at the Software Languages Lab for tirelessly providing me with hot cups of caffeine-induced beverages.

I would like to express my gratitude to my family for always supporting me through my studies. Without them I am convinced that I would not have gotten to the point I am today.

Finally I would like to thank my girlfriend Kimber for making sure I didnt turn into a computer myself and the endless support she has given me throughout the last three years of my studies. Thank you for always believing in me.

Contents

1 Introduction **7**
1.1 Combining Concurrency Models . . . 8
1.1.1 Problem Statement . . . 9
1.2 Contributions . . . 9
1.3 Roadmap . . . 10

2 Pre-Science **11**
2.1 Actors . . . 11
2.1.1 Agha Actors . . . 12
2.1.2 Erlang Actors . . . 13
2.1.3 Semantic Properties . . . 14
2.2 Transactional Memory . . . 17
2.2.1 Hardware Transactional Memory . . . 18
2.2.2 Software Transactional Memory . . . 19
Introduction . . . 19
Multi Version Concurrency Control . . . 20
2.2.3 Semantic Properties . . . 21
Serializability . . . 22
1-Copy Serializability . . . 24

3 Combining Actors and STM **26**
3.1 Inadequacies of the Actor Model . . . 26
3.1.1 Shared State in Actors . . . 26
3.1.2 Motivating Example . . . 29
Syntax . . . 29
Apples and Oranges . . . 31
3.2 Ad-hoc Combination of the Actor Model and STM . . . 33
3.2.1 Motivating example revisited . . . 33
Apples and Oranges with STM . . . 34
3.2.2 Discrepancies of the the Ad-Hoc Combination . . . 35
Sending Messages . . . 35
Receiving Messages . . . 35
3.3 Conclusion . . . 38

4 Actransors **39**
4.1 Actransor Concepts . . . 39
4.1.1 Context . . . 39
4.1.2 Dependency . . . 40
Dependency Propagation . . . 40
Dependency Gathering . . . 40

4.2 Discrepancies resolved 40
4.3 Conclusion 46

5 Operational Semantics 47
5.1 Introduction 47
5.1.1 Concept 47
5.1.2 Notation Conventions 49
5.2 Grammar 49
5.2.1 Values 49
5.2.2 Expressions 50
Standard Expressions 50
Actor Expressions 50
Transaction Terms 51
Syntactic Sugar 51
5.2.3 Semantic Domains 51
Messages 51
Actors 52
Transactional Memory Entities 52
Configuration Entity 53
5.2.4 Auxiliary Operations 53
Operations on Actors 54
Operations on Transactions 54
Operations on Messages 54
Operations on In-boxes 55
Operations on Transactional Variables 55
Operations for receive 56
5.3 Rules 56
5.3.1 Sending 56
5.3.2 Receiving 57
5.3.3 End Receive 59
5.3.4 Transactions 60
5.3.5 Transactional Variables 61
5.3.6 Committing 63

6 Implementation 65
6.1 Multi-version Concurrency Control STM 65
6.2 Actor library 68
6.2.1 Syntax 68
6.2.2 Details 69
6.3 Combination of Both Models 70
6.3.1 STM 70
6.3.2 Actor model 72
6.4 Caveats 73
6.5 Conclusion 73

7 Evaluation **74**
7.1 Ant Simulation . . . 74
7.1.1 Implementation . . . 74
7.1.2 Ant Simulation with Actransors . . . 76
7.1.3 Conclusion . . . 77
7.2 Semantics . . . 77
7.2.1 Actors . . . 77
7.2.2 Software Transactional Memory . . . 78
7.3 Conclusion . . . 79

8 Related Work **80**
8.1 Transactors . . . 80
8.1.1 Introduction . . . 80
8.1.2 Transactors . . . 80
Stabilizing and Checkpointing . . . 80
Dependencies . . . 81
Rollback . . . 81
Example . . . 81
8.1.3 Conclusion . . . 82
8.2 Communicating Memory Transactions . . . 83
8.2.1 Introduction . . . 83
8.2.2 Message Passing . . . 84
8.2.3 Software Transactional Memory . . . 84
Moverness . . . 84
8.2.4 Committing Transactions . . . 86
8.2.5 Conclusion . . . 87

9 Conclusion **89**
9.1 Problem Statement Revisited . . . 89
9.2 Contributions . . . 90
9.3 Semantics in a nutshell . . . 90
9.4 Future Work . . . 91
9.4.1 Validation . . . 91
9.4.2 Formal Semantics . . . 91
9.4.3 Implementation . . . 91
9.5 Conclusion . . . 92
References . . . 93

1 Introduction

Multi-core processors have become ubiquitous in modern hardware. However, efficiently exploiting the parallelism offered by these architectures still remains challenging. Concurrency models are models of computation that allow the programmer to abstract away of the notion of cores and threads and write more intuitive code while still exploiting parallelism or concurrency.

Each concurrency model however has its limitations. To fully exploit the capabilities of a concurrency model one has to understand its limitations as well as its capabilities.

Concurrency models allow the programmer to structure his code in logically separated concurrent tasks. How the different operations are structured into logically separated concurrent tasks and how these different tasks are synchronised heavily depends on the concurrency model that is used. Concurrency models come in a wide array of flavours and each of them is well suited for expressing a particular type of concurrency. The starting hypothesis of this thesis is that application developers can benefit from combining different concurrency models to parallelise the different parts of their application. For example, futures are easy to use for forking off a disk write. Atomic variables (e.g., `AtomicInteger` in `java.util.concurrent.atomic`) allow the programmer to safely share a single variable across multiple threads and avoid race conditions on that variable.

In a sequential program there is only one path of execution. A program is executed as a single stream of operations. Concurrent programming allows the programmer to define multiple streams of operations in which each of those streams executes as a single stream of operations. When a computer executes these streams they are not all necessarily executed on an individual unit of computation (processor core). Hence, individual execution steps can be interleaved. This is the main source of nondeterminism in multi-core programming. Due to the interleaving of execution streams *race conditions* can occur. Two or more threads modifying shared memory in an uncoordinated fashion. The thread depends on a proper interleaving with other threads in order for the entire program, i.e., all execution streams, to produce a correct result. When the programmer writes code that manipulates or reads from shared memory no guarantees are made about data coherence spanning multiple statements in a single execution stream. In general, concurrent programs are nondeterministic and thus hard to reason about. Programmers can introduce a synchronisation mechanism such as locks to prune that non-determinism. However,

these synchronisation mechanisms typically can lead to issues of their own such as deadlocks. (Lee, 2006).

1.1 Combining Concurrency Models

Any given problem that requires concurrency control has a particular concurrency model that is best suited to solve it and others that are less well suited. For example, if one wishes to share an integer across threads as a counter an atomic variable would be well suited. Programmers do not need to consider using explicit synchronisation when using the atomic variable since it publishes an interface to increment, decrement, read and possibly compare and swap. If one is writing a program using the actor model it is not unthinkable to use the atomic variable model as well. This would be a combination of the actor model and atomic variables.

Suppose one is writing an email client. One could reason that the core of the application should be modelled as a continuous thread and a seperate thread for the interface rendering. The reason being that the latter thread should not be interrupted by other computations in order to maintain a smooth user experience. To update the in-box of the client a lot of communication with the mail server is required. Such communication could be done inside a future. When the new data has arrived one might use a transaction to update existing items in the mailbox their read status (read or unread). A background task that continuously checks for new emails could be modelled as an actor that receives messages when a forced update is required. Other parts of the program such as a calendar, a todo list, etc. could also be modelled by an actor. If one of the widgets were to fail and crash the rest of the program would remain unaffected.

As we will show below, some concurrency models are better suited to express certain tasks than the actor model. However, different concurrency models are not always integrated properly with one another(Swalens, Marr, De Koster, & Van Cutsem, 2014).

Tasharofi, Dinges, and Johnson (2013) have studied a corpus of programs that are written in the Scala language, an actor language for the JVM. The corpus of programs is selected out of all the publicly available Scala and Akka programs on the public code repository website GitHub. As Tasharofi et al. (2013) discovers, the actor model in its pure form (i.e., no shared memory or other concurrency paradigms) does not really meet all the requirements programmers have of a language or framework such as Scala or Akka. Hence, programmers tend to mix concurrency models that are better suited for the task at hand. One of the reasons that this happens in Scala for example, is the seamless interoperability with Java. It may be easier and more comfortable to use another concurrency paradigm such as a future than to write an entire actor for a simple job, especially when no extra effort is required for interoperability. In Clojure one can use futures in combination with any other available concurrency model, however, they are an abstraction of pure Java futures. This gives the guest language (Clojure) more control over the execution of the future in the host language (Java) and allows the language or library designer to increase composability.

Tasharofi et al. (2013) have shown that the main motivations for combining concurrency models with the actor model can be divided into three categories. Shortcomings of the actor libraries, shortcomings of the actor model or inadequate programmer experience. One of the reasons amongst the first category are managing I/O actions efficiently. The programmers are unsure about how well Scala handles big data transfers between actors as the actor model is designed to transfer small messages by design. Note that enforcing messages to be send by value (*safe message passing*) would require deep copies of big data. Hence, they chose to handle file transfers using *Runnable*s (*java.lang* library) to avoid blocking an actor thread too long that is running on the actor thread pool.

Ideally one would like to obtain the benefits of the actor model such as scalability and safety but still be able to use other concurrency models as they wish without breaking the semantics of the actor model or the model it is combined with.

In general, combining concurrency models with each other requires careful reasoning and additional bookkeeping by the runtime. Given that each model has a set of semantic properties that guarantee certain properties about the concurrency model does not mean that these properties still hold when one model is combined with another. Swalens et al. (2014) have studied the concurrency models present in Clojure. Clojure has a lot of concurrency models built-in: transactions, threads, CSP, channels, agents, futures, atomic variables, etc. Swalens et al. discuss the combination of each model and check if they fulfill the requirements of liveness and safety(Lamport, 1977). Each combination of models is checked to see if it introduces *additional* liveness or safety issues. That is, a model itself does not necessarily guarantee liveness or safety but a combination does not preserve the guarentees made by each model in isolation. Liveness is the criterion that a program, given a correct input, does not end up in a deadlock or livelock. Safety on the other hand guarantees that a program, given a correct input produces a correct output.

1.1.1 Problem Statement

The goal of this thesis is to investigate the properties of a combination of two concurrency models, namely the Actor Model and Software Transactional Memory. As a stand alone concurrency model the Actor Model has a number of safety guarantees that can be beneficial for the application developer. The actor model avoids any race conditions on memory values by disallowing shared state. Furthermore, given that the model operates by means of message passing it is a *lock-free* model. No explicit locking mechanisms have to be provided by the programmer. However, the downside is that it offers up expressiveness, especially concerning access to shared state (See subsection 3.1.1). STM allows for threads to modify a set of variables by means of a transaction. This model is lock-free as well and as such avoids race conditions and deadlocks. These properties make it an attractive model to model shared state in concurrent programs. For the given reasons it can be beneficial to integrate STM with the Actor model as a way to represent shared state.

However, an ad hoc integration of both concurrency models does not necessarily preserve the properties of both individual models. The goal of this thesis is to design a conglomerate model that combines both models in such a way that none of the properties of the individual models are invalidated.

1.2 Contributions

- We provide a unified list of semantic properties for the actor model and STM based on existing literature. Furthermore, existing literature does not agree on a single definition for the semantic property *serializability.* Consequently we have based ourselves on an existing formal model(Bernstein & Goodman, 1983) and have defined *serializability* and *1-Copy serializabilty* using this model based on the work of Papadimitriou (1979).

- We provide an informal proof by example that an ad hoc combination of the actor model with software transactional memory invalidates semantic properties of both models by pinpointing problem scenarios.

- We informally provide a solution to the problems that arise when combining the actor

model and software transactional memory. The informal solution is formally described using formal semantics.

- We provide an implementation in Clojure for both the actor model and software transactional memory which provide a platform for experimentation.

- We provide an implementation in Clojure of our conglomerate model under the name Actransors which is a proof of concept of our formal semantics.

1.3 Roadmap

Chapter 2 discusses the actor model and software transactional model in detail. The chapter will introduce each model and discuss their semantic properties.

Chapter 3 discusses the inadequacies of the actor model and how shared state can be modelled in a pure actor language. Furthermore we will also discuss the problems that arise when the actor model is combined with transactional memory in an ad-hoc fashion.

Chapter 4 discusses the proposed solution at a higher level. It also serves as an introduction to the formal semantics given in chapter 5.

Chapter 6 discusses the implementation of the formal semantics in Clojure. The chapter focusses on where the semantic and the implementation diverge and explains certain shortcomings of the implementation with respect to the formal semantic.

Chapter 7 introduces the evaluation of the implementation and the semantic. A small program will be written using the provided actor library and will then be translated to use the software transactional memory library as well.

Chapter 9 concludes this thesis and elaborates on future work.

2
Pre-Science

In this chapter we will discuss the actor model and software transactional memory in more detail. We will discuss different actor models and how they came into existence. For software transactional memory we will shortly discuss hardware transactional memory, the first definition of software transactional memory followed by the implementation found in Clojure: Multi Version Concurrency Control STM. We will also give a short overview of some of the most important semantic properties of each model.

2.1 Actors

The Actor model is a computational model first formally introduced by Hewitt, Bishop, and Steiger (1973) as an architecture for artificial intelligence. The main goal of Hewitt et al. was to exploit the architectures that consisted of multiple processing units that each have local memory and thus make it possible to achieve high parallelism. The first actual implementation of the Actor model was by Smith and Hewitt (Smith & Hewitt, 1975) as well; the language named PLASMA. Since the actor model has been invented it has reincarnated in a wide array of variations. Some of the more popular languages include SALSA, Scala(Haller & Odersky, 2009), E and Erlang. The aforementioned languages are all languages designed from the ground up. There are also *library languages*. These are languages that are built atop of another language (e.g., Java or C#) and can be used as a library.

An Actor is a unit of computation that works in isolation. It is an autonomous unit that operates concurrently with other Actors and has its own state that is not accessible to other Actors. The only means of communicating with an Actor or manipulating an Actor's behavior is by sending asynchronous messages to it. This means that the sender of the message will not wait until the message has arrived. The messages will be inserted into the in-box of the receiving actor.

Since all communication with Actors happens asynchronously, messages need to be buffered at the receiver in order for the receiver to process them at his own pace. Each message will be buffered until the receiving actor processes the message from his in-box. An Actor will

continuously take messages from this in-box and execute behavior that is determined by the Actor based on the incoming message.

To this day there are wide array of Actor-like implementations. We will discuss a few key Actor models and discuss on which aspects each model differs from other implementations.

2.1.1 Agha Actors

Agha (1986) defined a variation on the Actor model which forms the foundation for most of the Actor models in use today. The main goal of the thesis was to develop a computational model that would allow the programmer to exploit massive parallelism with relative ease on distributed systems. In the paper Agha shows two languages, SAL and Act. Both languages showcase a minimal Actor language. An Actor system according to Agha is comprised of *tasks* and *behaviors*. A task in the context of an Actor system is what drives computations. If a system contains no tasks then the Actors in the system will not perform computations since an Actor only executes computations based on a task. A task contains three elements:

- **Tag** A tag that is used to distinguish this task from all other tasks that might contain an identical target and communication.
- **Target** The target is the mail address of the recipient Actor.
- **Communication** The *communication* is a payload. It contains values exposed by the sending Actor to the receiving Actor.

The sending Actor can only send messages to existing mail addresses. An Actor has three possible ways to know the existence of an address. It can either be included in the received communication, it is the address of a newly created Actor or finally, the address was known before.

The next important element in an Actor system is a behavior. An Actor's behavior is a function. Each time an Actor processes a communication it has to execute this behavior. According to Agha an Actor can be conceptually thought of as a computational agent that maps communications onto a triple containing a finite set of communications (messages) sent to arbitrary Actors, a new behavior which will act on the next communication and a finite set of newly created Actors. This means that each time an actor processes a message from its in-box it can define a new behavior for the next communication, create zero or more new actors and send zero or more messages to other actors.

The `become` construct allows an Actor to specify new behavior for subsequent messages. Many Actors can have the same behavior. Each time an Actor accepts a message and processes it, a new behavior has to be defined that will apply to the next communication. In this perspective, an Actor is defined by its behavior. Defining new behavior in SAL means that a new Actor is created or an existing actor is reused that will process the next message by calling `become` and passing in a mail address as an argument. From that point on, each time the Actor receives a message it will be forwarded to the Actor which was the subject of the call to `become`. In case the Actor does not compute a new behavior after processing a communication the exact same behavior is reused.

```
1  def Factorial() [val, cust]
2    become Factorial() ||
3    if val = 0
4      then send [1] to cust
5          else let cont = new FactorialCont(val, cust)
6            in send [val - 1, cont] to self
7  end def
8
9  def FactorialCont(val, cust) [arg]
10   send [val * arg] to cust
11 end def
12
13 // Top level expression
14 let x = [recep] new Factorial()
15     in send [5] to x
```

Listing 2.1: Implementation of factorial using the SAL actor language.

Listing 2.1 depicts an example of a factorial actor using the SAL language defined by Agha (1986). The behavior of an actor is defined using the `def` construct. A new actor with this behavior is then created by calling `[recep] new Factorial()` (line 14). The `recep` binding indicates that the actor is able to receive messages from the entire system. If a message is sent to the actor the body will be executed (line 2 through 6). In this particular case the actor will execute two actions concurrently using the `||` construct. Since the factorial actor is stateless it can process factorial messages concurrently and thus immediately becomes `Factorial()` again to increase performance.

If the value is 0 the actor sends a message `[1]` to the `cust` continuation (line 4). If it is not, the actor creates a new continuation with the current value and the previous continuation and sends the decreased value and the continuation to itself (line 5-6).

2.1.2 Erlang Actors

A model that is based on the Agha(Agha, 1986) Actors is the Erlang Actor model introduced by Armstrong (2007). The language itself was invented at Ericsson in 1986. Erlang is based on the model introduced by Agha (1986) but has a few variations. For example, in the Agha model one had to specify a new behavior after processing each message (recall `become`). In Erlang an Actor is spawned by using the `spawn` keyword and passing it a function with optional parameters. The given function will then be executed in a separate process, i.e., an Actor. In the body of this function the process can execute arbitrary statements, spawn new Actors, send messages to other Actors or call new functions. In Agha actors, each behavior counts as a new turn. This implies that a new behavior marks the end of the message processing. In Erlang an Actor can do an arbitrary number of executions before executing a receive block somewhere in the body which marks the end of a turn. This means that an isolated step in Erlang is not bound by the end of a function body but by the beginning of a receive block which can be placed anywhere. Thus, in contrast with traditional Agha(Agha, 1986) actors, Erlang actors are processes that run from beginning to end. Anywhere in the execution flow of that process the actor can execute a receive block and process communication.

Example Listing 2.2 shows an example of an Erlang program. The program creates an actor that computes the sum of two numbers. Upon reception of two numbers in the form of a message the actor will compute the sum of these numbers and print them to the screen.

The entry point of the application is the `start()` function (line 13-18). The body creates a new Actor whose address is stored in the variable `Sum_act` (line 14). Next, two numbers are sent to the in-box of the Actor (line 15-16), followed by a "go" message, which tells the Actor to compute the sum (line 17).

The `sum` function waits until the executing process has a "go" message in its in-box. Until then the Actor is idling. Upon reception of the "go" message the Actor receives a first number using the `receive_num` function. This is an example of a nested receive. Even though the `sum` Actor is in the process of handling a message, it can continue to process other messages; in this case numbers. The `receive_number` function returns the received number to the caller. Next, `sum` receives a second number to finally print the result on the screen, which is the end of the processing of the go message. This example clearly shows that a single atomic turn of an actor is the computation that is done between two receive statements. In between those statements no other actor can intervene and manipulate the execution flow of the `Sum_act` actor.

```
1  receive_num() ->
2      receive {Sender, Number} ->
3          Number
4      end.
5
6  sum() ->
7      receive go ->
8          First = receive_num(),
9          Second = receive_num(),
10         io:format("Sum of numbers is ~p~n", [First + Second])
11     end.
12
13 start() ->
14     Sum_act = spawn(?MODULE, sum, []),
15     Sum_act ! {self, 1},
16     Sum_act ! {self, 2},
17     Sum_act ! go,
18     nil.
```

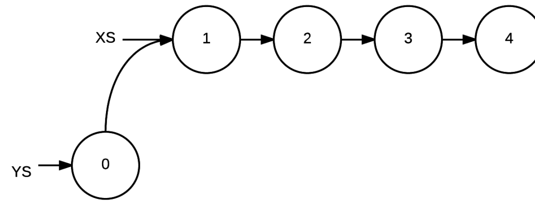
Listing 2.2: Adding two numbers in Erlang

2.1.3 Semantic Properties

Agha (1986) defined two atomic properties of a pure actor language: *atomicity* and *encapsulation*. We will discuss both of them together with three other semantic properties of actor languages(Karmani, Shali, & Agha, 2009): *location transparency*, *mobility* and *fairness*. For each of these properties we will discuss their importance and implications they have on the language. It is to be noted that not all these properties hold for all Actor languages. The discussed semantic properties are by no means an exhaustive list.

Encapsulation Encapsulation is a concept that is well known from object-oriented programming. It allows restricting access to an objects inner state (fields) and functionality (implementation of methods). An encapsulated object its state can only be modified or manipulated by

Figure 2.1: Persistent data structure with structural sharing



means of the interface it publishes to the outside, i.e., public methods and fields. Encapsulation ensures integrity of the object's internal state and disallows sharing state between objects that is intended to be private. In context of Actor languages there are two important parts to ensure encapsulation: *state encapsulation* and *safe message passing*.

- **State Encapsulation** ensures that an actor does not share state with any other actor. This is one of the key aspects that facilitates modularity and mobility in Actor languages. The only interface an Actor has to other Actors is the means of message passing. An Actor can never manipulate the state of another Actor directly. If this would be the case the Actor can no longer guarantee atomicity and consistency. There are languages that do not adhere to this property (e.g., Kilim). In Kilim (an Actor language for the JVM) Actors can have memory references to other Actors' in-box (Karmani et al., 2009). This would allow them to circumvent the sending mechanism and immediately modify the in-box of an actor.

- **Safe Message Passing** guarantees that sending a message with a payload (attached values) will send a copy of the value (*send-by-value*). One might reason that in a non-distributed Actor language it would be safe to send a memory reference to a value instead of making a deep copy (*send-by-reference*), however, this effectively allows another Actor to break encapsulation and consequently the safe message passing semantics.

Atomicity Given the encapsulation property of Actors, a second important semantic property follows: *atomicity*. Since the only interface to communicate with an Actor is message passing and an Actor only processes one message at a time it is possible to reason about a message processing as an atomic single step, also called the *isolated turn principle*. The isolated turn principle states that while an actor is processing a single message it can execute many statements which can be interleaved at will (De Koster, 2014). However, the effects of these statements, i.e., the altered state of the actor, are not visible to the rest of the system until the next message is being processed. This means that only a single message can influence the computation of an actor and that during its computation other messages have no effect on the outcome. In other words, an actors behavior is deterministic inside each atomic turn, provided that the programmer does not introduce nondeterminism himself, e.g., by using I/O.

This property goes hand in hand with encapsulation. If one were to break encapsulation and allow any Actor to manipulate the internal state of an actor the atomicity can no longer be guaranteed. Even more so, if an Actor is allowed to process multiple messages concurrently, the two messages can influence the internal state of the actor, effectively influencing the processing of other concurrently processed messages. This would no longer make it possible to reason about

the computations that take place given only the message and the initial state of the actor because the result of the computation could depend on multiple messages, meaning atomicity no longer holds. However, there have been variations on this semantic by allowing multiple actors to read the same shared state of an actor by publishing it in a read-only fashion(De Koster, Marr, D'Hondt, & Van Cutsem, 2013). De Koster et al. state that even though there are concurrent reads, atomicity is still preserved.

Atomicity is an important property of a system, that aids reasoning about that system. A system that is highly modular allows one to reason about the properties of each module in isolation. Each module of such a system could thus be reasoned about in terms of its in- and output without knowing the actual internal workings (*black box principle*). However, as stated before, due to the inherent nondeterminism of concurrent computations the behavior of any composition of such modules can not be guaranteed due to the interleaving of concurrent computations of separate modules. Atomicity mitigates this problem by guaranteeing that the execution of one operation (message processing) can not be interleaved in such a way that it would break the semantics of each individual component.

Location Transparency In general location transparency is the concept of abstracting actual addresses of resources to a more generic identifier. In the context of distributed Actor systems this means that an Actor is possibly running on an other node over the network and can still be addressed in the same way as an actor running within the same virtual machine or runtime. In any Actor language an Actor has an address to send messages to. By applying location transparency the programmer does not have to differentiate between remote or local Actors, i.e., the programmer is oblivious to the actual location of an Actor. This allows the runtime system (e.g., Erlang runtime) to move Actors from their physical location at runtime which could improve scalability and load balancing. Furthermore, it allows the runtime to replace an actor at runtime as well. This is known as *mobility*(Fuggetta, Picco, & Vigna, 1998).

Mobility Mobility is the ability of a current control state of a computation to be moved from either a processor or computer. Fuggetta et al. (1998) differentiates between *weak* mobility and *strong* mobility. The former is a classification for when it is only possible to move code with a possible set of initialization data, e.g., current state of an actor. For example, moving an idle actor with an empty in-box would imply that only the code of the actor needs to be moved along with its state. Hence, an actor that is not in the middle of an execution. The latter is a bit more complex. Strong mobility means that it is possible to move code along with its current execution state. In most languages this implies the current state of execution (continuation) as well as the current execution stack. (Lange, Oshima, Karjoth, & Kosaka, 1997) have created "Aglets". Aglets are mobile Java agents that facilitate strong mobility of Java programs. When an aglet is moved from one computer to another its current state, its code, its data and execution state are passed. Hence, aglets offer strong mobility.

In most Actor languages code mobility should come quite naturally. For example, Erlang most certainly has weak code mobility. Moving an idle actor with an empty in-box is as simple as spawning the function on another node and passing the values for the formal parameters of that function. Considering the design of the actor model it should be intuitively clear that mobility is quite natural to the Actor model. Most Actor languages enforce encapsulation which makes it easy to move the entire actor to another location.

Fairness Fairness is the notion that an Actor can not be *starved* (i.e., delaying a message delivery indefinitely). An actor that has a message sent to him but can not process it because

the system is too busy spending resources computing is called a *starving* actor. The starving actor does not get scheduled by the runtime and consequently can not execute any operation.

Listing 2.3 depicts a scenario where fair scheduling would be required to avoid starvation of an actor. At the entry point of the application (line 11) an actor is spawned that will execute the `endless_loop` function. This will keep sending messages to the executing actor, in effect creating an infinite loop. Next, we spawn an actor that will print out every message it receives to the standard output. Finally we send a message to the printer. In case of unfair scheduling the printer would not be able to process its message because the `endless_loop` actor might not *yield* control back to the runtime. However, in Erlang there is indeed fair scheduling and thus the printer will print the message eventually.

```
1  printer() ->
2      receive {print, Message} ->
3              io:format("~p~n", [Message])
4      end.
5
6  endless_loop() ->
7      receive loop ->
8              self ! loop
9      end.
10
11 start() ->
12     Endless_Loop = spawn(?MODULE, endless_loop, []),
13     Printer = spawn(?MODULE, printer, []),
14     Printer ! {print, "Here's a message!"},
15     nil.
```

Listing 2.3: Erlang program that might have a starving actor (printer) in case the runtime would lack fair scheduling.

2.2 Transactional Memory

Transactional Memory is a concurrency paradigm that was introduced to help create highly concurrent programs. The problem with concurrent programs is that the critical sections need to be managed with care. If not done properly there can be deadlocks, race conditions or bad interleavings(Lee, 2006).

Taking too much liberty with delimiting critical sections can decrease concurrency severely because critical sections are executed sequentially per definition. Hence it is important to make the critical sections as small as possible. However, if the critical sections are too small they might introduce race conditions.

A technique to avoid deadlocks is to acquire the locks in the same order for each critical section (Lea, 1996). For example, in Java one could acquire locks in increasing order based on their hash.

Transactional Memory aims to avoid writing explicit locking mechanisms by optimistically writing to shared state. Transactional memory uses the notion of transactions to delimit sections of a program that read from and write to shared memory. Instead of pessimistically locking all resources before modifying global memory transactions work in a separate copy of the global memory and then try to *commit* these changes to global memory.

Transaction A transaction is a finite list of read and write operations on shared memory references. The entire dataset of a transaction is the set of written references and read references. A transaction ends when all the operations have been applied. It may then either abort or *commit*. Committing a transaction applies its changes in a single atomic step to global memory and makes them visible to other threads atomically. If a transaction aborts due to a conflict it will be executed again on a fresh snapshot of the current global memory and apply all the operations again. There are different approaches to transactional memory, mainly hardware and software implementations. We will now discuss Hardware Transactional briefly and then continue with Software Transactional Memory.

2.2.1 Hardware Transactional Memory

The main idea behind Hardware Transactional Memory (HTM)(Herlihy & Moss, 1993) is to change the cache coherence mechanism in a multi-core processor. A *cache line* is a part of global memory that a CPU core reads into its own small local memory (cache), which is not shared with other cores. This cache line can be in 4 different states (MESI-protocol). Other protocols (e.g., MOESIE-protocol) allow for other states.

- **Modified** The cache line has been read by a single core, which has written new values to that cache line, therefore invalidating the global memory. The updates are not present in global memory yet and no other cores have read that same cache line.
- **Exclusive** A core has read the cache line but has not written any new values to the cache line. No other core has read the cache line either.
- **Shared** Multiple cores have read the same cache line but no core has written values to the cache line. Therefore the global memory is valid still. Any core can write to the cache line. If any core writes a new value to the cache line it will become *invalid* for all cores.
- **Invalid** A cache line is invalid if it has been read by one or more cores. An arbitrary core has written new values to the cache line. This invalidates the cache line for all cores that have read it and invalidates global memory.

A cache coherence mechanism ensures that a cache line that is shared between processors is always in a consistent state with respect to the cores that read it. If a core invalidates a cache line, the cache coherence mechanism will invalidate it for the other cores as well. Consequently, a core can only write a cache line back to global memory if it is in an *exclusive* or *modified* state.

HTM exploits the cache coherence mechanism by using the core's local cache as a buffer for memory writes in transactions. This allows a single core to execute a transaction (i.e., a series of memory reads and writes) by copying the required data to cache and applying the operations on the cache. If at the end of the transaction the cache lines that were subject to the operations are in the *modified* (writes) or *exclusive* (reads) state, the core knows that no other core has made modifications to cache lines that were read or written in a transaction. Consequently the changes can be committed to global memory and the transaction has succeeded.

Note that this requires the cache for a single core to be big enough to fit all the cache lines that hold the data of locations the transaction wants to write to. This limits the size of the dataset a single transaction can operate on.

If a transaction fails, multiple approaches are possible. A transaction has failed when it wants to commit changes to global memory but an other core has made changes to the same cache line or has read a cache line the transaction has already modified, i.e., one or more cache lines that are in a *invalid* state were target of the transaction. As a result the transaction will flush the cache

and start the operation again under the assumption that the core that invalidated the cache lines is now finished with the write operations and no other core might invalidate these cache lines again. Note that there are plenty of approaches on how to handle a failing transaction. However, they will not be discussed for HTM.

2.2.2 Software Transactional Memory

Introduction

Software Transactional Memory is a software implementation of Hardware Transactional Memory(Herlihy & Moss, 1993). It was introduced by Shavit and Touitou (1995). The most important downsides of HTM are that it requires specific hardware and limits transaction size. Semantically speaking HTM is *blocking* as well. As we will see STM does not suffer from this problem.

A TM implementation is *non-blocking* if, when a transaction is repeatedly retrying due to collisions, there is at least one transaction in the system that will commit after a finite number of steps. This does not have to be that specific transaction. A transaction τ can fail multiple times because it conflicts with another transaction τ'. If after a finite number of steps τ' successfully commits, eventually τ will commit as well. This is an important property to consider because it implies that no two transactions will perpetually retry because they conflict with each other. It could be the case that τ' had to retry n times as well because it conflicted with an arbitrary transaction τ''. If a transactional system is not *non-blocking* the programmer could write programs that result in a *live-lock*. An STM implementation is *wait-free* if a transaction that has retried an arbitrary number of times succeeds.

Transactional Variable In STM implementations transactional variables are typically a special kind of variable. For example, in Clojure a transactional variable is any data structure wrapped inside a `ref` and in Haskell a transactional variable is wrapped inside a `TVar`. As such it is impossible to write to such a transactional variable without starting a transaction. Because data structures are wrapped inside a transactional variable the runtime can attach metadata to each transactional variable. For example, the runtime can store the transaction that has current ownership of this variable.

Transaction The implementation by Shavit and Touitou (1995) covers *static* transactions. A static transaction knows the dataset it operates on beforehand. A static transaction can be thought of as a function that executes in a single atomic step. It takes a parameter (e.g., an array) and applies a modification on this array inside a transaction.

A transaction can be represented as a data structure that holds specific information about that transaction instance: its *size*, *old values* and *version*. Recall that each transaction works on a dataset. *Size* will hold the length of the dataset. *Old values* will contain a copy of the values in global memory. Finally, *version* will hold the iteration of the transaction. Each time a transaction ends by either failing or succeeding this field will be incremented.

At the start of a transaction it will try to acquire the ownership of each reference in *old values*. This is done by putting a reference to the transaction object in a global ownership vector using a *Store_Conditional* operation. If the ownership contains no owner, the transaction can put its reference in there. This vector holds an entry for each transactional variable. When the transaction succeeded in acquiring ownership for all entries in the dataset the transaction is marked as *succeeded*. The transaction will copy the current values from global memory to the *old values* vector. On this vector it will apply the operations in the transaction and return the vector

with the updated values. Finally the transaction copies these values back to global memory and releases ownership of all references in order for other transactions to execute.

If the transaction τ fails to obtain ownership of a reference it will fail. After trying to acquire the ownership τ knows which transaction owns the reference (e.g., τ'). If τ' is not in the *succeeded* state it means that it has not yet obtained all the ownerships for its transaction. Consequently, τ will execute the transaction body of τ' as well. It can obtain the dataset from the ownership field (which is a reference to the transaction object of τ'). Running this transaction might increase contention but it might also succeed faster than τ' itself.

Multi Version Concurrency Control

Multi-version Concurrency Control (MVCC) is a concurrency control mechanism that stems from database systems(Bernstein & Goodman, 1981). In database systems there is also a notion of transactions. Two applications can execute queries on the same table, which might introduce conflicting changes. For example, one program updating the pay grade of a list of employees and another program reading out the pay grade to calculate their paycheck. If updating a list of records takes a long time it would infeasible to lock the entire table for this transaction (i.e., forcing sequential reads and writes). Bernstein and Goodman (1981) address this issue with MVCC. We will explain MVCC in the context of STM. For an in-depth explanation of how we implemented MVCC, see Chapter 6.

MVCC stores versions of each transactional variable to reduce conflicts. Each time a write operation is applied on a transactional variable, that new value is stored as the latest version of that transactional variable.

In a traditional implementation a transaction T_1 will fail if it can no longer read a consistent state of the entire system. This means that at a given point in time, one or more transactional variables have become outdated (i.e., they have been written to by a transaction T_2) since T_1 started. As a result the transaction should abort and execute again with a consistent snapshot of the system state.

Two running transactions T_1 and T_2 will run into a conflict if transaction T_1 writes to variable x and transaction T_2 tries to read the value of x.

Using MVCC, any time a transaction writes a new value to a transactional variable the old version will be stored as well. Doing so it is possible for a transaction to write new values to a transactional variable while any other transaction that is running concurrently can still see a consistent snapshot of the current state of the system by using older values. This concept lends itself to true snapshot isolation (Fekete, Liarokapis, O'Neil, O'Neil, & Shasha, 2005), which implies that no transaction can start with an inconsistent snapshot of the global system state or commit and leave the system in an inconsistent state.

Snapshot isolation can be achieved by applying a total order on the start times of transactions. Each time a new transaction starts (or retries) in an MVCC system it will increment a global counter, called the *global write point*. This value will then be stored in each transactional variable by the writing transaction when it is written to. This allows any transaction to read values from any transactional variable before the transaction itself started. If any transactions have written new values to this variable, the transaction simply reads an older version. This makes sure that each transaction reads values that were present before the transaction started. However, transactions can still commit newer values in the mean while.

If a transaction would try to update a transactional variable that has been written to since it started however, the transaction still has to restart.

Such a system requires a lot more memory than a traditional STM system. However, the price paid in memory space returns a possible speed improvement given there can be potentially less

collisions and thus less transactions that have to execute over and over again. Clojure alleviates this problem by only keeping a limited history for each transactional variable. If a transaction can not read a value of a variable that was written before it started that transaction would have to retry in order to have a consistent snapshot.

2.2.3 Semantic Properties

Transactional memory (Software or Hardware implementations) has a few semantic properties that are used to reason about its correctness. We will discuss the most important ones in detail.

Formal Model To explain correctness criteria for transactional memory a few conventions and concepts must be established. We will base ourselves on a simplified version of the formal model introduced by Bernstein and Goodman (1983).

A transaction T_i reading from a transactional variable x is denoted as $read_i[x]$. A write by transaction T_i on x is denoted as $write_i[x]$. If the transaction that performs either operation is not relevant to the context $read[x]$ will be used.

A transaction executed by a process has a *log*. A log L_i holds the list of operations performed on each transactional variable in the order the transaction T_i executed them. In such a log a transaction can execute multiple operations in parallel. Any given log about a transaction defines an *allowed* execution according to the transactional memory system. Intuitively, the transaction logs are a representation of how the model has executed these transactions. Fig 2.2 is an example of a transaction performed by T_i.

$$L_i = write[x] \rightarrow read[x] \rightarrow read[y]$$

Figure 2.2: Example transaction log

Any log L representing an execution of a transaction has to respect the *program order* of the transaction. The program order is defined as the order in which the transaction expressions appear in the source text. Thus, if a transaction contains two expressions e_1 and subsequently e_2, the log must preserve this order.

Furthermore, in any given transaction log, no transaction can read from a variable x if it has never been written to by another transaction or itself.

A log L_t over a set of transactions $T = \{T_1, ..., T_n\}$ is the union of all the log entries for each individual transaction. A log over one or more transactions introduces a *partial order* on the operations. Intuitively this means that two transactions can execute a read or write operation concurrently. In (Bernstein & Goodman, 1983) the author makes a distinction between logs and their sequential notation. A log over a single transaction per definition only enforces a partial order. This means that several operations can happen concurrently. However, for the purpose of the given definitions below we can use the constraint that a log introduces a total order and thus each operation happens strictly before or after another operation. Furthermore, software transactional systems tend to execute a single transaction sequentially while (Bernstein & Goodman, 1983) is in context of database systems and as such concurrent operations within a single transaction are possible. However, a log over more than 1 transaction still introduces a partial order.

Two logs for a given transaction are equivalent if for each read operation in both logs, the operation yields the same result. Furthermore, the value that each read operation reads must

be written by the same transaction. This is the *read-x-from* relation. If a transaction T_i writes a value to x and T_j reads this value, then in any equivalent log T_j reads the value produced by T_i. Finally, each transaction log must leave the system (i.e., the transaction variables that were subject of the transactions) in the same state after execution. That is, each transactional variable must have the same value.

Serializability

Serializability(Papadimitriou, 1979) is a criterion most commonly used to verify transactional systems.

Suppose a transaction execution that is allowed by the transactional memory system. It comprises of a set of k processes that each execute 0 or more transactions. The k processes are the only processes running in the system and no new processes can start. If, given the set P, of all the transactions that the k processes execute we can create a log T with a partial order over these transactions. Recall that given the context of software transactions we enforce that a single transaction log has a total order. In this log no operations can execute concurrently with any other operation. If the log T is serializable the execution is trivially correct because it proves that the execution is equivalent to a sequential execution. Hence, no issues due to concurrency can arise.

A *serializable log* L is any sequential total ordered log over a set of operations that adheres to four constraints:

(1) For each pair of transactions in T, T_i and T_j, every operation of T_i must precede T_j or every operation of T_j must precede every operation of T_i.

(2) Each read operation in the log L must still yield the same result as it did in the concurrent execution that was accepted by the transactional system.

(3) Each transactional variable must have the same value after executing the sequential log as it had when the transactions were executed concurrently.

(4) Any sequence of transactions issued by a single process must be executed in that same order in the serial log.

Thus, given any log over a set of transactions that has an equivalent serializable log, that execution is said to be correct.

Listing 2.3: Two transactions that should respect the invariant that x and y are always equal.

```
1 ;; T1
2 ;; x and y initially 0
3 (dosync
4   (ref-set x (+ @x 1))
5   (ref-set y (+ @y 1)))
```

```
1 ;; T2
2 ;; x and y initially 0
3 (dosync
4   (ref-set x (- @x 1))
5   (ref-set y (- @y 1)))
```

Listing 2.3 represents two transactions that are safely accepted by the concurrency model of Clojure. This means that it considers them a correct execution. To prove that the transaction

log of these two transactions is serializable, it suffices to construct a serial log with a total order. Such a log is given in Figure 2.4.

Assuming that the first transaction starts first, the Clojure STM runtime will force the second transaction to retry if the second transaction is scheduled before the first one finishes. Consequently, the second transaction will read the values written by the first transaction. Hence, the serialized log shown in Figure 2.4 states the the execution is correct because it is equivalent to the concurrent execution.

If the second transaction were to start first the same rules apply, except the first transaction would read the values of the second transaction. The serialized log for this scenario is depicted in Figure 2.5.

$$L = \underbrace{read[x] < write[x] < read[y] < write[y]}_{\text{T1}} < \underbrace{read[x] < write[x] < read[y] < write[y]}_{\text{T2}}$$

Figure 2.4: Example transaction log

$$L = \underbrace{read[x] < write[x] < read[y] < write[y]}_{\text{T2}} < \underbrace{read[x] < write[x] < read[y] < write[y]}_{\text{T1}}$$

Figure 2.5: Example transaction log

Clojure Serializability The STM system present in Clojure is based on multi version concurrency control. This system keeps older values of transactional variables such that other transactions can read these whilst they are being written to by other transactions. This makes the STM system accept transactions that are per this definition not serializable. To prove this a counter example is given.

Given the code in Figure 2.6 one would expect that given a correct serialization either y or b would be 1, assuming that each variable in the listing is initialized to 0. A serial execution would force T_1 as either the first or last transaction. In which case it would either read `x` with a value of 0 or 1 respectively. The serialized log of the intuitively expected execution is shown in Figure 2.7. However, this execution does not match the actual execution. What actually can happen is that both transactions read a value for `a` and `x` that was the newest value when both transactions started, both being 0. After these two transactions execute concurrently the final value of the variables is shown in Table 2.1. The final values are impossible to recreate with a serial log. This is default behavior in Clojure.

Variable	Initial Value	Final Value
`x`	0	1
`y`	0	0
`a`	0	1
`b`	0	0

Table 2.1: Values of variables in Fig 2.6

Listing 2.6: Two transactions that should respect the invariant that x and y are always equal.

```
1 ;; T1
2 ;; x, y and a initially 0
3 (dosync
4   (alter x inc)
5   (ref-set y @a))
```

```
1 ;; T2
2 ;; a, b and x initially 0
3 (dosync
4   (alter a inc)
5   (ref-set b @x))
```

$$L = \underbrace{read[x] < write[x] < read[a] < write[y]}_{\text{T1}} < \underbrace{read[a] < write[a] < read[y] < write[b]}_{\text{T2}}$$

Figure 2.7: Example transaction log for Figure 2.6

1-Copy Serializability

1-copy serializability is a form of serializability that is useful when testing the correctness of multi-version concurrency control systems (Bernstein & Goodman, 1983). It is a stricter form of serializability in the sense that it takes versions of variables into the equation. Simply put, this correctness criterion holds for all MVCC transactional models that can be serialized as if there was only one version per variable. Using the formal model previously defined 1-copy serializability can be defined as follows.

Each write by process T_i, previously written as $write[x]$ now produces a specific version, x_i. Hence, we write $write[x_i]$. When any process T reads a variable, the version will be denoted; we write $read[x_i]$. Thus, a read operation can have a different result depending on the position in a sequential log. Finally, the *reads-x-from* relation changes. The relation between two processes T_i and T_j only holds when T_j reads the value of x and the last write done before that read operation was a write by T_i, and thus T_j reads the value that was produced by T_i. Operation conflicts now only arise when two transactions read or write the same *version* of a variable.

A 1-copy serializable log is a stricter form of a serializable log. It has to satisfy an additional property with respect to serializable logs (5). A serializable log is 1-copy serializable if and only if the following properties hold.

(1) For each pair of transactions in T, e.g., T_i and T_j, every operation of T_i must precede T_j or every operation of T_j must precede every operation of T_i.

(2) Each read operation in the log L must still yield the same result as it did in the concurrent execution that was accepted by the transactional system.

(3) Each transactional variable must have the same value after executing the sequential log as it had when the transactions were executed concurrently.

(4) Any sequence of transactions issues by a single process must be executed in that same order in the serial log.

(5) If a process T_j reads a version x_i from x then the last write operation to x that happens before the read by T_j in the serializable log is a write operation by T_i.

The previous Clojure example (Figure 2.6) is not serializable and thus not 1-copy serializable. We can apply the `ensure` operations to make the example serializable and 1-copy serializable. The modified source code is shown in Figure 2.8 and the new result set is shown in Table 2.2.

If a transaction τ calls `ensure` on a variable x, x can not be written to in global memory as long as τ is running. Intuitively `ensure` has the same semantics as a so-called *dummy write*. τ could read the value of x and write that value back. This would have no effect other than adding a new version with the same value when τ commits. Thus, while τ is running, τ is guaranteed that when it commits, it has used the latest version of that variable before τ started and no other versions have been created since τ started. Since this is a common pattern, *ensure* mimics these semantics.

To prove that this example is serializable a serialized log is shown in Figure 2.9. One can verify that this log adheres to the properties of 1-Copy Serializability as well.

Variable	Initial Value	Final Value
x	0	1
y	0	0
a	0	1
b	0	1

Table 2.2: Values of variables in Fig 2.6

Listing 2.8: Two transactions that should respect the invariant that x and y are always equal.

```
1 ;; T1
2 (dosync
3   (alter x inc)
4   (ref-set y (ensure a)))
```

```
1 ;; T2
2 (dosync
3   (alter a inc)
4   (ref-set b (ensure x)))
```

$$L = \underbrace{read[x_0] < write[x_1] < read[a_0] < write[y_1]}_{\text{T1}} < \underbrace{read[a_0] < write[a_1] < read[y_1] < write[b_1]}_{\text{T2}}$$

Figure 2.9: Example transaction log

3
Combining Actors and STM

The actor model has a number of properties that can be beneficial for software developers. The strict isolation of different actors increases security and fault-tolerance and avoids safety issues such as low-level race conditions. The downside is that the actor model compromises on expressiveness, especially when modeling access to a shared resource. Other researchers have identified that developers mix the actor model with other concurrency models (Tasharofi et al., 2013) to overcome these issues. This chapter presents the inadequacies of the actor model as a stand alone concurrency model, especially concerning the modeling of access to a shared resource. Additionally, this chapter shows that an ad hoc integration of another concurrency model such as software transactional memory compromises on the safety properties of the original actor model and that a tighter integration of both models is necessary.

3.1 Inadequacies of the Actor Model

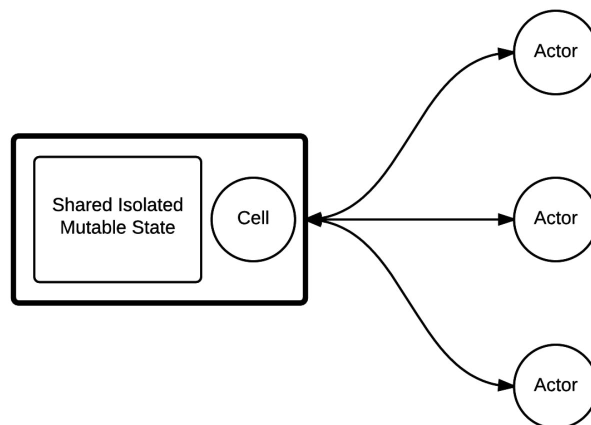
3.1.1 Shared State in Actors

The actor model traditionally does not support shared memory. This is ensured by the *isolation principle* (Karmani et al., 2009) (chapter 2). No actor can have synchronous access to the state of any other actor. The only means of accessing state outside of the actor is by means of the asynchronous messaging paradigm. However, a shared mutable resource can be modeled by the actor model nonetheless, this is called a *delegate actor*. A delegate actor is a common idiom in actor programming because it allows one to encapsulate state into an actor and sharing this resource by passing the address of the actor. The delegate actor will then expose the value to the rest of the system using a specific vocabulary (e.g., `update-value` and `read-value`). Any other actor that has the address can then read and write to this actor. This effectively yields shared mutable and isolated state. However, as we will see it has of downsides as well.

Figure 3.1 is a schematic representation of a delegate actor. The circles represent actors. The square represents the isolated state of a single actor that is only accessible from within that actor. As we can see the state of the *cell* actor (i.e., a delegate actor) is only accessible through the *cell* actor by means of message passing. The actor can then reply with a message that carries

with it the current internal state. Messages are represented by arrows.

Figure 3.1: Model of a delegate actor representing shared mutable isolated state



In Listing 3.1 a small program is shown that represents a cell actor. The `cell` function represents the behavior of a delegate actor. The actor understands two messages: `"set"` and `"read"`. The former allows any actor to set a new value in the actor and the latter allows any actor to request the current value of the cell actor. By passing the address of this cell around actors can share common state.

Using an actor in this fashion introduces race conditions on the message level that are well known in concurrent programs that allow shared memory references. Just as with threads, this can cause some unexpected interleavings. The reason for this is that the actor model does not enforce any order on messages. As a result, messages coming from multiple actors can be interleaved in an arbitrary order.

```
1  cell(Value) ->
2      receive
3          % Set a new value
4          {set, NewValue} ->
5              cell(NewValue);
6          % Read the current value
7          {read, Sender} ->
8              Sender ! {value, Value},
9              cell(Value)
10     end.
11
12 increment(Cell) ->
13     Cell ! {read, self()},
14     receive {value, OldValue} ->
15             Cell ! {set, OldValue + 1}
16     end.
```

Listing 3.1: Erlang implementation of a delegate actor.

As an example of a bad interleaving suppose two actors A and B that want to increment the value of a `Cell` actor. First A and B would need to send the `read` message in order to receive the current value. Upon reception of the current value from *Cell*, A and B could then compute the new value and send it to *Cell* to update. Supposing the initial value of *Cell* is 0. If actor A sends `read` to *Cell* to request the current value and immediately receives the value. Before A can send back his updated value, B might have already sent `read` and received the reply with the current value (0). If both actors then send their new updated value, the final result will be 1, while one would expect it to be 2.

This problem could easily be fixed in many ways. The first one could be to semantically lock the *Cell*. This could be done by introducing a protocol to access the actor. This changes the behavior in such a way that once any actor A has sent a message, the *Cell* will only process messages coming from A. When A is done using *Cell* he sends a message to release the ownership of the actor. Only from that point other actors could change or read values back from the cell. An example implementation in Erlang is given in Listing 3.2.

To acquire ownership of the *Cell* actor, an actor sends the `start` message to the actor together with his own address (line 18). The *Cell* actor will then change his behavior to the first definition of `safeCell` (line 1). At that point the actor only processes messages coming from the sender that acquired the ownership. Extra code has been added to ensure that only the current owner of the actor can update and read the state (lines 1, 4, 8, 11-12 and 18-19). The `read`, `set` and `release` messages now expect an argument `Sender`. This will be the address of the sender. When this address does not match the address of the currently owning actor, the message is not processed.

One can immediately notice that the code is much more verbose and contains a lot of code to properly handle the protocol to have exclusive access to the actor. There is extra state required in the *Cell* actor as well. It now has to keep track of who the current owner is, explicitly. This works reasonably well for a small *Cell* actor but it introduces the same issues as with manual locking, which was one of the reasons to use actors in the first place.

Another thing to note is that the user of the code (i.e., `safeIncrement`) has to know that the actor has an extended vocabulary to support the ownership protocol. The user itself then has to adapt to this ownership protocol and implement extra vocabulary to operate properly with the *Cell* actor. This damages modularity which is in contradiction with the Actor philosophy.

As for composability, using this form of ownership protocol, composability is most certainly damaged. There is no way to lock the *Cell* from actor A, and then modify it from another arbitrary actor. Since in traditional actor systems *everything is an actor*, chances are that communication with another actor than `Owner` is required which is using the above transaction system impossible.

Considering the above solution one will also notice that in a scenario where only read operations are applied to the *Cell* actor, they are all performed in sequential order. This is a very strict bottleneck for operations that in theory could be executed in parallel. However, there is no way to emulate this behavior in a pure actor framework.

In conclusion, the proposed protocol solution does solve the problem at hand, however it introduces accidental complexity that makes it very hard to reason over a system implemented this way.

```
1  safeCell(Value, Owner) ->
2      receive
3          % Read the current value
4          {read, Owner} ->
5              Owner ! {value, Value},
6              safeCell(Value, Owner);
7          % Set a new value
8          {set, NewValue, Owner} ->
9              safeCell(NewValue, Owner);
10         % Release the lock
11         {release, Owner} ->
12             safeCell(Value)
13     end.
14
15 safeCell(Value) ->
16     receive
17         % Lock the actor for Sender
18         {start, Sender} ->
19             safeCell(Value, Sender)
20     end.
21
22 safeIncrement(Cell) ->
23     Cell ! {start, self()},
24     Cell ! {read, self()},
25     receive {value, Old} ->
26             Cell ! {set, Old + 1, self()},
27             Cell ! {release, self()}
28     end.
```

Listing 3.2: Erlang implementation of a lockable delegate actor.

3.1.2 Motivating Example

In Listing 3.3 a variation on the consumer-producer problem is depicted. This scenario is a classic example of thread synchronization in multi-threaded programs. The problem is that one or more threads produce values and one or more threads consume values. The values are stored in a shared buffer. The consumer should only consume a value if the shared buffer meets a certain condition, e.g., the buffer is not empty. The producer can only produce values if a certain condition is met as well, e.g., the buffer must be empty or it can not exceed a certain value. To achieve this in Java for example, one would use locks to create a critical section in which each thread can read and optionally update the shared buffer in a mutual exclusive manner. However, the actor language, as we saw does not support shared memory. As such it is required to create a protocol to synchronize access to a delegate actor. Listing 3.3 depicts such a protocol.

Syntax

The following example and scenarios are written in Clojure syntax, including the actor syntax of the library that is built for this thesis. This allows for more visual explanations. We will require a few key constructs: transactions, transactional variable read and write and message sending. We will use `"...+"` to denote one or more arbitrary Clojure expressions.

- To create an actor one has to call the `create-actor` function. That function takes a

keyword (a word that starts with a colon) to specify the actor name and a function that defines the initial behavior of that actor. Each formal parameter of the function that is given has to be initialized with a value as the last arguments of the call to `create-actor`.

```
1 (create-actor
2   :counter-actor
3   behavior-fn)
```

- A transaction is executed by wrapping an arbitrary number of statements in a `dosync` block. The transaction ends with the last statement in the block. To write to a transactional variable one can use the `ref-set!` construct. It takes a transactional variable and a new value. To read the value of a transactional variable – which can be done outside of a transaction as well – one can use the `@` construct.

```
1 (dosync
2     (ref-set! a (+ x 1)) ;; Set the value of a
3     (println @a) ;; Print the value of a to stdout
4     ...+)
```

- To process messages from the in-box the `receive` block is used. Intuitively speaking the `receive` block works the same as a `case` statement in a lot of languages. Each case defines a message and possible arguments that are attached to that message. The `receive` will block until one of the cases has been matched with a message.

```
1   (receive
2     ([ :message1 ]
3      ...+)
4     ([ :msg2 arg1 arg2]
5      ...+))
```

- To send a message to an actor the `!` construct is used. It takes a keyword that identifies an actor and a message with possible arguments. If there are arguments they are put as last argument to `!` in a map. The keys are the names of the parameters of the receiving actor.

```
1 ;; Send message "message1" to actor.
2 (! actor :message1 )
3 ;; Send message with two arguments, x and y.
4 (! actor :message1 {:x x :y y}])
```

Apples and Oranges

The program in Listing 3.3 shows the consumer-producer program using solely actors. In this program a single producer will produce apples and oranges. The condition on the buffer in order to produce new apples and oranges is that the buffer must be empty. The consumers in this example can only consume values if the the buffer is not empty.

Lines 1-13 define the delegate actor that will be used as the shared buffer in this example. Lines 15 - 29 define the consumer actor and lines 33 to 36 are top-level statements to initiate the program.

The `data-actor` has a complex vocabulary because a protocol is used to make sure a single thread can "lock" the actor until it is done. Note that this protocol is specific for this program and is thus not generically applicable.

If an actor wants to read and possibly update the values in the delegate actor it has to first send a `:lock` (line 4) message to the delegate actor and then await an acknowledgement of the delegate actor which will reply with an `:ok` message (line 5). After the acknowledgement the actor that uses the delegate actor has exclusive control over it. It is allowed to read the state by sending the `:read` message (line 7). After that the delegate protocol will only accept `:unlock` or `:update` (lines 10-13). Upon reception of either messages the delegate will release the lock and accept a new `:lock` message (lines 11 and 13).

The `consumer` actor will wait for `:consume` messages (line 17). If the actor receives such a message it will first lock the delegate actor (line 19). After that the actor blocks until it receives an acknowledge message (line 20). To read the state it sends the `:read` message (line 22) afterwards and awaits for the delegate actor to reply the current state of the buffer (line 23). If the buffer is not empty the consumer will reply with an `:unlock` message (line 29), if not, it will reply with a data update in which there is one less apple and one less orange (line 27-28).

Lines 49 to 52 are the top level statements that start off this example. Line 49 creates 100 consumers. Line 50 creates a single producer and line 52 sends the initial `:produce` message to the producer actor.

```
(defn data-actor
  [data]
  (receive
   ([:lock sender]
    (! sender :ok) ; Acknowledge lock to sender
    (receive
     ([:read sender] ;; Wait for a read
      (! sender :data {:data data}) ;; Send the data
      (receive
       ([:unlock] ;; Wait for an unlock or an update
        (recur data))
       ([:update [newdata]]
        (recur newdata))))))))

(defn consumer
  [data-actor]
  (receive
   (([:consume]
     (! data-actor :lock {:sender *self*})
     (receive
      ([:ok]
       (! data-actor :read {:sender *self*})
       (receive
        ([:data {:apples appls :oranges orngs}]
         (if (and (> appls 0)
                  (> orngs 0))
           (! data-actor :update {:newdata {:apples  (dec appls)
                                            :oranges (dec orngs)}}])
           (! data-actor :unlock)))))))
    (recur data-actor))

(defn producer
  [data-actor]
  (receive
    ([:produce]
     (while true
       (! data-actor :lock {:sender *self*})
       (receive
        ([:ok]
         (! data-actor :read {:sender *self*})
         (receive
          ([:data {:apples appls :oranges orngs}]
           (if (and (= appls 0)
                    (= orngs 0))
             (! data-actor :update {:newdata {:apples  (inc appls)
                                              :oranges (inc orngs)}}])
             (! data-actor :unlock)))))))))

 ;; Create 100 consumers and 1 producer
(let [consumers (map (fn [x] (create-actor (keyword x) consumer))(range 100))
      producer  (create-actor :producer producer consumers)]
  ;; Start off the producer loop
  (! producer :produce))
```

Listing 3.3: Producer Consumer problem implemented in the Clojure actor library using solely actors.

The example shows that using shared mutable state in the actor model is not really convenient.

The code is very verbose and most of the program is dedicated to ensuring the correctness of the protocol.

The example relies heavily on convention. If the user of the shared state actor does not obey to this convention the system will fail. One actor who is waiting on the reply for the `:read` message can not be ensured that the message will be sent. It is very much possible that another actor did not obey to the convention and sent a `:read` message without acquiring ownership.

The example should reflect that it can be very desirable to be able to use other concurrency paradigms in certain scenarios.

3.2 Ad-hoc Combination of the Actor Model and STM

Given the previous consumer-producer example it is desirable to combine another concurrency model with the actor model to simplify certain programs. We will combine the actor model with software transactional memory to reduce the size of the example and improve readability and safety. However, as we will shortly see, the ad-hoc combination of these two models introduces several issues that need to be addressed.

3.2.1 Motivating example revisited

The example from subsection 3.1.2 can be improved vastly by using software transactional memory to manage the critical section, i.e., reading and updating the shared state. Listing 3.4 depicts the same program but this time software transactional memory was used.

Apples and Oranges with STM

```
(def apples  (ref 0))
(def oranges (ref 0))

(defn consumer
  []
  (receive ([:consume]
              (dosync
               (let [orangecount (ensure oranges)
                     applecount  (ensure apples)]
                 (when (and (> orangecount 0)
                            (> applecount  0))
                   (ref-set! oranges (dec orangecount))
                   (ref-set! apples (dec applecount))))))))

(defn producer
  [consumers]
  (receive ([:produce]
             ;; Keep producing
             (while true
               (dosync
                (let [orangecount (ensure oranges)
                      applecount  (ensure apples)]
                  (when (and (= orangecount 0)
                             (= applecount  0))
                    (ref-set! oranges (inc orangecount))
                    (ref-set! apples (inc applecount))
                    ;; Notify consumers
                    (map (fn [c] (! c :consume)) consumers)))))
             (recur consumers))

 ;; Create 100 consumers and 1 producer
(let [consumers (map (fn [x] create-actor (keyword x) consumer) (range 100))
      producer  (create-actor :producer producer consumers)]
  ;; Start off the producer loop
  (! producer :produce))
```

Listing 3.4: Producer-Consumer pattern implemented in the Clojure actor library using STM.

Listing 3.4 defines, just as in the previous version of the example, a producer and a consumer. Lines 4 to 13 define the consumer. Each time the consumer receives a `:consume` message (line 6) the consumer will initiate a transaction (lines 7 to 12). The consumer will then read the transactional variables (lines 10 and 11). If the there are more than 0 apples and oranges the consumer will take one of each and update the global variables accordingly. Finally, the consumer will await the next `:consume` message.

The producer (lines 15 to 29) will await a single `:produce` message. Upon reception it will start an infinite loop. In each iteration of this loop the producer will read out the current value of the apples and oranges. If there are no apples and oranges the producer will create one of each (lines 25 and 26). After the update the producer will send a message to each consumer (line 28). Finally, the transaction ends and the producer starts the next iteration.

3.2.2 Discrepancies of the the Ad-Hoc Combination

Given that transactions are executed one or more times the programmer has to avoid expressions with side effects in a transactions. Examples are disk I/O or network calls. The reason being that those expressions are not aware of their context (i.e., executing inside a transaction) and transaction can possibly re-execute. It is the task of the transaction to be aware of the statements executed inside of it and handle them appropriately.

When combining the actor model with STM, message sends that happen inside a transaction have to be treated differently than those outside of a transaction. Furthermore, transactions have to be aware of the context they are executed in themselves.

In context of actors, sending a message can be considered an I/O action. When executed it changes the state of a part of the system; the in-box of the addressee is changed. This action can not be undone strictly speaking. If it could be undone by the sending actor this would violate the state encapsulation principle(Karmani et al., 2009) because the sender would be able to modify the receiver's in-box. Ergo, care has to be taken on the receiver side when processing a message that was sent from within a transaction. Combining actors and STM also requires us to reason about what happens when a transaction fails or what has to be kept track of if a transaction is started inside of a receive block.

We will now discuss a few key scenarios that are the foundation of the problems that arise when combining the actor model with software transactional memory.

Sending Messages

Listing 3.2: Sending a message inside a transaction.

```
1 (dosync
2   (! :counter :increment)
3   (ref-set! var 42))
```

Listing 3.2 implies that we are doing an I/O action inside of a transaction: a message send. A simple example of a send within a transaction is shown in Listing 3.2. The piece of code depicts a transaction that sends a message ":increment" to actor `:counter` (line 2). After that, the snippet depicts a `ref-set!` (line 3). As discussed before, this is a write to a transactional variable and thus is prone to failure. If the write fails due to another transaction having written a newer value to it since the transaction started, the transaction will retry. Consequently the message to `:counter` will be sent again. Without any further action the previous message will still be in the in-box of `:counter`, unless the actor has already processed it. This is a problem because the message is sent an arbitrary amount of times depending on the number of retries of the transaction. This is in conflict with the intended behavior of the program as the increment message should only be sent once.

Receiving Messages

Listing 3.3 will be used to explain the notion of *dependencies*. The example depicts a scenario of a simple counter actor. Upon reception of the ":increment" message, the actor will increment its state by one. Hence, the message receive has a side effect. The statement on line 6 is a recursive call to the actor's behavior function. The parameter for the recursive call is thus the incremented counter.

Listing 3.3: A receive example that changes the state of an actor.

```
1 (create-actor
2   :counter-actor
3   (fn [counter]
4     (receive
5       ([:increment ]
6         (recur (inc counter))))))
7
8 ;; Sending message
9 (dosync
10   (! :counter :increment)
11   (ref-set! var 42))
```

The actor in isolation does not have a problem with transactions, as it does not use them. However, when combined with a system that does use STM additional bookkeeping is required. Considering that a message sent to this actor can come from within a transaction. If that transaction is not yet committed, it can potentially abort and be re-executed. In that case, the message can potentially be resent or not sent at all depending on how that transaction is re-executed. This means that the processing of messages that are sent from within a transaction needs to incorporate the transactional behavior of that transaction.

Listing 3.4: A counter actor that increments a local variable and logs this action.

```
1 (create-actor
2   :counter-actor
3   (fn [counter]
4     (receive
5       ([:increment ]
6         (! :log-actor :log {:line "incremented counter"})
7         (recur (inc counter))))))
8
9 ;; Sending message
10 (dosync
11   (! :counter :increment)
12   (ref-set! var 42))
```

A second scenario to consider is what happens when a message is being sent from within a receive. Listing 3.4 depicts the definition of an actor `:counter-actor`. The actor processes `:increment` messages and upon each message it logs the increment operation by sending a message to the `:log-actor` actor. After that the behavior is executed again by calling `recur`, which calls the function again with the incremented counter.

Looking at a modified example in Listing 3.4 we can see that this time a new send is embedded. The counter now sends a message to a logging actor that asks it to log the increment operation (line 6). As stated before, in case of failure a transaction should undo all its operations. However, the message send can not be undone and therefore the line will be logged as many times as the transaction is executed while the counter is only incremented once.

Another problematic scenario is a transaction that is started as the result of receiving a message. Transactions are used to synchronize access to a set of variables such that the updates happen atomically. Hence, it is not unthinkable that a receive body uses a transaction to achieve this effect. In Listing 3.5 the `:counter-actor` actor is depicted. This time it has no local state but uses a transactional variable to which a reference is stored in the state of the actor. As such,

Listing 3.5: Receiving a message and executing a transaction inside the receive body.

```
1  (create-actor
2    :counter-actor
3    (fn [counter]
4      (receive
5        ([:increment ]
6          (dosync
7            (ref-set! counter (+ @counter 1))
8            (! :log-actor :log {:line "incremented counter"}))))
9        (recur counter))))
10
11 ;; Sending message
12 (dosync
13   (! :counter :increment )
14   (ref-set! var 42))
```

a transaction is required to modify it (lines 6-8). In the receive body of `:increment` the actor starts a transaction. It reads out the previous value and sets it with the incremented value (line 7).

If the `:increment` message did not come from within a transaction there is no immediate problem. The actor can simply execute his receive body and the including transaction as many times as he needs to until it succeeds. However, the problem resides in the possibility that the `:increment` message can be sent from within a transaction.

If the `:increment` message has been sent from within a transaction it is possible that safety can not be guaranteed. Traditionally, in STM, a nested transaction becomes part of the parent (i.e., enclosing) transaction. In order for the parent transaction to succeed, all nested transactions have to succeed. However, when those nested transactions cross actor boundaries this is no longer the case. In Listing 3.5 the inner transaction on line 6 is independently executed from the parent transaction that was initiated before sending the `:increment` message on the sending actor.

Suppose that the transaction that sent the message `:increment`, T_{send}, is still running while the `:counter-actor` actor reaches the end of its transaction T_{count} (line 8). If it commits it changes to the global heap there is no way to revert those changes. Thus, if T_{send} should retry the counter is incremented for each attempt at the transaction.

The code listing in Listing 3.6 depicts one of the more difficult situations. There are two actors present in this example, `:swap-actor` and `:compute-actor`. The latter responds to a message `:compute` (line 16-20). The message has three arguments, three references to transactional variables (line 16). Upon reception the `:compute-actor` will start a transaction. Inside that transaction it sends a message to the `:swap-actor` to swap the values in both transactional variables (line 18). After that the actor waits for an acknowledgement from the `:swap-actor` (line 19). The final expression in the transaction (line 20) stores the difference of the swapped variables in the `result` variable.

The former actor, `:swap-actor`, responds to a `:swap` message with 3 arguments. The expected arguments are two transactional variables and a name of an actor (line 5). Upon reception of the message the actor will start a transaction. Inside the transaction it will swap the values of the two given transactional variables (lines 7-9) and finally, after the transaction reply to the sender that the values have been swapped.

The actor `:compute-actor` starts a transaction to send the swap request to the `:swap-actor`. This means that the message can be sent multiple times. The `:compute-actor` receives this message and starts a transaction as well. Hence, if the transaction in the compute actor fails

Listing 3.6: Transaction in a receive body that was initiated from within a transaction.

```
1 (create-actor
2   :swap-actor
3   (fn []
4     (receive
5     ([:swap ref-a ref-b sender]
6           (dosync
7             (let [temp @ref-a]
8               (ref-set! ref-a @ref-b)
9               (ref-set! ref-b temp)))
10            (! sender :finished ))
11    (recur)))
12
13 (create-actor
14   :compute-actor
15   (fn []
16     (receive
17       ([:compute a b c]
18        (dosync
19          (! :swap-actor {:ref-a a :ref-b b :sender *self*})
20          (receive ([:done]))
21          (ref-set! result (- @a @b)))))))
```

twice, the variables will be in their initial position, i.e., not swapped. The second problem that can arise in this scenario is when the transaction of the sender, `:compute-actor` fails after it has received the acknowledgement of the `:swap-actor`. The actor will write the difference of both values into the `result` transactional variable. If that variable has been written to in the mean time the transaction will abort. As such, one might intuitively expect the swap to never have taken place but that action can not be undone. Consequently, the second attempt of the transaction will request the variables to be swapped again. This would yield a faulty result if the transaction succeeded.

3.3 Conclusion

This chapter has made it clear that the actor model used in isolation is lacking when it comes to expressiveness regarding certain programming tasks. It is certainly helpful to be able to combine it with software transactional memory to increase readability and maintainability. However, combining both models can not be done ad-hoc but requires specific runtime support.

4 Actransors

In the previous chapter we discussed the discrepancies that exist when the actor model is combined with transactional memory without any intervention of the runtime. In this chapter we will explain a solution that defines what actions the runtime should take in each of the previously stated problem scenarios and as such mitigates the discrepancies of the ad-hoc combination of actors and STM. A short introduction will be given to concepts that are the foundation of the solution which is followed by a high level explanation of the solution to theproblem scenarios given in the previous chapter.

The solution proposed in this thesis will allow one to use transactions in combination with the actor model. The solution will absorb message sends into the transactional model such that messages are handled properly in a transactional context and can be safely sent inside transaction.

4.1 Actransor Concepts

In this section we will explain the basic concepts that are used when discussing problem scenarios and solutions.

4.1.1 Context

A context is an implicit status of any given actor at runtime. The context helps us reason about which course of action the runtime should take during a transaction or a receive block. A *volatile* context is a context that may fail and re-execute. The body of a transaction is thus a volatile context. A *stable* context is the opposite of a volatile context: a context that will surely finish executing – disregarding application level exceptions and programmer errors such as infinite loops. And finally, a *depending* context is a context in which the executing actor is depending on another actor or transaction. Thus, an actor can be in a volatile state while being in a depending state as well.

4.1.2 Dependency

In the context of the actor model each computation is initiated by means of a message send. A message send will eventually invoke a computation in an actor (fairness). If an actor starts a transaction and sends a message inside that transaction that message too will invoke a computation. At the point of invocation of that computation the executing actor is `depending` on the sending actor. The sending actor can not guarantee that its transaction will finish. The sender is in a volatile context. Hence, the receiving actor will have to take this into account before ending his atomic turn, i.e., processing the next message.

Dependency Propagation

Dependency propagation is the notion of passing along dependencies during message sending. If an actor A depends on an actor B and sends messages to other actors these actors should also depend on B. Thus, each time an actor sends a message to other actors it passes its dependencies along with that message.

Dependency Gathering

The actor model we are using for our experiments and on which the formal semantics discussed in chapter 5 are based, is inspired by the Erlang actor language. This means that an actor is a running process that can execute receive statements at any given point during the execution of that process. A receive statement is a normal expression so they can be nested arbitrarily. As such, an actor that depends on another actor can still receive another message which also contains a dependency. This allows an actor to gather dependencies and depend on a set of actors at any given point during its execution.

4.2 Discrepancies resolved

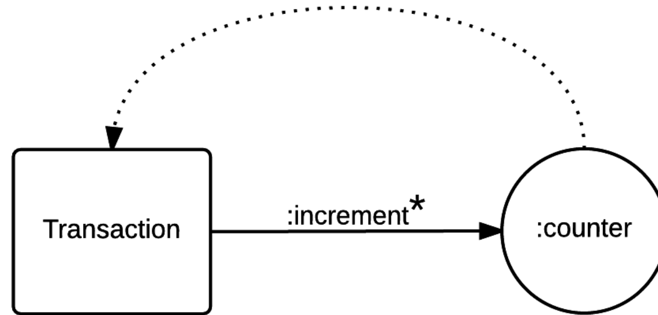
Listing 4.1: Sending a message inside a transaction.

```
1  ;; Sending message
2 (dosync
3   (! :counter :increment)
4   (ref-set! var 42))
5
6 ;; Counter actor
7 (create-actor
8   :counter
9   (fn [counter]
10    (receive
11      ([:increment ]
12       (recur (inc counter))))))
```

Listing 4.1 depicts the first example from the previous chapter (lines 1-4). Lines 6 to 12 show the actor the `:increment` message is being sent to.

Lines 1 to 4 show a transaction. Inside that transaction a message is sent to the `:counter` actor and afterwards the transactional variable `var` is updated to 42. Lines 6 to 12 depict the actor that will handle the message. Upon reception it will simply end the behavior function and re-execute it with the counter incremented by one (line 12). The proposed solution here is to

Figure 4.2: Dependency graph of Listing 4.1



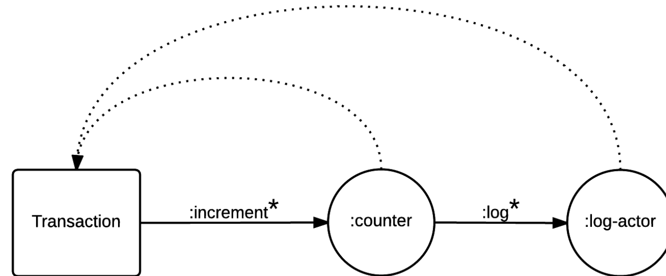
attach metadata to the message. The message will contain a reference to the transaction object which allows the receiver to determine if the sending transaction is still executing, has committed or has retried at any time. In this particular case the sender (line 3) will attach the metadata to the message. Upon reception (line 10-12) the counter actor will notice that this message comes from a volatile context and thus become depending on the sender. When the actor reaches the end of its receive body (line 12) it will wait for the transaction of the sender (line 2-3) to change to a committed or aborted state. If that transaction successfully commits the counter actor can safely process the next message. Hence, in between the end of the receive and the processing of the next message the counter actor is in a stable state. If the transaction were to fail or retry the counter actor should abort its receive body as well. As a consequence the counter inside the actor will not be incremented, which is the desired behavior. The message `:increment` will be removed from the in-box of the message and the counter actor will simply wait for the messages to be sent again (line 3). This yields the desired behavior. The counter of the counter actor will only be incremented once if the transaction safely commits. If it does not the counter will be unaffected.

Figure 4.2 shows the dependency graph for Listing 4.1. The rectangle represents the transaction from the example (lines 2-4). The circle represents an actor, in this case the `:counter` actor. A solid line going from the transaction to the actor indicates a message send. In this case the sent message is the `:increment` message. The message is marked with a star on the right hand side to indicate that it carries a dependency. Upon reception the counter actor becomes dependent on the transaction, which is depicted by a dotted arrow.

Listing 4.3: A counter actor that increments a local variable and logs this action.

```
1  (create-actor
2    :counter-actor
3    (fn [counter]
4      (receive
5        ([:increment ]
6          (! :log-actor :log {:line "incremented counter"})
7          (recur (inc counter))))))
8
9  ;; Sending message
10 (dosync
11   (! :counter :increment)
12   (ref-set! var 42))
```

Figure 4.4: Dependency graph of Listing 4.3



The program in Listing 4.3 depicts a variation on the previous problem. Recall that in the previous chapter we discussed that the issue with this program is that the `:counter-actor` sends a message to another actor while it depends on a transaction. If the sending transaction (line 10-12) re-executes the `:counter` actor should also retry. The previous example in this chapter covered this scenario. This scenario introduces *dependency propagation*. Sending `:log` to the logging actor does not happen in a transaction (line 6). However, the context it is sent in is a *depending* context. The counter actor depends on the transaction that sent the `:increment` message (line 10-12). As such, that dependency is propagated by attaching it to the message. This effectively makes the logging actor depend on the transaction in lines 10 - 12. This reduces the problem to the problem of the previous example. The logging actor will also wait until the transaction that sent the `:increment` message commits. If it were to fail the logging actor will abort as well and process the next message.

Figure 4.4 depicts the dependency graph for Listing 4.3. The only addition with respect to the previous version (Figure 4.2) is the `:log-actor`. The actor receives a message from the `:counter` actor and consequently depends on the initial transaction. Notice that the `:log` message has a star on its right hand side as well. This is due to the fact that the message propagates dependencies.

Listing 4.5: Receiving a message and executing a transaction inside the receive body.

```
1  (create-actor
2    :counter-actor
3    (fn [counterref]
4      (receive
5        ([:increment ]
6          (dosync
7            (ref-set! counterref (+ @counter 1))
8            (! :log-actor :log {:line "incremented counter"}))
9          (recur counterref)))))
10
11 ;; Sending message
12 (dosync
13   (! :counter :increment )
14   (ref-set! var 42))
```

The example in Listing 4.5 shows the counter actor again. This time the counter actor will have a reference to a transactional variable in the global heap to store its current counter value. The actor will keep the reference to this variable as its state (`counterref`). Upon reception of the `:increment` message (line 4) the actor will start a transaction to increment the counter by

one (line 7). Afterwards the actor sends a message to a logging actor to log the operation. Note that the message to the `:counter` actor is being sent from within a transaction (line 11-13).

Recall that the problem here is that the `:counter` actor is receiving the `:increment` message from within a transaction and executes a transaction itself. If the transaction in the `:counter` actor commits before the transaction of the sender commits the changes can not be undone. As a consequence, if the sending transaction (lines 12-14) executes again the counter will be incremented whilst it should not have incremented yet.

The proposed solution in this scenario is introducing an intermediary state for a transaction. The issue in this case is that the receive statement executed by the `:counter-actor` (lines 5-9) depends on the sending transaction in its entirety. This means that not only the transaction of the `:counter-actor` is depending, the context it resides in is also depending on the sender. For the sake of argument we will assume that this example is executed in isolation and the `:counter-actor` does not have any other dependencies. Thus, when the transaction would normally commit, i.e., at its syntactical end (line 8), the transaction will go into an intermediary state. This allows the rest of the receive block to finish (line 9) and reach the end. It is at the end of this receive block that the the dependencies are checked. If the dependencies – in this case the sender of `:increment` – have all committed the transaction in the `:counter-actor` can safely commit as well and the receive block can safely finalize.

In case the sending transaction fails the counter actor will fail to resolve the dependency at the end of the receive block. As a result it will not commit the transaction log to the global heap and jump back to the beginning of the receive statement. The message that initiated the receive block, `:increment` will be removed and the actor will wait for the next message.

The protocol described above ensures that no changes are committed to the global heap and as such the counter will not be incremented unless the sending transaction commits. This effectively ensures that the intended program behavior is preserved.

Listing 4.6: Example of nesting a receive and executing a transaction inside the receive body.

```
1  (create-actor
2    :logging-actor
3    (fn []
4      (receive
5        ([:log line]
6          (receive [:logwriter writer]
7            (write writer line))))
8      (recur)))
9
10 ;; Sending message
11 (dosync
12   (! :logging-actor :log {:line "log this line"})
13   (ref-set! var 42))
```

The program depicted in Listing 4.6 depicts the `:logging-actor` (line 1-7) . This actor's dictionary defines a single message `:log`. The message has a payload `line`. This contains a line that the actor has to write to a log. The outputstream to write the line to will be sent as an additional message. After receiving a `:log` message the actor will block until it receives a `:logwriter` message. This message will carry with it a reference to a writer, e.g., a reference to `stdout`. Upon reception of that message the logging actor will write the line.

Recall that the problem in this scenario is that the logging actor possibly receives the `:log` message multiple times if the sending transaction (line 10-12) re-executes. The `:logwriter` message however is not sent from within a transaction and as such will only be sent once. If the

logging actor retries it will thus not receive that message again.

To solve this issue we propose the following. When an actor receives a message that has no messages associated with it the receiving actor can rest assured that this message is intended to be sent once and will only be sent once. If it were to be sent in a volatile context – and potentially multiple times –it would have had a dependency attached. As such, the actor marks the received message accordingly and does not remove it from its in-box upon failure.

If the actor's receive body were to fail it will iterate all the processed messages during that receive block and nested receive blocks. If any of the messages are marked as having no dependency the message is remarked as a newly arrived message. When the actor then retries his receive body he can simply receive the same message again without the sender having to resend it.

Applying this protocol to the scenario depicted in Listing 4.6 will ensure that the program behaves as intended. If the transaction in which `:log` is sent fails the logging actor will abort his receive body at line 7. Before re-executing the receive statement the `:logwriter` message will be reprocessed such that the logging actor observes it as a newly arrived message. Consequently, the transaction in line 10-12 can retry together with the receive body of the logging actor independently of the sender of `:logwriter`.

Listing 4.7: Transaction in a receive body that was initiated from within a transaction.

```
1  (create-actor
2    :swap-actor
3    (fn []
4      (receive
5        ([:swap ref-a ref-b sender]]
6            (dosync
7              (let [temp @ref-a]
8                (ref-set! ref-a @ref-b)
9                (ref-set! ref-b temp)))
10             (! sender :finished ))
11     (recur)))
12
13 (create-actor
14   :compute-actor
15   (fn []
16     (receive
17       ([:compute a b result]
18        (dosync
19          (! :swap-actor :swap {:ref-a a :ref-b b :sender *self*})
20          (receive ([:finished ]))
21          (ref-set! result (- @a @b)))))))
```

Listing 4.7 depicts the problem scenario of the transaction in a depending context. The problem in this scenario is that the actor `:compute-actor` its transaction may retry when it tries to write a new value to `result`. As such the swap actor will be invoked again and that will in turn swap the variables again. We want to programmer to remain oblivious of the fact that transactions can retry.

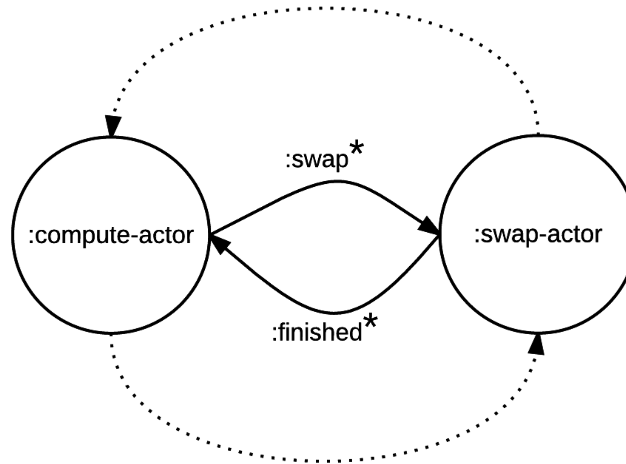
The proposed solution of waiting before committing a transaction does not work here. The message send of the acknowledgement (line 10) only happens after the transaction has committed (line 9). If the `:swap-actor`'s transaction were to wait on the sender's transaction (line 17-20) to commit the program would reside in a deadlock because `:compute-actor`'s transaction will only commit after it received the acknowledgement.

To this end we propose to *stall* a transaction. Intuitively speaking the effects of the swap

actor are part of the side-effects of the transaction running in the `:compute-actor`. As such, they should only become visible when `:compute-actor` finalizes its receive block.

A stalled transaction will not commit and not retry. The transaction simply pauses before executing the commit protocol, i.e., commiting to the global heap. This allows for the depending receive block of `:swap-actor` to fully execute and send the acknowledgement. As a result both actors will reach the end of their receive blocks.

Listing 4.8: Dependency graph of Listing 4.7



At this point `:swap-actor` depends on the transaction of `:compute-actor` and is thus in a depending state. Figure 4.8 depicts this scenario. The `:compute-actor` sends the `:swap` message which carries a dependency to `:swap-actor`. When `:swap-actor` processes this message it becomes depending on the `:compute-actor`, hence the dotted line going from `:swap-actor` to `:compute-actor`.

`:compute-actor` depends on the transaction of the `:swap-actor` because the acknowledgement message carried a dependency. Hence, the actors have a dependency cycle. Figure 4.8 shows this cycle by means of the dotted arrows. `:swap-actor` has an outgoing dotted line to `:compute-actor` and vice versa.

`:compute-actor` can not commit its transaction safely until the transactions on which it depends are never going to fail anymore. The transaction of `:swap-actor` is in the waiting state, which is a promise to not retry anymore. This means that it is bound to commit if all its dependencies resolve. Note that these are all either committed or waiting as well and thus can not retry anymore either. As such, the transaction of `:compute-actor` can safely commit and the rest of the enclosing receive block can be executed. In this case the rest of the receiveblock is empty and `:compute-actor` reaches the end of its receive block. This block has no dependencies and consequently the actor is finished.

`:swap-actor` is waiting to finalize its receive statement. The dependencies of that receive statement have been resolved at the point that `:compute-actor` commited its transaction and thus it is safe to change the state of the stalled transaction to waiting and subsequently commit. The receive statement no longer has any dependencies left either so that can finalize as well.

As a result both transactions committed. They both committed at a point at which it was certain that all the dependencies would no longer abort and retry. Thus making sure that the runtime was not left in an inconsistent state.

4.3 Conclusion

This chapter has introduced the solution proposed in this thesis at a high level. It has shown that the actor model and transactional memory can be combined while still maintain correctness of the program. However, the runtime has to do additional bookkeeping.

5 Operational Semantics

5.1 Introduction

This chapter introduces a formalisation of the rules proposed in this thesis. First we will elaborate on some concepts used throughout the chapter to allow concise reasoning. Next we will introduce the grammar of the semantics and auxilary operations used in the rules. Finally we will elaborate each rule briefly.

5.1.1 Concept

Contexts A context is an implicit status of an execution context. It will be used while reasoning about the current state of an actor.

- `volatile` A transaction is a "volatile" context in the sense that the computations inside the transaction can possibly be executed multiple times. Hence, every computation done inside that block should be either a pure computation (i.e., no side effects), revertible (i.e., undo the side effects) or be used with the possibility in mind that it might be executed multiple times. For example, a print statement does not cause any immediate harm if executed multiple times and can thus be used in a transaction. `launch-missiles()` on the other hand can not be put inside the transaction unless the effects can be undone.
- `depending` A "depending" context is a kind of *volatile* context as well. An actor that receives a message from a *volatile* context executes his receive body in a *depending* context. The reason being is that the sending actor can retry its transaction and this should consequently abort the computation of the receiving actor.
- `stable` Any other context of execution (i.e., not a transaction body) is therefore a *stable* context.

Dependencies A message being sent from a volatile context attaches the current transaction identifier to the message. All actions taken by the receiver after reception of the message are only

valid if and only if the sending actor's transaction succeeds. If it were to fail the receiving actor should discard its computations and wait for the message to be resent. Suppose that the sender sent the message as a consequence of a conditional. If the second attempt of the transaction no longer sends this message by taking the other branch of the conditional the programmer intuitively expects these computations to never take place. To make sure this can be done we propose to make a receiver *depend* on the sending actor if there was a transaction identifier piggybacking on the message, ergo, the message reception executes in a *depending* context.

Dependency Propagation Suppose that an actor a is in a *depending* context because it received a message from an actor b that sent the message from within a transaction and was thus in a *volatile* context. Upon reception of the message, a started a transaction as well and is thus *volatile* and *depending*. If, inside the transaction in actor a a message is being sent the transaction identifier of a should be passed along, as well as the dependency that was piggybacking on the message a received. This means that the message has two dependencies. This approach can build up an arbitrary long list of dependencies when actors keep sending messages in transactions.

Dependency Gathering Since the language we are modeling is an Erlang-like language the programmer can write nested receives. If an actor is in a *depending* context and executes another receive block in that *depending* context it might receive another message which holds another dependency. This means that the actor is now depending on two different actors.

Message statuses When an actor processes a message it has to do some additional bookkeeping in order to act appropriately when the message has to be discarded as a consequence of the context it came from reexecuting. For instance a message that has been sent from a *volatile* context that has now retried. This means that the message has to be discarded. However, it would be cumbersome to keep track of where each message has been sent to and subsequently make the sender delete them. Therefore we propose to make the receiver act appropriately when it notices that a context has retried. This process has a lot of essential complexity. An actor can send or receive messages from a *stable*, *depending* or *volatile* context. We will now discuss each combination of contexts in which a message can be sent from and arrive in.

- `stable` → `stable` The basic case is an actor sending a message in a *stable* context and an actor receiving it in a *stable* context. This means that the sender nor the receiver can revert and retry computations, except at an application level. The receiving actor can thus immediately mark the status of the message as `processed`, indicating it can be deleted from his in-box.

- `stable` → `volatile` When an actor a in a *stable* context sends a message to an actor b who receives it in a *volatile* context we can reason that the sending actor is not susceptible to failure. However, b is. As such, b will mark the message as `tx-processing` to indicate that the current actor is processing the message in a *volatile* context. If the transaction were to retry the actor can mark every message that is flagged as `tx-processing` as `new` again and retry in isolation of the sender.

- `stable` → `depending` applies the same rules as `stable` → `volatile`.

- `volatile` → `stable` If the sending actor a is in a volatile context it will attach its transaction identifier to the message. The receiving actor b will therefore execute the receive behavior in a *depending* context. The message will be marked as `processing` by the receiving transaction.

- `volatile` $\rightarrow$ `volatile` This case applies the same strategy as `stable` $\rightarrow$ `volatile`.
- `volatile` $\rightarrow$ `depending` This case applies the same strategy as `volatile` $\rightarrow$ `stable`.
- `depending` $\rightarrow$ `stable` This case applies the same strategy as `volatile` $\rightarrow$ `stable`.
- `depending` $\rightarrow$ `volatile` This case applies the same strategy as `volatile` $\rightarrow$ `volatile`.
- `depending` $\rightarrow$ `depending` This case applies the same strategy as`volatile` $\rightarrow$ `stable`.

5.1.2 Notation Conventions

Ordered Lists An ordered list will be delimited by square brackets [ ]. A list can be prepended to by using the double colon operator (::). A shorthand for list creation is available too. $a :: (b :: [])$ can be written as $[a\ b]$. A shorthand to determine the length of the list l is $\texttt{len}(l)$. A list l can be deconstructed as $x :: xs$. This implies that x is the first element of the list and xs is the tail of the list.

Tuples A n-ary tuple will be delimited by rectangular brackets. A 2-ary tuple containing 1 and 2 will be shown as $\langle 1,\ 2 \rangle$.

Maps A map is an unordered bag of unique values. An arbitrary map $\mathcal{S}$ can map a value onto another arbitrary value. They will be used primarily to map identifiers onto entities. A map can be accessed like a function call: $\mathcal{S}(x)$ will return the associated value of x out of the map. $\mathcal{S}[a \mapsto b]$ maps a to the new value b overwriting any possible existing value.

Values Values are represented by small letters. An arbitrary set of variables is represented by a calligraphic letter (e.g., $\mathcal{X}$). The entire set of all possible variables in any system will be denoted by the name in bold (e.g., "**Variables**").

5.2 Grammar

5.2.1 Values

As in most calculi there is a minimal set of terms and values required to model standard behavior. Table 5.1 lists all the values that are built into the semantics.

- **Primitives** is the set of all built-in functions such as `-` or $+$.
- **Actor Ids** represents the set of all globally unique actor identifiers.A new unique actor id can be obtained by writing "a **fresh**".
- **Numbers** are the set of all natural numbers, $\mathbb{N}$.
- **Boolean Values** is the set of `true` and `false`.
- **Variables** is the set of all possible variables. These are globally unique. A new unused variable identifier can be obtained by writing "x **fresh**".
- **Message Ids** is the set of all possible message ids. In actual implementations a message is usually labeled by a string (e.g., "compute"). However, for operational semantics it is possible, without loss of generality, to use abstract globally unique identifiers.

$f \in$	$\mathcal{F}$	$\subseteq$	**Primitives**	::=	$\{+, -, *, ..\}$
$a \in$	$\mathcal{A}$	$\subseteq$	**Actor Ids**	::=	$\{a_1, a_2, .., a_n\}$
$i \in$	$\mathcal{N}$	$\subseteq$	**Numbers**	::=	$\mathbb{N}$
$b \in$	$\mathcal{B}$	$\subseteq$	**Boolean Values**	::=	$\{\texttt{true}, \texttt{false}\}$
$x \in$	$\mathcal{X}$	$\subseteq$	**Variables**	::=	$\{x_1, x_2, ..x_n\}$
$r \in$	$\mathcal{R}$	$\subseteq$	**Transactional Variables**	::=	$\{r_1, r_2, .., r_n\}$
$m \in$	$\mathcal{M}$	$\subseteq$	**Message Ids**	::=	$\{m_1, m_2, .., m_n\}$

Table 5.1: **Values**

5.2.2 Expressions

Standard Expressions

Table 5.2 depicts the terms of the operational semantic. As one would expect all values shown in Table 5.1 are proper terms. Table 5.3 depicts the standard expressions to do basic computations: application of abstractions, application of a primitive function and chaining of expressions using ";".

$v \in$	$\mathcal{V}$	::=		*Values*
		|	f	
		|	r	
		|	a	
		|	i	
		|	b	
		|	x	
		|	m	
		|	$\lambda x.\mathcal{E}$	*Abstractions*

Table 5.2: **Values**

$\mathcal{E}$	::=	$\mathcal{V}$	*Terms*
	|	$(\mathcal{E}\ \mathcal{E})$	*Application*
	|	$(f\ v_1, .., v_n)$	*Primitive app*
	|	$\mathcal{E}_t$	
	|	$\mathcal{E}_a$	
	|	$\mathcal{E}\ ;\ \mathcal{E}$	*Chaining*

Table 5.3: **Regular Terms**

Actor Expressions

To model actor behavior there are not many terms needed. Evaluating "`spawn` $a\ \mathcal{E}$" creates a new actor with identifier a and behavior $\mathcal{E}$. "$!\ a\ m\ [\mathcal{E}]$" sends a message m to actor a with a possible payload. The payload is represented by an ordered list of standard expressions. "`receive` $[\langle m\ [x]\rangle\ \mathcal{E}]$" starts a receive block with one or more receive cases. Recall that [] represents an ordered list. A receive block defines a message identifier for the message (m), a list of unbound variables to bind to the payload ($[x]$) and a specific behavior to evaluate upon reception of the message ($\mathcal{E}$).

$\mathcal{E}_a$	$::=$	`spawn` $a\ \mathcal{E}$	*Spawn new actor*
	$\mid$	`!` $a\ m\ [\mathcal{E}]$	*Message send*
	$\mid$	`receive` $[\langle m\ [x]\rangle\ \mathcal{E}]$	*Receive*

Table 5.4: **Actor Terms**

Transaction Terms

The terms relating to transactional memory are `atomic`, `set!` and `read`. `atomic` takes an expression $\mathcal{E}$ which will be executed in a transaction. It is only in such an expression that transactional variables can be written to using the `set!` term. It requires a unique identifier and a new value to be written to it. Reading a transactional variable can be done anywhere by using the `read` term followed by a unique identifier.

$\mathcal{E}_t$	$::=$	`atomic` $\mathcal{E}$	*Transaction*
	$\mid$	`set!` $r\ v$	*Set transactional var*
	$\mid$	`read` r	*Read transactional var*

Table 5.5: **Transaction Terms**

Syntactic Sugar

Syntactic sugar is defined syntax that enables easier programming, yet only uses constructs that are already present in the language.

The chaining of expressions is syntactic sugar where the second expression is evaluated in a lambda that takes as a parameter the first expression.

$$\mathcal{E}_1\ ;\ \mathcal{E}_2 \quad \equiv \quad ((\lambda\ x\ .\ \mathcal{E}_2)\ \mathcal{E}_1) \qquad \textit{Transaction}$$

Table 5.6: **Syntactic Sugar**

5.2.3 Semantic Domains

Semantic domains are entities that are used in the following rules. An entity is represented by a calligraphic letter (e.g., $\mathbb{D}$).

Messages

Table 5.7 depicts all the entities that are related to messages. Each one will explained in detail.

- $\mathbb{M}$ is an actual message. It can be deconstructed as a 6-tuple. The first entry m represents the message identifier. a_s and a_r represent the actor identifier of the sender and receiver respectively. z represents the status of the message (e.g., `tx-processing`). $[v]$ represents the payload of the message. A message can contain zero or more values. These will be matched with a receive block's unbound variables. Finally $\mathbb{D}$ represents the transaction identifiers that piggyback on the message.
- $z \in \mathbb{Z}$ represents the set of message statuses. The statusses are listed below.
- $\mathbb{I}$ is an in-box. An in-box is an ordered list of messages and is unique per actor.

- Σ is a mapping from actor identifiers to in-boxes. It will be used to contain all in-boxes in a global configuration.

Message	$\mathbb{M}$		$::=$	$\langle m, a_s, a_r,\ z,\ [v],\ \mathbb{D}\rangle$
Message Status	$z \in \mathbb{Z}$	$\subseteq$ **Flags**	$::=$	$\{\texttt{processing, new, tx-processing, processed}\}$
Message In-box	$\mathbb{I}$		$::=$	$[\mathbb{M}]$
In-boxes	Σ		$=$	$a \mapsto \mathbb{I}$

Table 5.7: Message Entities

Actors

Table 5.8 represents the entities that relate to actors. Each of them will be discussed in detail.

- $\mathbb{D}$ represents a set of transaction identifiers.
- $\mathbb{A}$ represents an actor. The first part of the entity, $\mathbb{D}$, contains all the dependencies that are introduced while the actor is in a *depending* or *stable* context. When the actor is in the *stable* context $\mathbb{D}$ is empty. When the first element is inserted the context of the actor implicitly changes to *depending*. $\mathbb{D}_\tau$ contains all the dependencies that are introduced while the actor is in a *volatile* context, (i.e., running a transaction). τ will hold the transaction identifier of the currently running transaction and is $\perp$ if no transaction is running. $\mathcal{E}$ is the current evaluation context. $\mathcal{E}_r$ will hold a backup of the evaluation context. If an actor changes from a *stable* context to a *depending* context (i.e., a message is received introducing the first dependency into $\mathbb{D}$) the current evaluation context is stored in $\mathcal{E}_r$. If at the end of the receive block the actor concludes that any dependency in $\mathbb{D}$ has aborted it has to retry its own evaluation of the receive block. $\mathcal{E}$ will then be replaced by $\mathcal{E}_r$.
- Θ contains a mapping from actor identifiers a to actor entities $\mathbb{A}$.

Dependencies	$\mathbb{D}$	$::=$	$\{\tau\}$
Actor	$\mathbb{A}$	$::=$	$\langle \mathbb{D}, \mathbb{D}_\tau, \tau, \mathcal{E},\ \mathcal{E}_r\rangle$
Actors	Θ	$=$	$a \mapsto \mathbb{A}$

Table 5.8: Actor Entities

Transactional Memory Entities

- $\tau \in \mathbb{U}$ represents a unique transaction identifier. Each time a new transaction is started a new globally unique identifier is created using "τ **fresh**".
- $s \in \mathbb{S}$ represents a transaction status. A transaction can be `running` which means it is simply running. `aborted` means that the transaction encountered a conflict while reading or writing a transactional variable and consequently aborted. An aborted transaction never goes back to any other status, instead a new transaction is created. `waiting` means that the transaction reached the syntactic end of its body but has not yet been committed. `committed` means that the transaction has been committed and the changes are visible in the global heap of transactional variables. Finally, `stalled` indicates that a transaction has reached the syntactic end of its body but was not able to commit. If a transaction has been started in a *depending* context it can not commit its changes at the end of the body.

The reason being that the context it is executed in can still abort and be re-executed. Hence, the changes of the transaction can only be committed at the point the *depending* context is sure it can no longer be retried.

- Δ is a heap that contains all the transactional variables and their value. As explained before *Multi version Concurrency Control* is used for the software transactional memory. As such, a single transactional variable is represented as $\langle [(i, v)],\ \tau \rangle$. The first list represents the history of the transactional variable. v represents the value and i represents the write point (when it was written). The second part is a transaction identifier. If a transaction is running and has written a value to a ref the transaction identifier is stored there. This tells other transactions that try to write to it that it is in use and they should retry.
- $\mathbb{T}$ represents a transaction entity. i_τ is the write point at which the transaction started. The configuration (discussed below) contains the global write point. i_τ is a copy of the value of the global write point taken at the moment the transaction is started (i.e., when evaluating `atomic`).
- Ω is a mapping from transaction identifiers to actual transaction entities. It is used for all actors to be able to access the current status of a transaction.
- Ψ is mapping of transactional variable identifiers to values. A transaction will keep track of the current value of a transactional variable when it has written to it or read from it. It is this map that will be merged with the global heap (Δ) when the transaction commits.
- $\mathcal{R}$ is a set of written variables. Each time a transaction writes a new value to a transactional variable this variable is stored in $\mathcal{R}$.
- $\mathcal{E}_\tau$ is used to store a backup of the transaction's execution context. In case of failure it is possible to restore the execution context and reexecute the entire transaction.

Transaction id	$\tau \in \mathbb{U}$	::=	$\{t_1, t_2, .., t_n\}$
Transaction Status	$s \in \mathbb{S}$	::=	$\{\texttt{running}, \texttt{waiting}, \texttt{aborted}, \texttt{committed}, \texttt{stalled}\}$
Ref Heap	Δ	=	$r \mapsto \langle [(i, v)],\ \tau \rangle$
Transaction	$\mathbb{T}$	::=	$\langle i_\tau,\ s,\ \Psi_\tau,\ \mathcal{R},\ \mathcal{E}_\tau \rangle$
Transactions	Ω	=	$\tau \mapsto \mathbb{T}$
Local Heap	Ψ	=	$r \mapsto \mathcal{V}$

Configuration Entity

A configuration $\mathbb{K}$ consists of a global heap of transactional variables (Δ), a global write point (i), a map of actor identifiers to in-boxes (Σ), a map of transaction identifiers to transactions (Ω) and finally a map of actor identifiers to actor entities (Θ).

Configuration	$\mathbb{K}$	::=	$\langle \Delta,\ i,\ \Sigma,\ \Omega,\ \Theta \rangle$

5.2.4 Auxiliary Operations

Below is a list of auxiliary operations that are used throughout the operational semantics. The main purpose of them is to make the rules easier on the eyes.

Operations on Actors

$$
\begin{array}{rcll}
\texttt{intransaction}(\langle _, _, \tau, _, \ _\rangle) & = & \textit{true} & \text{when } \tau \neq \bot \\
\texttt{intransaction}(\langle _, _, \tau, _, \ _\rangle) & = & \textit{false} & \text{otherwise} \\
\texttt{no-deps}(\langle \mathbb{D}, _, _, _, \ _\rangle) & = & \textit{true} & \text{when } \mathbb{D} = \emptyset \\
\texttt{no-deps}(\langle \mathbb{D}, _, _, _, \ _\rangle) & = & \textit{false} & \text{otherwise}
\end{array}
$$

Operations on Transactions

Transaction Status Listed below are a few "getters" that simply return whether a transaction is in a particular status.

$$
\begin{array}{rcll}
\tau_{\checkmark}(\langle _, \ s, \ _, \ _, \ _\rangle) & = & \textit{true} & \text{when } s = \texttt{committed} \\
\tau_{\checkmark}(\langle _, \ s, \ _, \ _, \ _\rangle) & = & \textit{false} & \text{otherwise} \\
\tau_{\rightarrow}(\langle _, \ s, \ _, \ _, \ _\rangle) & = & \textit{true} & \text{when } s = \texttt{running} \\
\tau_{\rightarrow}(\langle _, \ s, \ _, \ _, \ _\rangle) & = & \textit{false} & \text{otherwise} \\
\tau_{\times}(\langle _, \ s, \ _, \ _, \ _\rangle) & = & \textit{true} & \text{when } s = \texttt{aborted} \\
\tau_{\times}(\langle _, \ s, \ _, \ _, \ _\rangle) & = & \textit{false} & \text{otherwise} \\
\tau_{\diamond}(\langle _, \ s, \ _, \ _, \ _\rangle) & = & \textit{true} & \text{when } s = \texttt{waiting} \\
\tau_{\diamond}(\langle _, \ s, \ _, \ _, \ _\rangle) & = & \textit{false} & \text{otherwise} \\
\tau_{\bowtie}(\langle _, \ s, \ _, \ _, \ _\rangle) & = & \textit{true} & \text{when } s = \texttt{stalled} \\
\tau_{\bowtie}(\langle _, \ s, \ _, \ _, \ _\rangle) & = & \textit{false} & \text{otherwise}
\end{array}
$$

Operations on Messages

Reprocess When `reprocess` is applied on any given message and a set of transaction statuses (e.g., {`running`, `waiting`}) the message will be marked as `new` if the status of the message is an element of the given set of statusses. Otherwise the original message is returned.

$$
\begin{array}{rcll}
\texttt{reprocess}(\langle m, a_s, a_r, \ z, \ [v], \ \mathbb{D}\rangle, \ \mathbb{Z}) & = & \langle m, a_s, a_r, \ \texttt{new}, \ [v], \ \mathbb{D}\rangle & \text{when} z \in \mathbb{Z} \\
\texttt{reprocess}(m, \ _) & = & m \text{ otherwise} &
\end{array}
$$

Invalid determines if a given message is invalid. A message is invalid when it is carrying any dependency that at this point in time has been aborted. Intuitively this means that the *volatile* context has retried and thus might send the message again.

$$
\begin{array}{rcl}
\texttt{invalid}(m) & = & \textit{true} \text{ when } m = \langle _, \ _, \ _, \ \texttt{tx-processing}, \ _, \ \mathbb{D}_m\rangle \text{ and } \exists \ \tau \in \mathbb{D}_m : \tau_{\times}(\tau) \\
\texttt{invalid}(m) & = & \textit{true} \text{ when } m = \langle _, \ _, \ _, \ \texttt{processing}, \ _, \ \mathbb{D}_m\rangle \text{ and } \exists \ \tau \in \mathbb{D}_m : \tau_{\times}(\tau) \\
\texttt{invalid}(m) & = & \textit{false} \text{ otherwise}
\end{array}
$$

Inuse determines if a given message is "in use". A message is in use if has been read from the in-box by a receive block. If it has not been read it should have the `new` status. If it has already been processed it should have the `processed` status.

$$\begin{aligned}\texttt{inuse}(\langle _,\ _,\ _,\ s,\ _,\ _\rangle) &= \textit{true} \text{ when } s \in \{\texttt{tx-processing},\ \texttt{processing}\}\\ \texttt{inuse}(m) &= \textit{false} \text{ otherwise}\end{aligned}$$

Mark takes a message and marks it according to the status of the actor. If the reading actor is in a *volatile* context (i.e., in a transaction) the message should be marked as tx-processing. If the actor is in a *depending* context (i.e., $\mathbb{D} \neq \emptyset$) the message should be marked as processing. Otherwise the message can be marked as processed immediately

$$\texttt{mark}(\langle m, a_s, a_r,\ z,\ [v],\ \mathbb{D}\rangle) = \begin{cases}\langle m, a_s, a_r,\ \texttt{tx-processing},\ [v],\ \mathbb{D}\rangle & \text{when } \texttt{intransaction}(a_r)\\ \langle m, a_s, a_r,\ \texttt{processed},\ [v],\ \mathbb{D}\rangle & \text{when } \texttt{no-deps}(a_r) \textit{ and } \mathbb{D} = \emptyset\\ \langle m, a_s, a_r,\ \texttt{processing},\ [v],\ \mathbb{D}\rangle & \text{otherwise}\end{cases}$$

Operations on In-boxes

Releaserefs takes a global heap of transactional variables and a set of written refs. Recall that a transactional variable stores which transaction is currently running and has written to it. When a transaction aborts or commits these values should be cleared in order for other transaction to safely write to it. Releaserefs will create a new global heap Δ' that removes all transaction identifiers if and only if the transactional variable is $\in \mathcal{R}$.

$$\texttt{releaserefs}(\Delta,\ \mathcal{R}) = \forall x \in \textit{dom}(\Delta) : \begin{cases}\Delta(x) & \text{when } x \notin \mathcal{R}\\ \langle l,\ \bot\rangle & \text{when } x \in \mathcal{R}\\ & \text{where } \langle l,\ _\rangle = \Delta(x)\end{cases}$$

Resetin-box is a helper that will be used when an actor has to revert back to a previous execution context due to either a failing transaction or a failing receive block. It does so by iterating over the entire in-box. If a message is invalid it will be removed, otherwise it is marked as new such that it can be received again.

$$\begin{aligned}\texttt{resetInbox}(m :: ms) &= \texttt{resetInbox}(ms) \text{ when } \texttt{invalid}(\text{m})\\ \texttt{resetInbox}(m :: ms) &= m' :: \texttt{resetInbox}(ms) \text{ otherwise}\\ &\quad\ \text{where } m' = \texttt{reprocess}(m,\ \{\texttt{tx-processing},\ \texttt{processing}\})\\ \texttt{resetInbox}([]) &= []\end{aligned}$$

Cleanin-box will take an in-box and remove each message that has been received up to that point (i.e., marked as tx-processing or processing). The function will be used when an actor finishes a receive block successfully, because at that point all messages have been processed and can thus be removed.

$$\begin{aligned}\texttt{cleanInbox}(m :: ms) &= \texttt{cleanInbox}\ (ms) \text{ when } \texttt{inuse}(\text{m})\\ \texttt{cleanInbox}(m :: ms) &= m :: \texttt{cleanInbox}(ms) \text{ otherwise}\\ \texttt{cleanInbox}([]) &= []\end{aligned}$$

Operations on Transactional Variables

Value Given the value of a transactional variable and a write point value will return the latest value of the transactional variable that has a write point that is less than or equal to the given

write point. Recall that a transactional variable can be read even though a transaction may have committed to it after the reading transaction started.

$$
\begin{array}{lcl}
\texttt{value}((v,i) :: l, i_{\mathcal{T}}) & = & v \text{ when } i_{\mathcal{T}} \geq i \\
\texttt{value}((v,i) :: l, i_{\mathcal{T}}) & = & \texttt{value}(l, i_{\mathcal{T}}) \text{ otherwise} \\
\texttt{value}([], _) & = & \bot
\end{array}
$$

Operations for receive

Equal is used to pattern match a receive case with a message. A receive case matches to a given message if the message identifier is the same and when the number of parameters for that receive case match the number of values that are passed along in the message.

$$
\begin{array}{rcl}
\texttt{equal}(\langle m_1,\ _,\ _,\ _,\ [v],\ _\rangle,\ \langle m_2,\ [x]\rangle) & = & \textit{true} \text{ when } \texttt{len}([v]) = \texttt{len}([x]) \text{ and } m_1 = m_2 \\
\texttt{equal}(_,\ _) & = & \textit{false} \text{ otherwise}
\end{array}
$$

matchMsg takes a single receive case and an in-box. It will iterate over the entire in-box to find a message that is `equal` to the case. If such a message is found it returns a triple containing a new in-box, the marked message and the behavior defined in the case. The new in-box is defined as the old in-box, except that the matching message is replaced by a marked one. The third element of the triple is the behavior that was defined in the receive case.

$$
\begin{array}{lcl}
\texttt{matchMsg}(\langle m,\ [x]) \rightarrow \mathcal{E}\rangle,\ [\,]) & = & \bot \\
\texttt{matchMsg}(\langle m,\ [x]) \rightarrow \mathcal{E}\rangle,\ m' :: ms) & = & (\texttt{mark}(m') :: ms,\ \texttt{mark}(m'),\ \mathcal{E}) \\
 & & \text{when } \texttt{equal}(m',\ \langle m,\ [x]\rangle) \\
\texttt{matchMsg}(\textit{case},\ m' :: ms) & = & (m' :: \mathbb{I}',\ m,\ \mathcal{E}) \\
 & & \text{where } (\mathbb{I}', m, \mathcal{E}) = \texttt{matchMsg}(\textit{case},\ ms) \\
 & & \text{otherwise}
\end{array}
$$

matchCases takes a list of receive cases. The function will try to find a suitable message for each case. If no suitable message is found for the first case it tries the second one and so forth.

$$
\begin{array}{lcl}
\texttt{matchCases}([\,],\ \mathbb{I}) & = & \bot \\
\texttt{matchCases}(\textit{case} :: cs,\ \mathbb{I}) & = & \texttt{matchMsg}(\textit{case},\ \mathbb{I}) \\
 & & \text{when } \texttt{matchMsg}(\textit{case},\ \mathbb{I}) \neq \bot \\
\texttt{matchCases}(\textit{case} :: cs,\ \mathbb{I}) & = & \texttt{matchCases}(cs,\ \mathbb{I}) \\
 & & \text{otherwise}
\end{array}
$$

5.3 Rules

5.3.1 Sending

Sending a message means that an actor creates a new message which is placed into the in-box of the receiving actor. To this end we take the in-box of the receiving actor out of Σ and prepend a new message to it. The dependencies in the message are the $\cup$ of the parent dependencies of

the actor ($\mathbb{D}$), the dependencies of a possible running transaction ($\mathbb{D}_\tau$) and the possible running transaction (τ). Notice that a newly constructed message is always flagged as `new`. Finally the reduction context of the sending actor is reduced to *nil*.

[Send 1]

$$\frac{\mathbb{I}_r = \Sigma(a_r)}{\begin{array}{c}\langle \Delta,\ i_g,\ \Sigma,\ \Omega,\ \Theta[a_s \mapsto \langle \mathbb{D}, \mathbb{D}_\tau,\ \tau,\ e_\Box[!\ a_r\ m\ [v]],\ \mathcal{E}_r\rangle]\rangle \quad \longrightarrow \\ \langle \Delta,\ i_g,\ \Sigma[a_r \mapsto [\langle m,\ a_s,\ a_r,\ \texttt{new},\ [v],\ \mathbb{D} \cup \mathbb{D}_\tau \cup \{\tau\}\rangle :: \mathbb{I}_r]],\ \Omega,\ \Theta[a_s \mapsto \langle \mathbb{D}, \mathbb{D}_\tau,\ \tau,\ e_\Box[nil],\ \mathcal{E}_r\rangle]\ \rangle\end{array}}$$

5.3.2 Receiving

Receiving messages is one of the most important parts of this entire operational semantics. We will discuss each case separately and in depth.

Because the following operational semantics allows one to send and receive messages in- and outside of a transaction there are different steps to be taken in each scenario. The obstacle to overcome is to identify when a message is sent or received in a *volatile* context. This means that the flow of execution that sends or receives the message can be rolled back due to a transaction that retries. In consequence the message send or receive has to be undone as well, including subsequent sends or receives.

rcv 1 models the most basic scenario. An actor a_r receives a message that holds no dependencies. This means that once a_r has started the receive block there is no scenario in which it is forced to jump back to a previous execution point. The sending actor did not pass dependencies along with the message meaning that the message was not sent in a transaction, nor was the message sent from an actor that was depending on other actors. Hence, the message can be immediately flagged as `processed`.

[rcv 1]

$$\frac{(\mathbb{I}_r^{'},\ \langle m_m,\ a_s,\ a_r,\ \texttt{processed},\ \overline{v},\ \emptyset\rangle,\ \mathcal{E}_m) = \texttt{matchCases}(\overline{[(m,\ [x]) \rightarrow \mathcal{E}]},\ \Sigma(a_r))}{\begin{array}{c}\langle \Delta,\ i_g,\ \Sigma,\ \Omega,\ \Theta[a_r \mapsto \langle \emptyset, \emptyset,\ \bot,\ e_\Box[\texttt{receive}\ \overline{[(m,\ [x]) \rightarrow \mathcal{E}]}\ ],\ \bot\rangle]\rangle \quad \longrightarrow \\ \langle \Delta,\ i_g,\ \Sigma[a_r \mapsto \mathbb{I}_r^{'}],\ \Omega,\ \Theta[a_r \mapsto \langle \emptyset, \emptyset,\ \bot,\ e_\Box[[[v]/[x]]\mathcal{E}_m],\ \bot\rangle]\ \rangle\end{array}}$$

rcv 2 is a rule that introduces dependencies. When an actor a_r is executing a receive block and has no other dependencies, nor is executing a transaction, a_r is in a *stable* context. There is nothing that can stop its control flow except an application-level exception. Upon reception of a message that holds a dependency, a_r becomes *depending* on the sending actor a_s or the actor on which a_s is depending (*dependency propagation*). If a_r has to roll back due to one of the dependencies failing, there has to be a *stable* execution context to jump back to. This receive rule is that exact point where it needs to happen. By taking the entire current execution context ($e_\Box$[`receive` $\overline{(m,\ [x]) \rightarrow \mathcal{E}}$]) and storing it in $\mathcal{E}_r$ in a_r we effectively save the current state of execution. Finally the dependencies passed in the message will be the new value of $\mathbb{D}$ because it is the only set of dependencies at this point and no $\cup$ is required.

A new statement is chained to the evaluation of the message body: `end-rcv`. This allows for specific rules to apply when the receive block is evaluated.

[rcv 2]

$$\frac{(\mathbb{I}_r^{'},\ \langle m_m,\ a_s,\ a_r,\ \texttt{processing},\ \overline{v},\ \mathbb{D}_m\rangle,\ \mathcal{E}_m) = \texttt{matchCases}([(m,\ [x]) \rightarrow \mathcal{E}],\ \Sigma(a_r))}{\begin{array}{c}\langle \Delta,\ i_g,\ \Sigma,\ \Omega,\ \Theta[a_r \mapsto \langle \emptyset,\ \emptyset,\ \bot,\ e_\square[\texttt{receive}\ [(m,\ [x]) \rightarrow \mathcal{E}]\ ],\ \bot\rangle]\rangle \quad \longrightarrow \\ \langle \Delta,\ i_g,\ \Sigma[a_r \mapsto \mathbb{I}_r^{'}],\ \Omega,\ \Theta[a_r \mapsto \langle \mathbb{D}_m,\ \emptyset,\ \bot,\ e_\square[[[v]/[x]]\mathcal{E}_m; \texttt{end-rcv}],\ e_\square[\texttt{receive}\ [(m,\ [x]) \rightarrow \mathcal{E}]\ ]\rangle]\ \rangle\end{array}}$$

rcv 3 applies when the receiving actor a_r does not have any current transaction running but the message carries dependencies ($\mathbb{D}_m$). This means that the receiving context is not volatile, but the sending context is. Intuitively this means that the message can become invalid and the receiving actor might have to act accordingly. As a result we add the dependencies that are present in the message (i.e., the transaction identifiers) to the already gathered dependencies, $\mathbb{D}$. The current evaluation context is reduced to the evaluation of the expression that was returned by the `match` function, $\mathcal{E}_m$, in which we bind the unbound variables $[x]$ to the payload of the received message, $[v]$. Finally, the new in-box ($\mathbb{I}_r$), which contains the marked message is updated in the global in-box map.

[rcv 3]

$$\frac{(\mathbb{I}_r^{'},\ \langle m_m,\ a_s,\ a_r,\ \texttt{processing},\ [v],\ \mathbb{D}_m\rangle,\ \mathcal{E}_m) = \texttt{matchCases}(\overline{(m,\ [x]) \rightarrow \mathcal{E}},\ \Sigma(a_r))}{\begin{array}{c}\langle \Delta,\ i_g,\ \Sigma,\ \Omega,\ \Theta[a_r \mapsto \langle \mathbb{D}, \emptyset,\ \bot,\ e_\square[\texttt{receive}\ [(m,\ [x]) \rightarrow \mathcal{E}]\ ],\ \mathcal{E}_r\rangle]\rangle \quad \longrightarrow \\ \langle \Delta,\ i_g,\ \Sigma[a_r \mapsto \mathbb{I}_r^{'}],\ \Omega,\ \Theta[a_r \mapsto \langle \mathbb{D} \cup \mathbb{D}_m,\ \emptyset,\ \bot,\ e_\square[[[v]/[x]]\mathcal{E}_m],\ \mathcal{E}_r\rangle]\ \rangle\end{array}}$$

rcv 4 When a receiving actor a_r is in a *depending* context but receives a message m from a *stable* context, it is still possible for a_r to jump back to a previous execution context. To make sure that we can receive this message again it is marked as `processing`. If we were to remove each message from the queue upon reception, a_r would not be able to receive the message in the second try. If at some point a_r jumps back to a previous execution context the message can be identified and remarked as `new`.

[rcv 4]

$$\frac{(\mathbb{I}_r^{'},\ \langle m_m,\ a_s,\ a_r,\ \texttt{processing},\ \overline{v},\ \emptyset\rangle,\ \mathcal{E}_m) = \texttt{matchCases}([(m,\ [x]) \rightarrow \mathcal{E}],\ \Sigma(a_r))}{\begin{array}{c}\langle \Delta,\ i_g,\ \Sigma,\ \Omega,\ \Theta[a_r \mapsto \langle \mathbb{D}, \emptyset,\ \bot,\ e_\square[\texttt{receive}\ [(m,\ [x]) \rightarrow \mathcal{E}]\ ],\ \mathcal{E}_r\rangle]\rangle \quad \longrightarrow \\ \langle \Delta,\ i_g,\ \Sigma[a_r \mapsto \mathbb{I}_r^{'}],\ \Omega,\ \Theta[a_r \mapsto \langle \mathbb{D}, \emptyset,\ \bot,\ e_\square[[[v]/[x]]\mathcal{E}_m],\ \mathcal{E}_r\rangle]\ \rangle\end{array}}$$

rcv 5 models the same scenario as `rcv 4`, except in this case a_r is in a *volatile* context. This changes the point to where the flow of execution can jump back. In case a_r has a dependency that has become invalid by the end of the receive block it will jump back to $\mathcal{E}_r$. When the transaction τ, that a_r is executing, aborts, a_r jumps back to the start of the transaction. This means that all messages received since that point have to be marked for reprocessing. The actor might have already received messages before the atomic block, thus we differentiate between these two by flagging messages received inside an atomic block as `tx-processing`.

[rcv 5]

$$\frac{(\mathbb{I}_r^{'},\ \langle m_m,\ a_s,\ a_r,\ \texttt{tx-processing},\ \overline{v},\ \mathbb{D}_m\rangle,\ \mathcal{E}_m) = \texttt{matchCases}([(m,\ [x]) \rightarrow \mathcal{E}],\ \Sigma(a_r))}{\begin{array}{c}\langle \Delta,\ i_g,\ \Sigma,\ \Omega,\ \Theta[a_r \mapsto \langle \mathbb{D}, \mathbb{D}_\tau,\ \tau,\ e_\square[\texttt{receive}\ [(m,\ [x]) \rightarrow \mathcal{E}]\ ],\ \mathcal{E}_r\rangle]\rangle \quad \longrightarrow \\ \langle \Delta,\ i_g,\ \Sigma[a_r \mapsto \mathbb{I}_r^{'}],\ \Omega,\ \Theta[a_r \mapsto \langle \mathbb{D}, \mathbb{D}_\tau \cup \mathbb{D}_m,\ \tau,\ e_\square[[[v]/[x]]\mathcal{E}_m],\ \mathcal{E}_r\rangle]\ \rangle\end{array}}$$

5.3.3 End Receive

Ending a receive block requires to make sure a few things are in order. First of all, we have to make sure that all dependencies have been resolved (i.e., `committed` or `waiting`). Second of all we need to clean up remaining dependencies by removing them from the actor entity.

Note that it is impossible to evaluate an `end-rcv` inside a transaction. According to rule `rcv 2`, `end-rcv` only is introduced when the receiver is in a *stable* context. If there were a transaction running all dependencies would be stored in $\mathbb{D}_\tau$. Hence, evaluating `end-rcv` is a guarantee that no transaction is running.

end-rcv 1 checks all the dependencies of the receive block in $\mathbb{D}$ for their status. If they are all in a committed status this receive block can safely end because it will not be forced to retry by its dependencies. It is safe to discard the backup of the execution context $\mathcal{E}_r$ and the set of dependencies, $\mathbb{D}$. $\tau_\checkmark$`()` only holds when a transaction is in the `committed` status. Note that no transaction τ may be running in a_r. This would imply that the entire receive is part of a volatile context, τ and may be executed again. `cleanInbox` removes all messages from the in-box that have been processed in this receive block (i.e., all messages marked `tx-processing`and `processing`).

[end-rcv 1]

$$\frac{\forall\ \tau \in \mathbb{D} : \tau_\checkmark(\tau) \qquad \mathbb{I}' = \texttt{cleanInbox}(\Sigma(a))}{\begin{array}{c}\langle \Delta,\ i_g,\ \Sigma,\ \Omega,\ \Theta[a \mapsto \langle \mathbb{D}, \emptyset,\ \bot,\ e_\square[\texttt{end-rcv}],\ \mathcal{E}_r\rangle]\rangle \quad \longrightarrow \\ \langle \Delta,\ i_g,\ \Sigma[a \mapsto \mathbb{I}'],\ \Omega,\ \Theta[a \mapsto \langle \emptyset,\ \emptyset,\ \emptyset,\ \bot,\ e_\square[nil],\ \bot\rangle]\ \rangle\end{array}}$$

end-rcv-fail 1 is the scenario the actor has reached the the end of a receive block and one of the dependencies (τ') has aborted. Consequently, the currently running transaction, if there is one, has to abort as well. This means that we might receive a message again due to τ' retrying. Therefore, every message in our in-box that we processed during this receive block and possible transaction (i.e., the messages marked `processing` or `tx-processing`) have to be processed again. If a message contains dependencies that are invalid it has to be flagged as `processed`. All other messages have to be reflagged as `new` such that our actor can reprocess them. Note that this rule also applies when there is a transaction running. This would then be a `stalled` transaction (rule `commit 3`), hence it has not yet been committed. To make it known to all other actors we mark the current transaction as `aborted`.

[end-rcv-fail 1]

$$\frac{\exists\ \tau \in \mathbb{D} : \tau_\times(\tau) \qquad \mathbb{I}' = \texttt{resetInbox}(\Sigma(a)) \qquad \Omega' = \Omega[\tau \mapsto \langle \tau,\ i_\tau,\ \texttt{aborted},\ \Psi_\tau,\ \mathcal{R},\ \mathcal{E}_c\rangle],}{\begin{array}{c}\langle \Delta,\ i_g,\ \Sigma,\ \Omega,\ \Theta[a \mapsto \langle \mathbb{D}, \mathbb{D}_\tau,\ \tau,\ e_\square[\texttt{end-rcv}],\ \mathcal{E}_r\rangle]\rangle \quad \longrightarrow \\ \langle \Delta,\ i_g,\ \Sigma[a \mapsto \mathbb{I}'],\ \Omega,\ \Theta[a \mapsto \langle \emptyset, \emptyset,\ \bot,\ \mathcal{E}_r,\ \bot\rangle]\rangle\end{array}}$$

end-rcv 2 applies when there is a `stalled` transaction running (τ). This happens when an atomic block was executing in a *depending* context. (i.e., $\mathbb{D} \neq \emptyset$). The current receive block can only finalize when these transaction dependencies are all resolved as well. If there is a transaction τ running we are sure that it must be in the `stalled` status. Recall that `end-rcv` is only added to the evaluation context if the context is *stable*. Hence, `end-rcv 2` is evaluated at the end of the receive block that introduced the first dependency. If τ would not be in the `stalled` status it would mean that it did not try to commit yet and thus encloses this receive. But then all dependencies would reside in $\mathbb{D}_\tau$ and no `end-rcv` would have been introduced.

When this rule applies we first change the status of the transaction to `waiting`. This allows for the `commit 1` and `commit 4` rules to apply. This allows all transactions in the dependency group ($\mathbb{D}_\tau$) to commit safely. When this happens τ will be cleared from the actor, subsequently allowing the regular `end-rcv` rules to apply. Note that a check is made to ensure that all dependencies in $\mathbb{D}$ are resolved. The transaction can only and only then commit when these dependencies have been met and thus the receive block is sure to not abort. Note that the dependencies in $\mathbb{D}$ have to be in `committed` or `waiting` status otherwise it would be possible to commit τ and in the mean while any of the dependencies in $\mathbb{D}$ could abort and thus abort the enclosing receive block as well, making it impossible for τ to revert its changes from the global heap.

[end-rcv 2]

$$\frac{\forall \tau' \in \mathbb{D}_\tau \ : \tau_{\diamond}(\tau') \vee \tau_{\checkmark}(\tau') \qquad \Omega' = \Omega[\tau \rightarrow \langle \tau,\ i_\tau,\ \texttt{waiting},\ \Psi_\tau,\ \mathcal{R},\ \mathcal{E}_c\rangle],}{\begin{array}{c}\langle \Delta,\ i_g,\ \Sigma,\ \Omega,\ \Theta[a \mapsto \langle \mathbb{D}, \mathbb{D}_\tau,\ \tau,\ e_\square[\texttt{end-rcv}],\ \mathcal{E}_r\rangle]\rangle \qquad \longrightarrow \\ \langle \Delta,\ i_g,\ \Sigma,\ \Omega',\ \Theta[a \mapsto \langle \mathbb{D}, \mathbb{D}_\tau,\ \tau,\ e_\square[\texttt{end-atomic}],\ \bot\rangle]\rangle\end{array}}$$

5.3.4 Transactions

The following rules describe the actions to take when a new transaction is started. To do this properly a few things have to be taken into account: whether the transaction is started inside a receive block with or without dependencies and if there is already a transaction running or not.

atomic 1 starts a new transaction. Since there is no transaction running we have to create a new transaction entity and put it in the global map. A new identifier is created (τ `fresh`) and a new transaction is registered in Ω, the global transaction map. The transaction gets a new write point which is the value of the current global write point, i_g, a flag that marks it as `running` and a backup of the entire execution context. The latter is to make sure we can retry the transaction in case of a collision or a failed dependency.

[atomic 1]

$$\frac{\tau\ \texttt{fresh} \qquad \Omega^{'} = \Omega[\tau \mapsto \langle i_g,\ \texttt{running}, \emptyset,\ \emptyset,\ e_\square[\texttt{atomic}\ \mathcal{E}\ \texttt{end-atomic}]\rangle]}{\begin{array}{c}\langle \Delta,\ i_g,\ \Sigma,\ \Omega,\ \Theta[a \mapsto \langle \mathbb{D},\ \emptyset,\ \bot,\ e_\square[\texttt{atomic}\ \mathcal{E}\ \texttt{end-atomic}],\ \mathcal{E}_r\rangle]\rangle \qquad \longrightarrow \\ \langle \Delta,\ i_g,\ \Sigma,\ \Omega^{'}, \Theta[a \mapsto \langle \mathbb{D},\ \emptyset,\ \tau,\ e_\square[\mathcal{E};\ \texttt{end-atomic}],\ \mathcal{E}_r\rangle]\rangle\end{array}}$$

atomic 2 In case that there is already a transaction running on the current thread the atomic block is as simple as just running the expression in $\mathcal{E}$.

[atomic 2]

$$\frac{\tau \neq \bot}{\begin{array}{c}\langle \Delta,\ i_g,\ \Sigma,\ \Omega,\ \Theta[a \mapsto \langle \mathbb{D}, \mathbb{D}_\tau,\ \tau,\ e_\Box[\texttt{atomic}\ \mathcal{E}\ \texttt{end-atomic}],\ \mathcal{E}_r\rangle]\rangle \quad \longrightarrow \\ \langle \Delta,\ i_g,\ \Sigma,\ \Omega, \Theta[a \mapsto \langle \mathbb{D}, \mathbb{D}_\tau,\ \tau,\ e_\Box[\mathcal{E}],\ \mathcal{E}_r\rangle]\rangle\end{array}}$$

atomic 3 When `commit 2` puts a transaction in the `stalled` status it is still possible for the programmer to start a new transaction further down in the expression. In case there is a `stalled` transaction τ present, τ becomes `running` again, instead of creating a new transaction, although the programmer would expect it to be two separate transactions. Notice that the backup of the execution context ($\mathcal{E}_r$) is not overwritten. If the transaction fails the entire transaction should be retried, ergo, jump back to the first retry point of the transaction.

[atomic 3]

$$\frac{\Omega^{'} = \Omega[\tau \mapsto \langle \tau,\ i_\tau,\ \texttt{running},\ \Psi_\tau,\ \mathcal{R},\ \mathcal{E}_c\rangle]}{\begin{array}{c}\langle \Delta,\ i_g,\ \Sigma,\ \Omega[\tau \mapsto \langle \tau,\ i_\tau,\ \texttt{stalled},\ \Psi_\tau,\ \mathcal{R},\ \mathcal{E}_c\rangle], \Theta[a \mapsto \langle \mathbb{D}, \mathbb{D}_\tau,\ \tau,\ e_\Box[\texttt{atomic}\ \mathcal{E}\ \texttt{end-atomic}],\ \mathcal{E}_r\rangle]\rangle \quad \longrightarrow \\ \langle \Delta,\ i_g,\ \Sigma,\ \Omega^{'}, \Theta[a \mapsto \langle \mathbb{D}, \mathbb{D}_\tau,\ \tau,\ e_\Box[\mathcal{E}],\ \mathcal{E}_r\rangle]\rangle\end{array}}$$

5.3.5 Transactional Variables

refset 1 facilitates writing to a transactional variable r. This can only be done inside an atomic block. First r is retrieved from the global heap. If a transactional variable is clear to be written to $\bot$ should be in the entity its owner and the first entry in the history of r should have a write point that is either equal or less than i_τ. Finally, there should not be a value for r present in the in-transaction value store Ψ_τ. If all these requirements are met, the global heap is updated by setting τ as the owner of r, the newly set value v is stored in the in-transaction value store, r is added to the bag of written transactional variables $\mathcal{R}$ and finally the evaluation context is reduced to the newly set value.

[refset 1]

$$\frac{\begin{array}{c}\tau \neq \bot \qquad i_\tau \geq i \qquad \Psi_\tau(r) = \bot \qquad \Delta(r) = \langle (v',\ i) :: l,\ \bot\rangle \\ \Omega' = \Omega[\tau \mapsto \langle \tau,\ i_\tau,\ \texttt{running},\ \Psi_\tau[r \mapsto v],\ \mathcal{R} \cup \{r\},\ \mathcal{E}_c\rangle] \qquad \Delta' = \Delta[r \mapsto \langle [(v',\ i) :: l,\ \tau]\rangle\end{array}}{\begin{array}{c}\langle \Delta,\ i_g,\ \Sigma,\ \Omega[\tau \mapsto \langle \tau,\ i_\tau,\ \texttt{running},\ \Psi_\tau,\ \mathcal{R},\ \mathcal{E}_c\rangle], \Theta[a \mapsto \langle \mathbb{D}, \mathbb{D}_\tau,\ \tau,\ e_\Box[\texttt{set!}\ r\ v],\ \mathcal{E}_r\rangle]\rangle \\ \longrightarrow \quad \langle \Delta',\ i_g,\ \Sigma,\ \Omega',\ \Theta[a \mapsto \langle \mathbb{D}, \mathbb{D}_\tau,\ \tau,\ e_\Box[v],\ \mathcal{E}_r\rangle]\rangle\end{array}}$$

refset 2 handles the case where the current transaction has already written a value to the transactional variable r. The previously written value, v_l is retrieved from the local value store Ψ_τ. At this point all that is left to do is update the new value in the local value store and reduce the reduction context to the newly set value.

[refset 2]

$$\frac{\Psi_\tau(r) = v_l \qquad \Delta(r) = \langle (v',\ i) :: l,\ \tau\rangle \qquad i_\tau \geq i,}{\begin{array}{c}\langle \Delta,\ i_g,\ \Sigma,\ \Omega[\tau \mapsto \langle \tau,\ i_\tau,\ \texttt{running},\ \Psi_\tau,\ \mathcal{R},\ \mathcal{E}_c\rangle], \Theta[a \mapsto \langle \mathbb{D}, \mathbb{D}_\tau,\ \tau,\ e_\Box[\texttt{set!}\ r\ v],\ \mathcal{E}_r\rangle]\rangle \\ \longrightarrow \\ \langle \Delta,\ i_g,\ \Sigma,\ \Omega[\tau \mapsto \langle \tau,\ i_\tau,\ \texttt{running},\ \Psi_\tau[r \mapsto v],\ \mathcal{R},\ \mathcal{E}_c\rangle], \Theta[a \mapsto \langle \mathbb{D}, \mathbb{D}_\tau,\ \tau,\ e_\Box[v],\ \mathcal{E}_r\rangle]\rangle\end{array}}$$

refset-fail 1 In case that a transaction τ tries to write to a transactional variable r which has been written to in the global heap after τ started (i.e., a transaction committed changes to it) τ has to abort in order to maintain a consistent snapshot of the global state. τ is aborted and the current reduction context is replaced with a backup of the one stored inside $\mathcal{E}_c$. The final thing to do is clean the in-box. Inside of τ messages may have been received. To reprocess these in the next try of τ we `reprocess( )` all messages that are currently flagged as `tx-processing` by flagging them as `new` again.

[refset-fail 1]

$$\frac{\Delta(r) = \langle (v',\ i) :: l,\ _\rangle \qquad i_\tau < i \qquad \Omega' = \Omega[\tau \mapsto \langle \tau,\ i_\tau,\ \texttt{aborted},\ \Psi_\tau,\ \mathcal{R}, \mathcal{E}_c\rangle] \qquad \mathbb{I}_a = \{\texttt{reprocess}(m, \{\texttt{tx-processing}\}) \mid m \in \Sigma(a)\}}{\langle \Delta,\ i_g,\ \Sigma,\ \Omega,\ \Theta[a \mapsto \langle \mathbb{D}, \mathbb{D}_\tau,\ \tau,\ e_\Box[\texttt{set!}\ r\ v],\ \mathcal{E}_r\rangle]\rangle \longrightarrow \langle \Delta,\ i_g,\ \Sigma[a \mapsto \mathbb{I}_a],\ \Omega',\ \Theta[a \mapsto \langle \mathbb{D}, \emptyset,\ \bot,\ \mathcal{E}_c,\ \mathcal{E}_r\rangle]\rangle}$$

refset-fail 2 applies when a transaction tries to write to transactional variable r which is owned by another transaction τ'. As a result the transaction can not write to this variable. Subsequently the transaction has to abort.

[refset-fail 2]

$$\frac{\Delta(r) = \langle _,\ \tau'\rangle \qquad \tau \neq \tau' \qquad \mathbb{I}_a = \{\texttt{reprocess}(m, \{\texttt{tx-processing}\}) \mid m \in \Sigma(a)\} \qquad \Omega' = \Omega[\tau \mapsto \langle \tau,\ i_\tau,\ \texttt{aborted},\ \Psi_\tau,\ \mathcal{R}, \mathcal{E}_c\rangle]}{\langle \Delta,\ i_g,\ \Sigma,\ \Omega,\ \Theta[a \mapsto \langle \mathbb{D}, \mathbb{D}_\tau,\ \tau,\ e_\Box[\texttt{set!}\ r\ v],\ \mathcal{E}_r\rangle]\rangle \longrightarrow \langle \Delta,\ i_g,\ \Sigma[a \mapsto \mathbb{I}_a],\ \Omega',\ \Theta[a \mapsto \langle \mathbb{D}, \emptyset,\ \bot,\ \mathcal{E}_c,\ \mathcal{E}_r\rangle]\rangle}$$

refread 1 Reading a transactional variable that has been read before, or been written to before is as simple as reading the associated value from the in-transaction store Ψ_τ.

[refread 1]

$$\frac{\Psi_\tau(r) = v}{\langle \Delta,\ i_g,\ \Sigma,\ \Omega[\tau \mapsto \langle \tau,\ i_\tau,\ \texttt{running},\ \Psi_\tau,\ \mathcal{R},\ \mathcal{E}_c\rangle], \Theta[a \mapsto \langle \mathbb{D}, \mathbb{D}_\tau,\ \tau,\ e_\Box[\texttt{read}\ r],\ \mathcal{E}_r\rangle]\rangle \longrightarrow \langle \Delta,\ i_g,\ \Sigma,\ \Omega,\ \Theta[a \mapsto \langle \mathbb{D}, \mathbb{D}_\tau,\ \tau,\ e_\Box[v],\ \mathcal{E}_r\rangle]\rangle}$$

refread 2 In case there is no in-transaction value for the transactional variable r it is required to fetch it from the global store. Since r might have been written to since our transaction τ started it is needed to get the value it had before our transaction started. Given the history of r and our starting point, `value()` will return the value in that history which was present just before our transaction started. Finally, we store that value in our in-transaction values.

[refread 2]

$$\frac{\Psi_\tau(r) = \bot \qquad v = \texttt{value}(l_v,\ i) \qquad \Delta(r) = \langle l_v, _\rangle \qquad v \neq \bot}{\langle \Delta,\ i_g,\ \Sigma,\ \Omega[\tau \mapsto \langle \tau,\ i_\tau,\ \texttt{running},\ \Psi_\tau,\ \mathcal{R},\ \mathcal{E}_c\rangle], \Theta[a \mapsto \langle \mathbb{D}, \mathbb{D}_\tau,\ \tau,\ e_\Box[\texttt{read}\ r],\ \mathcal{E}_r\rangle]\rangle \longrightarrow \langle \Delta,\ i_g,\ \Sigma,\ \Omega,\ \Theta[a \mapsto \langle \mathbb{D}, \mathbb{D}_\tau,\ \tau,\ e_\Box[v],\ \mathcal{E}_r\rangle]\rangle}$$

refread 3 allows for a transactional variable r to be read outside of a transaction. r is looked up in the global heap and the latest value is returned.

[refread 3]

$$\frac{\Delta(r) = \langle (v,\ i) :: l_h,\ _\rangle}{\langle \Delta,\ i_g,\ \Sigma,\ \Omega, \Theta[a \mapsto \langle \mathbb{D}, \emptyset,\ \bot,\ e_\square[\texttt{read}\ r],\ \mathcal{E}_r\rangle]\rangle \quad \longrightarrow \quad \langle \Delta,\ i_g,\ \Sigma,\ \Omega,\ \Theta[a \mapsto \langle \mathbb{D}, \emptyset,\ \bot,\ e_\square[v],\ \mathcal{E}_r\rangle]\rangle}$$

5.3.6 Committing

commit 1 applies in two cases.

If an actor a has no dependencies built up during the transaction τ (i.e., $\mathbb{D}_\tau = \emptyset$) and the enclosing receive block is a *stable* context τ can safely commit because it has no other transactions to account to. If this is the case τ is in the running status, $\mathbb{D} = \mathbb{D}_\tau = \emptyset$, and $s \in \{\texttt{running},\ \texttt{waiting}\}$. Consequently, τ is changed to the `committed` status, the heap (Δ) is updated with the in-transaction values (Ψ_τ) and the evaluation context is reduced to *nil*.

In case $\mathbb{D}_\tau$ is not empty, τ has received one or more messages during execution which resulted in a dependency on an arbitrary actor. Ergo, τ can only commit when all these dependencies are in the `committed` or `waiting` status. If all dependencies are in the `waiting` status τ can safely commit. This is guaranteed by the $\forall \tau' \in \mathbb{D}_\tau\ : \tau_\diamond(\tau') \vee \tau_\checkmark(\tau')$ predicate. Consequently a cascade apply of this rule will make sure all dependencies can commit as well.

[commit 1]

$$\frac{\begin{array}{c} s \in \{\texttt{running},\ \texttt{waiting}\} \qquad \forall \tau' \in \mathbb{D}_\tau\ : \tau_\diamond(\tau') \vee \tau_\checkmark(\tau') \\ \texttt{merge}(\Delta,\ \Psi_\tau,\ \mathcal{R},\ i_\tau) \qquad \Omega' = \Omega[\tau \mapsto \langle \tau,\ i_\tau,\ \texttt{committed},\ \Psi_\tau,\ \mathcal{R},\ \mathcal{E}_c\rangle], \end{array}}{\begin{array}{c} \langle \Delta,\ i_g,\ \Sigma,\ \Omega[\tau \mapsto \langle \tau,\ i_\tau,\ s,\ \Psi_\tau,\ \mathcal{R},\ \mathcal{E}_c\rangle], \Theta[a \mapsto \langle \emptyset, \mathbb{D}_\tau,\ \tau,\ e_\square[\texttt{end-atomic}],\ \bot\rangle]\rangle \quad \longrightarrow \\ \langle \Delta,\ i_g + 1,\ \Sigma,\ \Omega',\ \Theta[a \mapsto \langle \emptyset,\ \emptyset,\ \bot,\ e_\square[nil],\ \bot\rangle]\rangle \end{array}}$$

commit 2 is applied when a transaction τ is running but has a dependency in $\mathbb{D}_\tau$ that is still `running` and a possible enclosing receive block has no dependencies (i.e., $\mathbb{D} = \emptyset$). To make sure that all transactions commit in group τ has to wait for its dependencies to finish (i.e., go into the `waiting` or `committed` status). To do this it goes into the `waiting` status itself and the reduction context remains `end-atomic`.

[commit 2]

$$\frac{\exists\ \tau' \in \mathbb{D}_\tau\ : \tau_\rightarrow(\tau') \qquad \Omega' = \Omega[\tau \mapsto \langle \tau,\ i_\tau,\ \texttt{waiting},\ \Psi_\tau,\ \mathcal{R},\ \mathcal{E}_c\rangle],}{\begin{array}{c} \langle \Delta,\ i_g,\ \Sigma,\ \Omega[\tau \mapsto \langle \tau,\ i_\tau,\ \texttt{running},\ \Psi_\tau,\ \mathcal{R},\ \mathcal{E}_c\rangle], \Theta[a \mapsto \langle \emptyset, \mathbb{D}_\tau,\ \tau,\ e_\square[\texttt{end-atomic}],\ \bot\rangle]\rangle \\ \longrightarrow \quad \langle \Delta,\ i_g,\ \Sigma,\ \Omega',\ \Theta[a \mapsto \langle \emptyset, \mathbb{D}_\tau,\ \tau,\ e_\square[\texttt{end-atomic}],\ \bot\rangle]\rangle \end{array}}$$

commit 3 When a transaction τ is `running` but there are dependencies in the enclosing receive block (i.e., $\mathbb{D} \neq \emptyset$), τ is not allowed to commit. The values can not be written to the global heap just yet because the enclosing receive is a *depending* context and can still be reverted. As a result it is not possible to commit yet. Consequently, we want every transaction that depends on this transaction to wait before committing as well. Thus, we want to refrain from committing to the global heap until the end of the receive block. At that point the dependencies in $\mathbb{D}$ and

$\mathbb{D}_\tau$ can be safely processed. Thus, the transaction will go to the `stalled` status. Intuitively this means that the transaction is not yet committed, yet not running. The reduction context will be reduced to *nil* in order to continue with the rest of the receive block. The only thing we can check is if there are any failed dependencies in $\mathbb{D}_\tau$ at this point. If so, it would be perfectly safe to restart the current transaction.

[commit 3]

$$\frac{\mathbb{D} \neq \emptyset \qquad \nexists\, \tau \in \mathbb{D}_\tau : \tau_\times(\tau) \qquad \Omega' = \Omega[\tau \mapsto \langle \tau,\ i_\tau,\ \texttt{stalled},\ \Psi_\tau,\ \mathcal{R},\ \mathcal{E}_c\rangle],}{\begin{array}{c}\langle \Delta,\ i_g,\ \Sigma,\ \Omega[\tau \mapsto \langle \tau,\ i_\tau,\ \texttt{running},\ \Psi_\tau,\ \mathcal{R},\ \mathcal{E}_c\rangle], \Theta[a \mapsto \langle \mathbb{D}, \mathbb{D}_\tau,\ \tau,\ e_\square[\texttt{end-atomic}],\ \mathcal{E}_r\rangle]\rangle \\ \longrightarrow \qquad \langle \Delta,\ i_g,\ \Sigma,\ \Omega',\ \Theta[a \mapsto \langle \mathbb{D}, \mathbb{D}_\tau,\ \tau,\ e_\square[nil],\ \mathcal{E}_r\rangle]\rangle\end{array}}$$

commit 4 is the rule that is needed to commit a previously stalled transaction τ. This happens when an atomic block is run inside a receive block that is a *depending* context. τ was not allowed to commit at the evaluation of `end-atomic` and was put into a `stalled` status by `commit 3` and has afterwards been reverted back to the `waiting` status by `end-rcv 2` at the end of the enclosing receive which introduced the dependencies in $\mathbb{D}$. Since τ has to be in the `running` status in order for `commit 3` to apply, checking that τ is in the `waiting` status, together with a the predicate that $\mathbb{D}$ is not empty is enough to ensure that this rule only applies in this particular case. If all the dependencies in $\mathbb{D}_\tau$ are in a `waiting` or `committed` status it is safe for τ to commit as well. We merge with the global heap and change the transaction to the `committed` status and insert `end-rcv` in the evaluation context. The latter allows for `end-rcv 1` to apply. Recall that all dependencies in $\mathbb{D}$ were checked before the transaction was committed. This is impossible to change in the meanwhile so we can safely assume that `end rcv 1` will apply.

[commit 4]

$$\frac{\begin{array}{c}\mathbb{D} \neq \emptyset \qquad \forall \tau' \in \mathbb{D}_\tau\ : \tau_\checkmark(\tau') \vee \tau_\diamond(\tau') \qquad \texttt{merge}(\Delta,\ \Psi_\tau,\ \mathcal{R},\ i_\tau) \\ \Omega' = \Omega[\tau \mapsto \langle \tau,\ i_\tau,\ \texttt{committed},\ \Psi_\tau,\ \mathcal{R},\ \mathcal{E}_c\rangle]\end{array}}{\begin{array}{c}\langle \Delta,\ i_g,\ \Sigma,\ \Omega[\tau \mapsto \langle \tau,\ i_\tau,\ \texttt{waiting},\ \Psi_\tau,\ \mathcal{R},\ \mathcal{E}_c\rangle], \Theta[a \mapsto \langle \mathbb{D}, \mathbb{D}_\tau,\ \tau,\ e_\square[\texttt{end-atomic}],\ \mathcal{E}_r\rangle]\rangle \\ \longrightarrow \qquad \langle \Delta,\ i_g,\ \Sigma,\ \Omega,\ \Theta[a \mapsto \langle \mathbb{D}, \emptyset,\ \bot,\ e_\square[\texttt{end-rcv}],\ \mathcal{E}_r\rangle]\rangle\end{array}}$$

commit-fail 1 When a transaction τ reaches the end of its atomic block and there are transactions in $\mathbb{D}_\tau$ that have aborted in the mean while, τ has to restart as well, regardless if there are dependencies in $\mathbb{D}$ or not. The reduction context is replaced with the evaluation context stored in τ, $\mathcal{E}_c$. The transaction is changed to the `aborted` status and transactional dependencies are cleared. Finally we have to reset the in-box. This means flagging all messages that are currently flagged as `tx-processing` as `new`. τ may hold zero or more transactional variables in $\mathcal{R}$ and thus these have to be released. This is facilitated by `releaserefs`.

[commit-fail 1]

$$\frac{\begin{array}{c}\exists\ \tau' \in \mathbb{D}_\tau\ : \tau_\times(\tau') \qquad \mathbb{I}' = \{\texttt{reprocess}(m, \{\texttt{tx-processing}\}) \mid m \in \Sigma(a)\} \\ \Omega' = \Omega[\tau \mapsto \langle \tau,\ i_\tau,\ \texttt{aborted},\ \Psi_\tau,\ \mathcal{R},\ \mathcal{E}_c\rangle] \qquad \Delta'_g = \texttt{releaserefs}(\Delta,\ \mathcal{R})\end{array}}{\begin{array}{c}\langle \Delta,\ i_g,\ \Sigma[a \mapsto \mathbb{I}],\ \Omega[\tau \mapsto \langle \tau,\ i_\tau,\ s,\ \Psi_\tau,\ \mathcal{R},\ \mathcal{E}_c\rangle], \Theta[a \mapsto \langle \mathbb{D}, \mathbb{D}_\tau,\ \tau,\ e_\square[\texttt{end-atomic}],\ \mathcal{E}_r\rangle]\rangle \\ \longrightarrow \qquad \langle \Delta'_g,\ i_g,\ \Sigma[a \mapsto \mathbb{I}'],\ \Omega',\ \Theta[a \mapsto \langle \mathbb{D}, \emptyset,\ \bot,\ \mathcal{E}_c,\ \mathcal{E}_r\rangle]\rangle\end{array}}$$

6
Implementation

In this chapter we will discuss the implementation strategies chosen for the actor library and STM library. We will discuss how each library works in detail. Next we will discuss which modifications had to be applied to each library in order to combine them and apply the behavior defined in the formal semantics.

6.1 Multi-version Concurrency Control STM

Multi-version Concurrency Control (MVCC) is a concurrency control mechanism that stems from database systems (Bernstein & Goodman, 1981). In database systems there is also a notion of transactions. Two applications can execute queries on the same table that might introduce conflicting changes. For example, one program updating the pay grade of a list of employees and another program reading out the pay grade to calculate their paycheck. If updating a list of records takes a long time it would infeasible to lock the entire table for this transaction (i.e., forcing sequential reads and writes). Bernstein and Goodman (1981) address this issue with MVCC. We will explain MVCC in the context of STM, specifically the implementation that is present in Clojure, and by extension how the meta-circular implementation (Van Cutsem, 2015) works.

In Clojure a transaction is delimited by putting a list of expressions inside an `dosync` block. A transactional variable is created by wrapping any data structure into a `ref`. A transactional variable has four metadata fields. An identifier, a lock, a history and an acquired-by field.

- **Identifier** This is a unique identifier for a transactional variable. Every call to `ref` creates a new globally unique identifier.
- **Lock** contains a lock that transactions can acquire. This is needed to make changes to a `ref` atomically.
- **History** The history of a transaction holds a list of all its previous values and the value of the global write point at the time the transaction that wrote it started. Upon creation of a `ref` with value *val* the history contains a single tuple `{val, global write point}`.

- **Acquired-by** holds a reference to the transaction that last successfully acquired the transactional variable. This transaction may either be running or committed.

A transaction is represented by a data structure that has five metadata fields.

- **read-point** holds a copy of the *global write point* that is taken when the transaction initialized.
- **In transaction Values** is a mapping of *ref*s to values. Each time a transaction writes a value to a *ref* it caches the new value in this map.
- **Written refs** is a set of transactional variable references. Each time a transactional variable is written to a pointer to it is stored in this list. This allows the transaction to keep track of which variables it has written to.
- **Commutes** This holds a mapping of transactional variables to lists of functions. Each entry in the list represents a function that was commuted on this transactional variable. Commuting will be explained in detail below.
- **Ensures** Ensures contains a list of transactional variables that have been ensured. Ensuring a `ref` explicitly means that the transaction can only commit when at the end of the transaction none of the ensured `refs` have been written to by another transaction.
- **Status** is a field that will reflect the status of the transaction. This can either be *running*, *committing*, *retrying* or *committed*.

Clojure provides four fundamental constructs regarding STM.

- **Dosync** takes as a parameter a series of expressions which constitute the transaction.
- **Ref-set** is a function that takes a `ref` and a value and updates the value of the given `ref`.
- **Alter** is syntactic sugar for `ref-set`. Instead of a value, `alter` takes as a parameter a function. This function is then applied to the current value of the `ref` and the result will become the new value of the `ref`.
- **@** is a macro that reads the newest value of a `ref` from global memory (or in-transaction values if present). It is also the only construct relating to STM that can be used outside of a `dosync` block.

Clojure STM keeps a *global write point* for the transaction runtime. The write point is a means to keep track of when a transaction started. Each time a transaction starts it makes a copy of the value of the current global write point by value. When it successfully commits, each transactional variable that it has written a new value to will contain a new tuple at the beginning of its history: `{value, writepoint}`. This allows transactions to determine at what point in time a transactional variable was last written to, relative to their start point.

A transaction starts by initializing in-transaction values, written refs, commutes and ensures to empty. The global write point is copied and the transaction is marked as *running*. Next the body of the *dosync* block is executed. If a transaction was already running on the current thread no new transaction is created. The transaction simply merges with the current one. Hence, if the nested transaction were to fail, the enclosing transaction fails as well.

Committing a transaction is the protocol that determines how changes are committed to the global heap. To make sure that a transaction does not create an inconsistent state in the global memory all `refs` are locked during the commit phase. As a consequence no other transaction can commit to refs that have been written to by this transaction at this point. It can continue computing but it can not read or write any values to the transactional variables. However, the commit phase holds these locks for a short period of time. As the first step in the commit phase the transaction will change its own status to *committing*, so that other transactions are aware of this.

Next, the written and *ensured* `refs` are checked. Each `ref` that had a value written to it may not have changed during the run of the transaction, otherwise one might reason that the newly written values are based on a stale `ref` value. Hence, if a written `ref` has been committed to and subsequently the last entry in its history contains a write point that is bigger than the start point of the transaction the transaction must retry. An *ensured* ref allows for this semantic to be applied to `refs` which have only been read from. In a regular scenario a read `ref` can be committed to in the mean while by another transaction. Since each `ref` has a history of values the runtime will just lookup the value the read `ref` had at the time the transaction started. This way a transaction does not have to abort every time a read-only ref has been committed to during the transaction runtime. However, if one *ensures* a `ref` in a transaction the transaction will only succeed if it has not been written to since the start of the transaction. This is also referred to as a *dummy write* because ensure has the same semantic as reading and writing the value from a ref.

All the *commuted* operations are re-applied when the transaction commits. `commute` is used by calling it with a `ref` and a function as parameters. The function is simply applied to the transactional variable. However, the semantics in a transaction differ from simply applying a function to a `ref` with `alter`.

When two functions are combined using function composition and for those two functions the composition is *commutative*, commute can be used without changing the result as if `alter` were used. For example, `inc`, which increments an integer by one and `dec`, which decrements an integer by one make function composition commutative. Applying `inc` after `dec` or vice versa has no effect on the final result.

Listing 6.1 and Listing 6.2 both show a scenario where two threads each execute a transaction. The first case, using `alter`, will cause one of either transactions to retry and subsequently "Transaction start" will be printed more than twice. If one transaction has already incremented `r` the second transaction will fail if the first one is still running because it will notice that the `ref` has been written to. Hence, it will retry hoping that the `ref` will not be modified during the second attempt.

In the second case both transactions will succeed after the first attempt. The reason is that `commute` functions are executed during the commit phase using the latest values from the global heap whilst locking the `ref` (recall that the commit protocol holds locks for all refs). During the transaction itself commuted values are kept in the in-transaction values. As such, consecutive reads of that `ref` will use the updated value from the in-transaction values. One of both transaction will initiate the commit protocol first. That transaction will hold the locks and update the values to the global heap. As a consequence the commutes of the second thread will be applied to the updated values from the global heap.

Finally while still holding all the locks, the transaction will update the history of each `ref`. Each ref has a new tuple prepended to its history log that holds the new value and the starting point of the transaction. And at last the transaction releases all the locks.

Listing 6.1: Two transactions using `alter`

```
1 (dosync
2   (println "Transaction start")
3   (alter  r inc))
```

```
1 (dosync
2   (println "Transaction start")
3   (alter  r dec))
```

Listing 6.2: Two transactions using `commute`

```
1 (dosync
2   (println "Transaction start")
3   (commute r inc))
```

```
1 (dosync
2   (println "Transaction start")
3   (commute r inc))
```

6.2 Actor library

The actor library that was we wrote for this thesis is a bare minimum library to allow experimentation. It allows for creation of actors, message sending and message payloads.

6.2.1 Syntax

An actor is created by calling the `create-actor` function. It expects a function that defines the initial behavior of the actor, a name and possible initial values for the state of the actor. An example of a simple actor is given in Listing 6.3. The name of the actor is `:counter-actor` (line 3), the function is inlined as second argument and the initial value of the counter is zero (line 11).

The example depicts the syntax of a receive statement (line 5 to 9). A receive block contains one or more receive clauses. This example depicts two receive clauses inside a single receive block. The receive block is executed for a message `:increment` that has one variable as its payload, or a message `:increment` wihtout payload. `delta` in this case specifies how much the counter should be incremented. The receive statement will block until a message has arrived that matches one of the clauses. After that the receive statement is evaluated to the value of the last statement of the matching clause. In this case that is the new counter value (line 7 and 9).

Listing 6.3: Actor creation

```
;; Counter actor
(create-actor
  :counter-actor
  (fn [counter]
    (let [new (receive
                ([:increment]
                 (inc counter))
                ([:increment delta]
                 (+ counter delta)))]
      (recur new))))
  0)
```

Sending a message is done by using the `!` function. The function takes as first argument the name of an actor, secondly the message. The last optional argument is a Clojure map that maps names onto values. Listing 6.4 depicts a simple message send to the counter actor shown in Listing 6.3. The last argument shown is the `delta` parameter for the second receive clause. The name of the value in the map should match the name in the receive clause of the actor. In this case `delta`.

Listing 6.4: Sending a message

```
(! :counter-actor :increment {:delta 10})
```

6.2.2 Details

Actors The implementation of this actor library primarily relies on threads to model actors. Each actor in the system is represented by a future that is running in the Clojure agent pool. This means that the fairness of this implementation is the responsability of the Java virtual machine. Each time an actor is created its name is stored in a global map that is accessible to all other actors. The name of the actor is mapped to its in-box instance. When an actor does not make any recursive calls or calls to other functions the process stops and the actor is simply removed from the global actor map and all its data is garbage collected.

Sending To send a message to any actor the programmer can use the `!` macro. Behind the scenes this macro takes a reference to the receving actor's in-box and inserts a map into the queue. Internallly a message is represented as a map that contains the actual message keyword, the possible empty map of arguments passed along with this message and a globally unique identifier for each sent message. Each message is appended to the back of the in-box of the actor.

Receiving To receive a message the actor can use the `receive` macro. The receive macro expands to a complex statement. First the receive statement will create a list of filters that match any of the messages in the receive block. Two messages are equal if they have the same keyword, e.g., `:increment`, and the arguments they expect match the number of values that the message contains. The in-box is then iterated with this map of filters. The first message in the queue that matches any of the filters is returned from the in-box. The arguments passed to this message are then bound to the free variables in the receive clause. Finally the receive clause is

executed and the entire receive expression is reduced to the value of the last expression in the executed receive body.

6.3 Combination of Both Models

The approach taken to accommodate both libraries was by means of *hooks*. Most rules defined in the formal semantics applied some form of predicates on a current state of an actor in order to decide the course of action. As such, it was possible to insert hooks in the right places in the runtime of both libraries and continue based on the result of the hook handler. We will now discuss where each hook was inserted in both implementations and the defined behavior for each of them. In addition we will discuss the other changes that had to be made to each library.

6.3.1 STM

Pre-commit To manipulate the way a transaction commits it was required to hook into the runtime before the transaction actually initiated the commit protocol (*pre-commit-hook*). Listing 6.5 shows the definition of the `pre-commit-hook` function. Before the transaction commits it is required to check the context of the transaction. The runtime needs to determine if the transaction is running in a stable or depending context and if it has built up dependencies during its execution. If the former is the case (line 13) the transaction can commit without changes in the behavior. However, if the transaction is executed in a depending context and that context has dependencies the transaction has to be stalled (line 19). If the transaction is running in a stable context but has built up dependencies during its execution the transaction has to wait for these dependencies (line 23). This will result in either all dependencies committing or one or more dependencies failing (lines 26 and 27). The hook will then return the course of action the STM runtime should take. Commit, retry or stall the transaction (line 14, 17, 21 and 26). The former case will let the STM runtime continue as usual. If the transaction should retry the runtime issues an exception such that the transaction retries. In the latter case the committing protocol of the transaction is skipped. The transaction immediatly returns without committing to the global heap.

Listing 6.5: The behavior executed before each transaction commits.

```
1  (defn pre-commit-hook
2    [tx-atom]
3    (let [tx-id   (:id @*current-transaction*)
4          status? ((comp deref :status deref) *current-transaction*)
5          tx-deps *tx-dependencies*
6          stable? (empty? *dependencies*)
7
8          safe?   (empty?
9                    (filter #(not (some (set [:COMMITTED :WAITING])
10                                       ((comp list deref :status deref) %)))
11                           *tx-dependencies*))]
12     (cond
13       ;; Regular tx
14       (and (empty? tx-deps) (empty? *dependencies*))
15       :commit
16       ;; Commit 1
17       (and stable? safe? (or (= status? :RUNNING) (= status? :WAITING)))
18       :commit
19       ;; Commit 3
20       (not stable?)
21       (do (reset! (:status @tx-atom) :STALLED)
22           :stalled)
23       (and tx-deps (not safe?))
24       (do (reset! (:status @*current-transaction*) :WAITING)
25           (let [all-committed? (wait-for-deps-tx)]
26             (if all-committed?
27               :commit
28               nil)))
29       :else.
30       nil)))
```

Creating transactions The second change that has to be applied to the default STM runtime is the point of creating a transaction. In the regular STM runtime a transaction is either running or not at the point a new transaction is started. In case of the latter the STM runtime creates a new transaction. In case of the former the runtime simply executes the body of the transaction as part of the currently running transaction. In the modified implementation there is the possibility that the current transaction is in the stalled status. If that is the case the transaction has to be put back into the running status before executing the function. Afterwards the transaction is put in to the stalled state again.

Before retry hook A final minor hook is the `before-retry-hook` which is depicted in Listing 6.6. If a transaction has to retry the in-box has to be cleared of all processed messages. The runtime will iterate over the entire in-box and mark each message that is flagged as `tx-processing` as new again (line 5).

Listing 6.6: The behavior executed before each transaction is retried.

```
1 (defn before-retry-hook
2   []
3   ;; Reprocess all the messages that are marked :tx-processing
4   (when *inbox*
5     (.map *inbox* #(reprocess! % [:tx-processing]))))
```

6.3.2 Actor model

Pre process message hook Listing 6.7 depicts the pre-handle-message-hook function. This function is called each time a message has been read from the in-box of an actor. When an actor reads a message from the in-box the runtime needs to check if there are already dependencies present in the actor. If this is not the case the actor becomes dependent by processing that message (lines 10-12). The runtime will then continue processing the receive body by setting a flag that indicates that that receive block is the block that introduced dependencies (line 11). Finally the dependency in the message is added to the set of dependencies of the actor (line 12). If the actor is already in a depending state the dependency is added to the set of actor dependencies (line 15). The final is case is when the actor is running a transaction. In that case the actor will have to add the dependency to the transaction dependency set (line 7-8).

Listing 6.7: Behavior executed before each receive body is executed

```
1  (defn pre-handle-message-hook
2    [message]
3    (let [msg-dependencies (:deps message)]
4      (cond
5        ;; The actor is running a transaction
6        *current-transaction*
7        (set! *tx-dependencies*
8              (set (into *tx-dependencies* msg-dependencies)))
9        ;; Actor is in a stable state
10       (and (empty? *dependencies*) (not (empty? msg-dependencies)))
11       (do (set! *jump-back* true)
12           (set! *dependencies* (set (into current-deps msg-dependencies))))
13       ;; Actor is in a depending state
14       (not (empty? msg-dependencies))
15       (set! *dependencies* (set (into *dependencies* msg-dependencies))))))
```

Post receive When an actor has finished a receive block the runtime needs to check the dependencies for failure. Listing 6.8 depicts the definition of this behavior. However, this only needs to happen after the receive block that introduced the first dependency, i.e., the receive block that has the *jump-back* flag set to true (line 3).

If this is the case the runtime will wait for all the dependencies to either resolve, i.e., commit, or for one of the dependencies to fail (line 4). According to the result of that function the runtime will either execute behavior specific to a failure or a succes (lines 6-7). In both cases the actor's dependencies are cleared. In case of failure the receive block is simply executed again.

Listing 6.8: Behavior executed after every receive body

```
1  (defn post-receive-hook
2    []
3    (when *jump-back*
4      (let [success? (wait-for-deps)]
5        (if-not success?
6          (end-receive-fail)
7          (do (when (not (nil? *current-transaction*))
8                (reset! (:status @*current-transaction*) :WAITING))
9              (end-receive))))
10     :else
11     :ok))
```

6.4 Caveats

Tail Recursion The current implementation has some limitations. For one, the implementation does not allow proper tailrecursion. However, this is a limitation of the Java virtual machine and by extension Clojure. This means that if an actor calls a another function in tail position the actor will build up stack. However, if the actor wants to do a recursive call the `recur` keyword, provided by Clojure, guarentees that no stack will be built.

Exception Handling The implementation given here does not handle application level exceptions. If any actor throws any unchecked exception the program will be in an inconsistent state. Any actor that is depending on an actor that has thrown an unchecked application level exception will still depend on that actor. However, the failed actor does not do any cleanup or deconstruction before failing so the system might reside in a deadlock.

6.5 Conclusion

This chapter discussed the actor library and the STM library in detail. The proof of concept implementation adheres to all the rules that are defined in the formal semantics given in chapter 5. However, not all semantic properties discussed in chapter 2 are maintained due to limitations of the implementation platform.

7 Evaluation

In this chapter we will discuss a small program that will serve as a real-world evaluation of our library. The program is a small ant colony simulation first written by Rich Hickey, the inventor of Clojure. Next we will discuss the semantic properties of the combination of both models. More specifically, which semantics are still guaranteed and which ones have been violated.

7.1 Ant Simulation

The ant simulation program is a classic program that demonstrates the power of Clojure concurrency models. The program simulates an ant colony in which the ants constantly look for food and return it to their home base. The world the ants live in is represented by a matrix of transactional variables. The ants are modelled by agents and move around the grid looking for food. Figure 7.1 shows a screenshot of the running program.

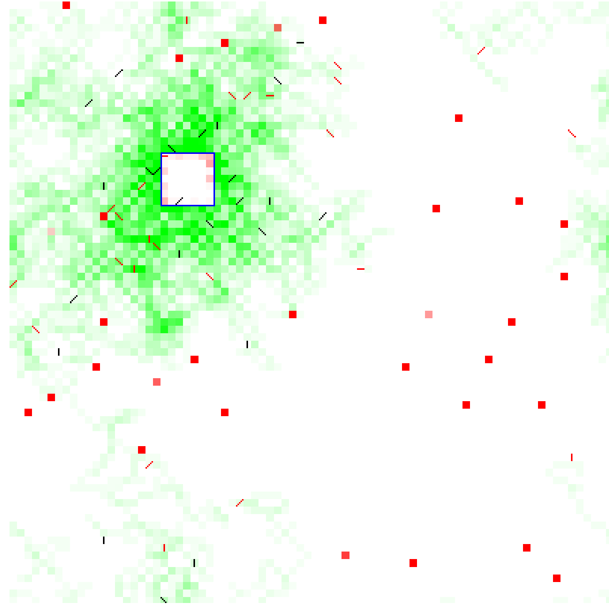
The red cells on the screen represent food. The blue square represents the home base of the ants. The slant lines represent ants and the direction they are walking. If an ant carries food it is drawn as a red line. If the ant is looking for food it is drawn as a black line.

An ant will walk from its home on the grid and forage for food. Each time the ant walks on a cell in the grid it leaves an amount of pheromones on that cell. When more ants walk over the same cell the cell contains more pheromones. As such, these cells become darker green. The pheromones are used to allow other ants to find a shorter path to food. An ant walks in a random direction each time it takes a step. If the ant can take a step in a direction which has pheromones it will take the route with the strongest pheromone smell until it reaches food or its home base. As such, after a while the ants should all be walking on the same straight line between their home base and a food source and thus all walk the shortest path.

7.1.1 Implementation

World The world the ants walk around in is represented by a matrix of transactional variables. Each cell contains the amount of pheromones present and the amount of food that is present. A few cells of the entire grid have food and all the other cells have a food amount of zero. A

Figure 7.1: Screenshot of the ant simulation program



cell is represented by a clojure map that contains a variable number of key-value pairs. If the cell contains food the map will have an entry `:food` with an integer representing the amount of food. If there is an ant present on that cell it will have a field `:ant` with a map containing the direction the ant is going and the amount of food it is carrying as value. Finally, if the cell is part of the home base of the ant it will have a key `:home` with value `true`. An example of the value of a cell is given in Listing 7.2.

Listing 7.2: Sample value of a cell in the ant world.

```
1 {:home true :food 0 :pher 0 :ant {:dir [0 1] :food true}}
```

Ants Ants are represented by agents. The value of the agent is the location of the ant on the grid represented using the Cartesian coordinate system. To make an ant move the `behave` function is sent off to the agent. This function is then applied to the current position of the ant, which is the value of the agent.

The `behave` function starts a transaction immediately. The first thing the function does is send off itself to the agent again. As a result, if the transaction fails the function will not be sent off multiple times and if it succeeds the ant will move again afterwards, i.e., the ant moves as the result of an infinite loop. The behave function makes the ant move one step into a single direction. If the ant is carrying food it can be in 3 possible positions. If the ant is in its home base it will drop the food. If the ant is one step away from its home base it will move into that direction and the previous condition applies. Otherwise the ant will move into the direction that has the most pheromones or otherwise a random direction. If the ant is not carrying food it will either take food if it is located on a cell that has food, or otherwise forage into a random direction that has the most pheromones.

Since the ant's behavior function executes all the operations inside a transaction it is certain that the world is updated atomically. If there are a lot of ants running around which are all executing transactions continuously there will be a lot of collisions between the ants. As such the transaction runtime will have to deal with a lot of transaction failures.

Drawing The final part of the program we will discuss is the gui agent. The system uses the Java Swing library to draw the ants on screen. An agent will iterate the entire grid and render each cell individually. Since the agent does not actually write to cells it can freely iterate the cells and read their current value. To make sure this happens in an infinite loop the agent sends off the redraw function to itself each time he redraws.

7.1.2 Ant Simulation with Actransors

The ant simulation is a good example to demonstrate the flexibility of transactional memory and agents. Since each ant moves concurrently with the other ants and each ant modifies parts of the grid on each move there is a high contention on the cells that are being modified by two or more ants. As a result an ant will often retry its transaction.

In our modified version of the ant system, ants and food are no longer represented by means of agents but as actors. An ant actor will respond to a `:go` message. Upon reception it will execute the behave function which will make the ant move in any direction, take food or drop food. In the original implementation the ant agent sent the `behave` function back to itself inside the transaction to create an infinite loop. This too, can be modelled with the actransor library. The ant actor sends a `:go` message to itself inside the transaction. This message then carries the dependency with it from the currently running transaction. If it retries or commits the actor is aware of this next time it receives the message and thus either removes the messages sent form a retrying transaction or processes the message that came from a committed transaction. As such every message in the in-box of the ant actor is processed accordingly.

Listing 7.3: Creation of an ant actor.

```
1 (create-actor
2      name
3      (fn [loc]
4        (let [new-loc (receive
5                        ([:go]
6                         (behave loc)))]
7          (recur new-loc)))
8      loc)
```

The food actors have some additional work. In the previous example the ants checked the food amount theirselves and took one from the pile. In our example we have extracted that logic into the food actor. In this case an ant will just ask the food actor for the maximum amount that it can get. If the food actor has less food than requested it returns the remainder of its inventory, otherwise it returns what the ant asked for. The important part of this code is that the request for food coming from the ant is sent from within a transaction. As such, if the ant's transaction fails the food actor will not decrement its food stash either. As such the program always remains in a consistent state. And finally, due to the isolation principle no other ant can reduce the food stash while the food actor is handling the request of a single ant.

Listing 7.4: Creation of a food actor.

```
1  (create-actor
2      name
3      (fn [stash place]
4        (let [new-stash
5              (receive ([:get-food delta sender]
6                         (if (> delta stash)
7                           (do (! sender :food {:amount stash})
8                               0)
9                           (do (! sender :food {:amount delta})
10                              (- stash delta))))
11                       ([:read-food sender]
12                         (! sender {:status stash})
13                         stash))]
14          (recur new-stash place)))
15     food place)
```

7.1.3 Conclusion

The ant simulation program is an example of how powerful transactional memory can be. By applying the actransor library to this example we have shown that the library is able to model programs that have been written by using the built in concurrency paradigms of Clojure, namely agents and STM. However, by using the actransor library instead of agents it is possible for ants to have intercommunication. For example it would be possible to implement a collision algorithm between ants. If two ants cross paths both could agree upon a direction to go such that one ant would not have to remain idle while the other ant moves out of the way. Using only agents such additions to the program are not possibly since agents themselves do not support two-way communication.

7.2 Semantics

The current implementation of Actransors respects most of the semantic properties listed for actors in chapter 2. We will now discuss informally each of the properties and reason why they will remain in tact. Note however that the current implementation is a library on top of Clojure, which itself is a library language. This poses certain limitations on the implementation of the language. For example, it would be a very tedious if not impossible task to enfore clojure vars or Java objects to be read-only by certain threads except the thread that created them. As such, not all semantic properties can be guaranteed and the following semantics rely on convention. However, implementing the language from the ground up gives a lot more freedom in design choices to safeguard semantics.

7.2.1 Actors

State Encapsulation The implementation of actransors uses a function definition as the behavior for an actor. The state of an actor is defined as the binding of a value to a formal parameter of that function. As such, by design of Clojure it is impossible to change the state of an actor from outside of its behavior. The state is not stored as a globally accessible variable and as such is immutable from the outside. Moreover, Clojure is a functional language and as such does not support mutability on native Clojure datastructures except vars. However, using the underlying Java object system or vars the programmer is able to create stateful mutable values. Hence, the

guarantee of this property relies on the convention of using immutable Clojure datastructures to hold the state of an actor.

Safe message passing Safe message passing is also guaranteed by the functionality of Clojure. Any created value is immutable and represented as a functional data structure. As such, passing a reference to a value does not damage safe message passing because any other function modifying this reference will obtain a new reference, leaving the original datastructure untouched. Hence, it is safe to pass Clojure datastructures as parameters in a message. As stated above however, it is indeed possible to circumvent this semantic by using stateful Java objects or vars and modifying fields or values of these objects. Thus, the safe message passing semantic is guaranteed if the programmer follows the convention of not modifying shared stateful object instances that it did not create itself.

Atomicity Atomicity is not guaranteed in this implementation. Since this thesis focusses on shared mutable state between actors by the means of software transactional memory it is possible for an actor's execution to depend on the value of such a mutable value. As such, by design this property has been damaged and can not be guaranteed.

Location Transparency and Mobility Location transparency and mobility do not apply to this implementation. The reason is that the implementation does not support distributed actors. Software transactional memory is a concurrency paradigm used in a non-distributed context. Hence, we chose to not model distributed actors explicitly.

Fairness Fairness is guaranteed in this implementation. The reason is that an actor is implemented as a future. This means that the future is executed in the thread pool of the Clojure runtime which in turn is a threadpool in Java. Hence, the responsibility for fairness is passed on to the JVM implementation.

7.2.2 Software Transactional Memory

Serializability and 1-Copy Serializability The two semantic properties discussed above are respected in the current implementation. In effect not much has changed in the runtime of transactional memory. The only change to consider is that transactions can now be in the waiting status instead of committed, running or aborted. The possible problem that could arise is in the barging process of the Clojure STM. A transaction that requires exclusive access to a transactional variable which is owned by another transaction will try to barge that transaction causing it to rerty, if it is not running longer than the barging transaction. As such, if a transaction is in the waiting state it can hold exclusive access to transactional variables and the runtime will try to barge such transactions. A simple modification to the barging protocol that disallows barging of waiting transactions resolves this issue. This is necessary because the point of a waiting transaction is to signal to its dependees that it is sure to commit but is held back by one of its dependencies. Consequently serializability and 1-Copy serializability are preserved.

Regarding liveness in software transactional memory there is a possibility of deadlocks. An actor language such as Erlang can reside in a deadlock by design. Suppose an actor A is waiting for a message x coming from actor B. At that same time B is waiting for a message from A. These actors reside in a deadlock because neither will send the message. Introducing transaction in this context does not resolve this issue. The actors will still block indefinitely.

7.3 Conclusion

This chapter has shown an example program that uses transactions extensively. We have translated this program such that it used the actransors library implemented in this thesis. The program has not lost any expressiveness and has gained potential for enhancements that would otherwise not have been possible using the agent concurrency paradigm.

After that it has been shown informally that all the semantic properties that we have listed in chapter 2 are still preserved under convention. However, implementing actransors from scratch would would allow us to enforce these semantics instead relying on convention.

8
Related Work

This chapter will cover two papers that are relevant to the research presented in this thesis. Each paper will be discussed and points of divergence between our research and that of the paper will be highlighted.

8.1 Transactors

8.1.1 Introduction

Transactors (Field & Varela, 2005) is a computational model built to model distributed actors, but much more fault-tolerant and composable than traditional actors such that it is possible to compose Transactor systems with ease and still maintain fault tolerance at a system level. The composition of elements using this model does not require any additional central coordination nor middleware. For example, two systems written using this model should be able to communicate without any additional glue code and still maintain the provided fault tolerance and distributed state consistency across services. The paper formalizes the model with the τ calculus and proves several liveness and safety aspects of the model.

8.1.2 Transactors

The goal of the paper is to create a new calculus, the τ calculus, and extensions to the actor model as originally described by Agha (1986) that guarantee that in case of failure a Transactor (actor) will still maintain a consistent snapshot of the distributed system's state by means of rolling back; resulting in a failure-resilient distributed actor model. Note that this paper does indeed handle distributed actors, which is in contrast with my proposed solution which does not work in a distributed setting.

Stabilizing and Checkpointing

A Transactor can call `stabilize` to become in a stable state. This is intuitively speaking a promise to not change its state (e.g., by calling `setstate`). An actor can always become stable,

no matter if it has dependencies or not. Any subsequent calls to `setstate` will then be a no-op. The only way for a transactor to exit a stable state is by either checkpointing, or by rolling back. If neither happens the Transactor can not modify its state.

A Transactor can save its state explicitly by calling the `checkpoint` function. If a Transactor does not have any dependencies at that point it can do this. The Transactor will save its current state so that it can be restored in case of failure. If it were to rollback the next time this is the state it will jump back to. An actor can also have dependencies and still checkpoint if all the dependencies have either stabilized or checkpointed, otherwise `checkpoint` is just a no-op. The Transactor can only determine this with the given information in its dependency list. If a dependency has stabilized in the meanwhile the actor can not know this unless it has received a message from an arbitrary actor that carries with it updated information about the state of the actor. As a result, the programmer should write code that sends "ping" messages to update the Transactor status to its dependees (see Figure 8.1). Hence, to make sure a program is resilient and can revert back to a previous state in case of failure in an orderly manner the programmer has to explicitly stabilize and checkpoint. Without explicit state management in this way the system will still go down in its entirety.

Dependencies

The model uses dependencies that piggyback on sent messages to percolate a Transactor's state across the system. A message send will contain all the dependencies of the sending actor. Each time a Transactor receives a message it becomes dependent on all the actors that are in the message's dependency map. If an actor a is depending on an arbitrary actor, x, and receives a message from actor b, who is also depending on x, a will update its dependency information on x by the possibly new values inside the message coming from b. If and only if a then notices that one of its dependencies has rolled back since it last communicated with it, a will roll back as well. Do note that an actor can only notice that a dependency has rolled back when it receives a message from it directly, or receives a message that contains the actors current status in a dependency map. As such, there is an arbitrary delay between the actual failure of an actor and the propagation of this information through the system.

Rollback

As stated before an actor can receive a message which contains dependency information which results in a rollback. If an actor has previously checkpointed that is the state the actor will revert back to. If no such checkpoint exists the actor will simply vanish. Reverting back to the old state allows the actor to receive messages again. If the actor sends a message to any actor it will carry the information with it that it has rolled back and the current version it is now in. Any receiving actor that was depending on the previous state of the actor now has to rollback as well but can only do so when it receives a message that contains the new information. If no such message is ever received the depending actor will not abort and the system will be in an inconsistent state.

Example

To elaborate the above notions an example (Figure 8.1) is taken from the paper (Field & Varela, 2005). It represents a simple bank account actor that will be used to represent a users bank account. The second code listing on the right hand side represents an ATM. It can transfer money from one account to another. Note that any of these three actors can run on their own network service and can thus be distributed. The bank account has a single value as its state:

the balance. The ATM has three values: *savings*, *checking*, and *acks* (line 22). The first two are two bank accounts for single user in between which money can be transferred. The third value is used to `checkpoint` and finish a transaction if a transfer was successful and will be explained in more detail further down.

If the owner of the ATM wants to transfer money he sends a message to *teller* with an amount (line 4). Upon reception *teller* will send a message to the savings account to increase its balance with *delta* (line 5). A message to the checking account is sent as well to decrease its balance (line 6). Finally *teller* will go back to the `ready` state to receive another message (line 3). Note that each message that has been sent to the two accounts carries with it a dependency, effectively making the two account depend on the teller.

When the *bankaccount* actor (savings account or checking account) receives a message from the teller it in- or decrements it current balance (line 4). Next a check is made to make sure that the balance did not go below 0 (line 6). If the check passes the transactor calls stabilize (line 10), promising it will not change its state until it can `checkpoint`. Finally it sends a message back to the sender (*ATM*) notifying it that it has adjusted the balance successfully (line 11). If the check fails the transactor will send a message back telling the ATM that it failed to adjust its balance and roll back to its initial state (lines 7-8). Implicitly the messages contain the new information of the account that it is either at a stable point or that it has rolled back. This updates the information in the ATM account upon reception of that message.

The ATM transactor will receive two *done* messages after it started a transfer. Upon reception of a *done* message it will send a message to *stdout* containing the message attached to the *done* message from the account (line 8). Next it increments the `ack` value in its state, which keeps track of how many *done* message it has already received (line 9). If the count is at 2, which means both accounts have replied, the ATM will stabilize (line 11). Finally the ATM has to make sure that each transactor involved in this scenario receives a *ping* message (line 12-15). This is needed to make sure all parties have up to date state information about each other. First of all it will send a message to both accounts requesting them to *ping* each other. This will make sure that both accounts have up-to-date information about each other. And finally ATM will request both accounts to *ping* back to him. Suppose that one of both account failed and rolled back. The failed account still receives the *ping* request and then sends a *ping* to the other account, which will receive it and notice that the account has rolled back when it calls `checkpoint`, effectively causing a rollback as well. Finally both accounts send a *ping* message to the ATM which will try to `checkpoint` as well, noticing that both accounts have rolled back. As a result the ATM will roll back too and the system is back in its original state.

8.1.3 Conclusion

Transactors does not explictly model software transactional memory in combination with Actors but it does use a notion of transactions and rollback in case of failure at system level (i.e., a failing actor). This paper forms the basis of the contexts that are present in our formal semantics.

The paper introduces three constructs to the actor model, `stabilize`, `setstate` and `rollback`. A transactor can receive messages and send messages. Not unlike our formal semantics these carry dependencies with them. However, using Transactors dependency information is carried only with messages and no actor can read the state of an other actor in real-time which is a logical consequence of the distributed nature of the system. As a result actors can not respond immediately to failure of a dependency. In our formal semantics an actor will read the state of an actor in real-time and act accordingly at the end of each atomic turn or transaction which allows for more fine-grained failure. Using transactors two actors can keep communicating and doing computations to only be notified of failure after an arbitrary long period of time. This

Listing 8.1: Bank account example of Transactors (Field & Varela, 2005)

```
1  let bankaccount = trans
2    declstate <bal> in
3      msgcase
4        adj <delta,atm> =>
5          bal := !bal + delta;
6          if !bal < 0 then
7            send done <'Not enough funds!'> to atm
8            rollback
9          else
10           stabilize;
11           send done<'Balance updated successfully'> to atm
12         fi
13
14     | pingreq <requester> =>
15         send ping<> to requester
16     | ping<> => // may cause rollback
17         checkpoint
18     esac
19   init
20    <0>
21   snart
```

```
1  let teller = trans
2   declstate <inacct, outacct, acks> in
3    msgcase
4     transfer <delta> =>
5       send adj <delta, self> to !inacct;
6       send adj <-delta, self> to !outacct;
7    | done <msg> =>
8        send println<msg> to stdout;
9        acks := !acks + 1;
10       if !acks := 2 then
11        stabilize;
12        send pingreq<!inacct> to !outacct;
13        send pingreq<!outacct> to !inacct;
14        send pingreq<self> to !outacct;
15        send pingreq<self> to !inacct
16       fi
17   | ping<> => // may cause rollback
18       checkpoint
19   esac
20  etats
21 init
22   <savings, checking, 0>
23 snart
```

could cause a system to do a lot of computations that will eventually be discarded. On the other hand our formal semantics do not allow for distributed actors but that would be in contrast with the shared local state that STM models.

A lengthy proof is provided showing that given two global states k and k' of a program which are related by a given execution trace t that might contain system failures, node failure, application-level failures or messages losses, there always exists an alternative equivalent execution trace t' that only contains possible message losses. This is a very important property that allows the programmer to completely disregard the notion of node failure in his program as the model takes care of all these issues.

A final important point of divergence between our formal semantics and Transactors is that by combining STM and Actors we do not want to introduce explicit new constructs to the language. All the defined behavior of our semantics should be implicit and the programmer should be able to be oblivious of the underlying semantic. Using Transactors the programmer has to write explicit code on how and when to checkpoint or rollback. This means that the programmer has the extra burden of doing bookkeeping, as well as programming additional *ping* algorithms to percolate the new state of a transactor explicitly to the system.

8.2 Communicating Memory Transactions

8.2.1 Introduction

The second and highly related paper we wish to discuss is "Communicating Memory Transactions" by Lesani and Palsberg (2011). As we will see below this paper also combines the Actor model together with software transactional memory. The means to do this is again by piggybacking dependencies onto messages and aborting transactions if any dependencies have failed.

8.2.2 Message Passing

The message sending semantic in the paper is not strictly speaking for the Actor model but a more generic approach; message passing using channels. Actors however are a form of message passing. The paper mentions an implementation for Scala as well, which is an actor framework.

The semantic for sending and receiving messages is much alike what we proposed. Sending a message from within a transaction carries the transaction identifier with it. It does not however carry the dependencies of the sender, which is the case in our implementation. Upon reception of the message the receiver is depending on the sender. Note that a receiver can depend on multiple transactions. The semantic does not provide any rules for sending and receiving messages outside of transactions.

8.2.3 Software Transactional Memory

As mentioned, the paper also uses software transactional memory. However, a different approach than the one we are using. The STM model we chose to formalize is the one that is present in Clojure which is based on *Multi Version Concurrency Control* (MVCC) which uses a pessimistic approach but allows for explicit optimism.

In an optimistic approach the transaction contains a local copy of the global heap and will always perform its read and writes to the transactional variables and check if the actions are still valid at the end of the transaction with respect to the global heap. This allows the transactional runtime to apply a simple or complex logic algorithm to determine validity of the transaction during the commit protocol. So if a transaction writes to a transactional variable that has been written to during the execution the transaction will just go on. In the end the transaction can then use any logic to determine if the write is still valid or should be retried and consequently the entire transaction should retry.

A pessimistic approach will determine validity at each read or write operation during the transaction with respect to the global heap, which is what MVCC does. If a transaction tries to write to a transactional variable that has been written to since the transaction started the transaction will abort immediately, assuming that the write would cause faulty semantics. Otherwise the written value is stored in the transaction's in-transaction value store. However, in Clojure there is the `commute` keyword. This keyword allows the programmer to apply an operation to a transactional variable that is *commutative*. The transactions in Lesani and Palsberg (2011), which are based on the transactions described by Koskinen, Parkinson, and Herlihy (2010) use this approach implicitly.

Moverness

As described in Koskinen et al. (2010), moverness is a relation between two operations that defines if either one can be executed before or after the other in order to preserve semantics. We will elaborate by an example taken from the paper how this helps with software transactional memory. Figure 8.2 depicts a scenario where two threads are executing a transaction simultaneously. We will discuss how the scenario will he handled by MVCC and STM as described by Koskinen et al. (2010).

Clojure MVCC For the example we will assume that no other threads are running nor any other transactions. Ergo, no other transactions can intervene with the two in the example. In case of the Clojure semantic the first thread will set the value of x to 3 (line 3, left hand side). If no other thread has written to x since the atomic block started the transaction will store 3 inside its transaction store and continue. At the same time thread 2 will increment the transactional

Listing 8.2: Illustration of moverness

```
// Thread 1
atomic {
  x := 3;
  z++;
}
```

```
// Thread 2
atomic {
  z++;
  y := 4;
}

```

variable z (line 3 right hand side). It does so by incrementing the value and storing the newly calculated value in the thread-local heap. Next, thread 1 will try to increment the z variable as well. However, the transaction of thread 2, which is not yet finished has already written to the z variable. As a result thread 1 encounters a collision and will abort. To be precise, the transaction of thread 1 will first try to *barge* the owning transaction (thread 2). *Barging* a transaction means forcing it to retry and release any transactional variables it holds. If that fails the transaction retries in the hope that the owning transaction will be finished by the time it retries. Formally, there is no serial execution of these transactions that will preserve semantics because the increment operation is considered a read and a write operation and the set of written refs of both transaction intersects. There is no such scenario possible using MVCC.

Coarse Grained Transactions According to Koskinen et al. (2010) both transactions can still continue. The reason for this is that the increment of the variable z is commutative. What this means is that even though two transactions apply ++ during two concurrent transactions, it does not matter who applies ++ first, the end result is the same. This is what moverness is about. Statements that can be swapped on a timeline of execution while still maintaining the same semantic.

For the example both transactions will keep a log of the operations they apply. Thread 1 and 2 will finish their transaction and end up with a log that has to be applied to the global state. The system proposed can handle this scenario better than common STM. It knows that the increment can be considered a single operation and thus both transaction logs can be interleaved in such a way that both transactions preserve their individual semantics and commit order is preserved. An example of such an interleaving is given in Table 8.1. The table shows an interleaving that can not be processed by common STM implementations such as Clojure STM. At the point of * in Table 8.1 the runtime of MVCC will notice that that variable has been written to and that it can not continue the current transaction (Thread 1). However, Coarse grained transactions will notice that this operation is commutative with the operation done previously (** Table 8.1). The runtime can then construct an alternative interleaving that can pass which is shown in Table 8.2. Notice that the operation marked * from Table 8.1 has been moved to the left of of the operation marked **. This is possible due to the fact that both operations are commutative.

Thread 1	$x := 3$			z++*	commit	
Thread 2		z++**	$y := 4$			commit

Table 8.1: Logged interleaving

Listing 8.3: Clojure commute vs alter example

```
1 (dosync
2   (commute r inc))
```

```
1 (dosync
2  (commute r inc))
```

```
1 (dosync
2   (ref-set! r (+ @r 1)))
```

```
1 (dosync
2  (ref-set! r (+ @r 1)))
```

Thread 1	$x := 3$	z++	commit			
Thread 2				z++	$y := 4$	commit

Table 8.2: Corrected interleaving

Clojure `commute` However not modeled in our formal semantics, the implementation of Clojure allows for the programmer to explicitly mark an operation as commutative. Instead of using the regular `set-ref` operation, once can choose to use the `commute` operation. Figure 8.3 depicts the scenario where two transactions increment the same transactional variable. Once by using regular `ref-set!`, which takes a new value for the transactional variable and overwrites the previous value. The commute operation implies that the operation does not care about the value of the transactional value in the mean while. The operation is applied to the ref and the new value is stored inside the thread-local heap. When the transaction commits the commuted functions are applied to the global heap. The first part of the example, using `ref-set!`, will result in the first transaction succeeding after the first attempt and the second transaction retrying as long as the first one lives (or vice versa depending on which transaction is scheduled first). However, using `commute` both transactions succeed within the first try. Note the programmer has to instruct the runtime explicitly that the function application is commutative where as using the transactions defined by (Koskinen et al., 2010) this behavior is implicit and is determined at runtime by constructing an alternative interleaving of the transactions.

8.2.4 Committing Transactions

Again, much like our formal semantics the given semantics provides means to commit groups of transactions. The committing of transactions is key in making the actor model and STM work together. A transaction without dependencies or dependees can commit like a regular STM transaction and such case will not be discussed in detail. A transaction that has dependencies or dependees is a more tricky situation. It is very likely that sending a message to an actor eventually has the effect of a message being sent back. Since each message carries with it dependencies, cyclic dependencies are bound to happen (i.e., the set of dependencies and dependees intersects). On account that (Lesani & Palsberg, 2011) is based on the STM implantation by Koskinen et al.

(2010), committing transactions that are interdependent relies heavily on the transaction logs to allow for a safe commit. We will now elaborate the details of the commit protocol.

Suppose a scenario in which a transaction τ, running on actor A writes to a set of transactional variables S. Inside τ a send is made to actor B, which receives this message inside a transaction τ'. At the point of reception actor B becomes a dependee of τ. In response to receiving the message, actor B sends a message back to actor A who receives it inside τ. Consequently, both actors depend on each others transaction. Ergo, neither can wait for their dependency to resolve or a deadlock would occur.

According to the definition of the commit rule in (Lesani & Palsberg, 2011), these transactions form a cluster. A cluster is defines as follows. A set of transactions (dependencies) form a cluster when all of them have reached the syntactic end of their transaction. Each of these transactions can have unresolved (uncommitted) dependencies. The dependencies form a cluster if and only if these unresolved dependencies are in the cluster itself. All other dependencies outside of the cluster must be in a committed state. The reason that a cluster can contain uncommitted transaction is simple. It is impossible to require the opposite because it would imply deadlocks in case of interdepending transactions.

The second requirement is the moverness of the logs of all transactions. All transaction logs of the members of the cluster must be right movers with respect to already committed transactions. If this were not the case we could not apply the methods in the transaction because they would cause a collision with previously committed transaction.

Simply put, all entries in the logs of the transactions of the cluster have to be valid when applied to the global heap, after other transactions have committed. Recall from the example that using right moverness the entries of the log could be rearranged in such a way that committing was possible while maintaining commit order and semantics. Finally, all the logs of the cluster's transactions need to have a specific ordering such that they can be committed and still respect the semantics of each transaction in isolation. This can be obtained by applying the right moverness to the entire log of the cluster and finding an ordering such that each entry in the log is a right mover with respect to all entries before it in the log. If all these conditions have been met there is a cluster and the transactions can all be committed.

8.2.5 Conclusion

While the paper handles the exact same subject as this thesis there are plenty of differences. For one, the paper uses a different STM implementation than used in this thesis. The STM implementation defined in (Koskinen et al., 2010) allows for a more fine grained control over the commit protocol by means of inspecting transaction logs and rearranging them in such a way that commit order is preserved but no collisions occur. In contrast with MVCC, it allows for a more optimistic approach, while solving most issues with a more complex commit protocol. In our case there is no control over this process since the Clojure implementation uses pessimistic transactions (i.e., transactions fail upon writing a stale ref). Any decision about collisions is taken during the run of the transaction. Writing to a transactional variable during a transaction that has new values since the transaction started means an immediate abort. This has an impact on how to solve collective commits. Since the semantics of MVCC do not allow for transaction logs to be reordered the semantics of the commit protocol are less flexible.

The proposed semantic in the paper is rather dense compared to the one we have introduced in chapter 5. The paper states that it would be straightforward to implement or model sending and receiving outside of transactions. A receive block that is executed as a consequence of a message send in a transaction has a very different semantic than one that has been started from a message coming from a *stable* state. A receive block that does not contain a transaction still

has to wait for the dependency to be in a state which allows for the receive block to end, or abort. Ergo, we consider the expansion a valuable addition to this paper.

9 Conclusion

In this chapter we will revisit our intial problem statement and the contributions made in this dissertation. We will briefly review the formal semantics introduced in chapter 5. Finally we will discuss the future work of this thesis and conclude the dissertation.

9.1 Problem Statement Revisited

This dissertation addresses the issues that arise when programmers want to combine the actor model with software transactional memory. In chapter 1 we have discussed why programmers tend to mix concurrency models. One specific concurrency model is better suited for a particular task at hand than an other concurrency model. Hence, programmers tend to implement differents parts of a larger program using different concurrency models. Every concurrency model has certain semantic properties that are guaranteed by its design. However, combining concurrency models with one another can violate the semantics of either model. Swalens et al. (2014) have shown that not every concurrency model present in Clojure is well integrated with the rest of the models. For the actor model and software transactional memory the problems reside in the fact that transactional memory is designed such that transactional code can execute an arbitrary number of times. Sending and receiving messages in a transaction are considered IO operations because they alter the state of the entire system. STM does not integrate IO operations or side effects in general in transactions by design. As such, combining the actor model with STM without further thought causes programs to behave unreliably and produce faulty results.

To facilitate STM in the actor model one has to modify how the STM runtime handles message sends and how the actor model handles message receiving in order to act accordingly. However, as we have seen, deciding the proper course of action in each scenario is a complex decision tree.

9.2 Contributions

- In chapter 2 we have discussed the details of both the actor model and software transactional memory. In order to investigate the correctness of transactional systems and actor models we have created a unified list of semantic properties based on existing literature. Finally, existing literature does not agree on a single definition for serializability. Consequently we have based ourselves on an existing formal model (Bernstein & Goodman, 1983) and have defined serializability and 1-Copy serializabilty using this model based on the work of Papadimitriou (1979).
- In chapter 3 we motivate the desirability of being able to use transactional memory in combination with the actor model by means of the apples and oranges example. The example has shown that there are use cases where shared memory in the actor model by means of transactional memory can reduce code complexity and readability. We have also pinpointed the problem scenarios of the ad-hoc combination by means of examples.
- Chapter 5 introduced formal semantics that strictly define the behavior of a language that adheres to the semantic properties mentioned in chapter 2. The semantics not only model actor operations, i.e., message sending and receiving, in transactions but also model the operations outside of actors.
- For STM we have used the metacircular implementation by Van Cutsem (2015). However, for the actor model we have implemented a small library from scratch. An implementation as proof of concept of the conglomerate model has been provided under the name Actransors and is discussed in chapter 6. The implementation is provided as a Clojure library that allows programmers to use transactional memory and actors in Clojure.
- In chapter 7 we have revisited the semantic properties listed in chapter 2. We have found that all semantic properties listed in chapter 2 are guaranteed when the programmer follows the convention of not sharing mutable objects between actors. Furthermore we have concluded that if the actransors library would have been implemented as a language from scratch instead of a library these semantics could be guaranteed by design instead of relying on convention.

9.3 Semantics in a nutshell

The proposed solution in this dissertation uses the notion of volatile, stable and depending contexts to attach metadata to sent messages. Any message that is sent will have metadata attached to it identifying the context it was sent from and the current status of the context.

The notion of contexts in messages allows the receiving actor to be aware of a possible failure of the context it came from such as a transaction as it can track the status of the sending context in realtime. After an actor has executed the associated behavior for a received message it can then either block until the sending context notifies its dependents that it will finalize without failure or it can discard all changes made to its state and jump back to the point before the message was received if the sender has notified of a failure. In the latter case all messages that the actor has sent or received become invalid. The failure will percolate through to all the parts of the entire system that were affected by the initial message send by means of dependency propagation. Any time an actor is processing a message that has metadata identifying a volatile context, i.e., possible failure, each message sent from that point on also carries this data with it. As a result, sending messages in combination creates a complex dependency graph – that

is not always acyclic –which allows each actor to be responsible for his own in-box and resolve dependency problems in isolation.

9.4 Future Work

This dissertation was a step in the direction of fully integrating software transactional memory into the actor model. We will now discuss a few aspects of this dissertation that are opportunities for further research.

9.4.1 Validation

The current validation of the formal semantics is based on the ant colony program by Rich Hickey, the inventor of Clojure. It would be desirable to apply the Actransors library to bigger projects to have an overview of the overall impact on readability, complexity and overall verbosity of the code. However, the difficulty in this lies in the fact that Clojure does not support actors natively. As such programmers tend to resort to other concurrency models to implement concurrent programs in Clojure and the corpus of programs of Clojure programs that use actors and other concurrency models is slim.

9.4.2 Formal Semantics

Formal proof The formal semantics provided in chapter 5 are not proven to be correct by means of a formal proof. Proving that the model adheres to the listed semantic properties of chapter 2 would allow us to weed out possible discrepancies that still remain in the model but are unknown at this point in time.

Models The current formal semantics are based on an actor model that is based on actors implemented in Erlang. As explained in chapter 2 actors in this model are a continuously running process that can receive messages at any point during the execution of that process. As such this model allows for nested receives. Given the formal semantics presented in this dissertation it is possible, by using the current formal semantics as a basis, to provide formal semantics for other variations of these models. For example, the traditional actor model by Agha could possibly simplify the semantics drastically. That model does not allow nested receives and has an explicit statement to change an actor's behavior, namely the `become` statement. Consequently, the Agha actor model has a very clear boundary of atomic turns, which are rather vague in Erlang actors due to the fact that they are a continuous process.

9.4.3 Implementation

The implementation of Actransors is a proof of concept. As such it has room for enhancements. However, as it currently stands the library has been implemented using Clojure which is a language that runs on top of the Java Virtual Machine. By implementing the language from scratch or modifying an existing language such that Actransors are an integral part of the compilation or interpretation process would enable us to significantly improve the performance and would let us enforce the language semantics we have listed in chapter 2.

9.5 Conclusion

This dissertation has proven informally and by means of proof of concept that it is in fact possible to combine software transactional memory with the actor model while still maintaining the semantic properties of each model. More specifically, Multiversion Concurrency Control STM with Erlang-style actors. We have studied the correctness criteria for both models and have created a formal semantic that allows programmers to freely combine these models without having to worry about additional safety and liveness issues in their programs.

References

Agha, G. (1986). *Actors: A model of concurrent computation in distributed systems.* Cambridge, MA, USA: MIT Press.

Armstrong, J. (2007). *Programming erlang: Software for a concurrent world.* Pragmatic Bookshelf.

Bernstein, P. A., & Goodman, N. (1981, June). Concurrency control in distributed database systems. *ACM Comput. Surv.*, *13*(2), 185–221. Retrieved from `http://doi.acm.org/10.1145/356842.356846`

Bernstein, P. A., & Goodman, N. (1983, December). Multiversion concurrency control - theory and algorithms. *ACM Trans. Database Syst.*, *8*(4), 465–483. Retrieved from `http://doi.acm.org/10.1145/319996.319998`

De Koster, J. (2014). *Domains: Language abstractions for controlling shared mutable state in actor systems.* Brussels, Belgium: Vrije Universiteit Brussel.

De Koster, J., Marr, S., D'Hondt, T., & Van Cutsem, T. (2013). Tanks: Multiple reader, single writer actors. In *Proceedings of the 2013 workshop on programming based on actors, agents, and decentralized control* (pp. 61–68). New York, NY, USA: ACM. Retrieved from `http://doi.acm.org/10.1145/2541329.2541331`

Fekete, A., Liarokapis, D., O'Neil, E., O'Neil, P., & Shasha, D. (2005, June). Making snapshot isolation serializable. *ACM Trans. Database Syst.*, *30*(2), 492–528. Retrieved from `http://doi.acm.org/10.1145/1071610.1071615`

Field, J., & Varela, C. A. (2005, January). Transactors: A programming model for maintaining globally consistent distributed state in unreliable environments. *SIGPLAN Notices*, *40*(1), 195–208. Retrieved from `http://doi.acm.org/10.1145/1047659.1040322`

Fuggetta, A., Picco, G. P., & Vigna, G. (1998, May). Understanding code mobility. *IEEE Trans. Softw. Eng.*, *24*(5), 342–361. Retrieved from `http://dx.doi.org/10.1109/32.685258`

Haller, P., & Odersky, M. (2009, February). Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, *410*(2-3), 202–220. Retrieved from `http://dx.doi.org/10.1016/j.tcs.2008.09.019`

Herlihy, M., & Moss, J. E. B. (1993, May). Transactional memory: Architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, *21*(2), 289–300. Retrieved from `http://doi.acm.org/10.1145/173682.165164`

Hewitt, C., Bishop, P., & Steiger, R. (1973). A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on artificial intelligence* (pp. 235–245). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. Retrieved from `http://dl.acm.org/citation.cfm?id=1624775.1624804`

Karmani, R. K., Shali, A., & Agha, G. (2009). Actor frameworks for the jvm platform: A comparative analysis. In *Proceedings of the 7th international conference on principles and practice of programming in java* (pp. 11–20). New York, NY, USA: ACM. Retrieved from `http://doi.acm.org/10.1145/1596655.1596658`

Koskinen, E., Parkinson, M., & Herlihy, M. (2010, January). Coarse-grained transactions. *SIGPLAN Not.*, *45*(1), 19–30. Retrieved from `http://doi.acm.org/10.1145/1707801.1706304`

Lamport, L. (1977, March). Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng.*, *3*(2), 125–143. Retrieved from `http://dx.doi.org/10.1109/TSE.1977.229904`

Lange, D. B., Oshima, M., Karjoth, G., & Kosaka, K. (1997). Aglets: Programming mobile agents in java. In *Proceedings of the international conference on worldwide computing*

and its applications (pp. 253–266). London, UK, UK: Springer-Verlag. Retrieved from http://dl.acm.org/citation.cfm?id=645965.674418

Lea, D. (1996). *Concurrent programming in java: Design principles and patterns.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Lee, E. A. (2006, May). The problem with threads. *Computer*, *39*(5), 33–42. Retrieved from http://dx.doi.org/10.1109/MC.2006.180

Lesani, M., & Palsberg, J. (2011, February). Communicating memory transactions. *SIGPLAN Not.*, *46*(8), 157–168. Retrieved from http://doi.acm.org/10.1145/2038037.1941577

Papadimitriou, C. H. (1979, October). The serializability of concurrent database updates. *Journal of the ACM*, *26*(4), 631–653. Retrieved from http://doi.acm.org/10.1145/322154.322158

Shavit, N., & Touitou, D. (1995). Software transactional memory. In *Proceedings of the fourteenth annual acm symposium on principles of distributed computing* (pp. 204–213). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/224964.224987

Smith, B., & Hewitt, C. (1975, 10). *A plasma primer* (Tech. Rep.). Cambridge, Massachusetts: MIT AI Lab.

Swalens, J., Marr, S., De Koster, J., & Van Cutsem, T. (2014). Towards composable concurrency abstractions. In A. F. Donaldson & V. T. Vasconcelos (Eds.), Proceedings 7th Workshop on *programming language approaches to concurrency and communication-centric software,* grenoble, france, 12 april 2014 (Vol. 155, p. 54-60). Open Publishing Association.

Tasharofi, S., Dinges, P., & Johnson, R. E. (2013). Why do scala developers mix the actor model with other concurrency models? In *Proceedings of the 27th european conference on object-oriented programming* (pp. 302–326). Berlin, Heidelberg: Springer-Verlag. Retrieved from http://dx.doi.org/10.1007/978-3-642-39038-8_13

Van Cutsem, T. (2015, May). *stm-in-clojure: Initial release.* Retrieved from http://dx.doi.org/10.5281/zenodo.18165