Unifying STM and Side Effects in Clojure

By Daniel Rune Jensen, Søren Kejser Jensen & Thomas Stig Jacobsen
dpt1010f15, Spring 2015, Aalborg University

09-06-2015

The Faculty of Engineering and Science
Computer Science 10th term
Address: Selma Lagerlöfs Vej 300
9220 Aalborg Øst
Phone no.: 99 40 99 40
Fax no.: 99 40 97 98
Homepage: `http://www.cs.aau.dk`

Project title:
Unifying STM and Side Effects in Clojure

Subject:
Clojure, Side Effects, and Software Transactional Memory

Project periode:
Spring 2015

Group name:
dpt1010f15

Supervisor:
Lone Leth Thomsen

Group members:
Daniel Rune Jensen
Søren Kejser Jensen
Thomas Stig Jacobsen

Copies: 2

Pages: 126

Appendices: 5

Finished: 09-06-2015

Abstract:

In this project constructs were implemented into Clojure for handling side-effects in Software Transactional Memory (STM) transactions and for more explicit transaction control based on transactional data. Clojure's runtime and STM implementation were investigated and explored in a series of experiments that resulted in the implementation of an event handling system for side-effect handling and constructs for explicit data-based transactional control. The implemented constructs were evaluated through use cases. A part of the transactional control constructs were overlapping in terms of functionality with existing Clojure constructs. This part was usability evaluated using two metrics and a subjective discussion of implementations of the Santa Claus problem. The usability evaluation found a decrease both in terms of lines of code and development time. Furthermore the concurrency model of Clojure were found to be more explicit and expressive with the added constructs. The project concludes that the added constructs eases the development of concurrent programs but a larger usability evaluation and a performance evaluation should be done to validate the result.

Preface

The source code developed during this project is available for download at the following URL: `https://github.com/eXeDK/dpt1010f15` and can be found on the CD attached this report. This project is a continuation of work done in our previous semester project, where concurrency in multiple functional programming languages were evaluated.

Prerequisites

Intermediate knowledge of programming in Clojure, as well as basic understanding of Clojure's Software Transactional Memory (STM) implementation is recommended as the problems stems from the existing STM implementation, and the solutions presented in the report have been implemented in Clojure.

Citation

Citation numbers in square brackets are used throughout the report as references to existing work. A bibliography at the end of the report contains a detailed description of these sources and information on how to obtain them. Two styles of citation are used. The first style is when a citation is before a punctuation mark which means the citation is only associated with the current line. The second style is when a citation is after a punctuation mark at the end of a paragraph then it is associated with the entire paragraph.

Formatting of diagrams

Architectural diagrams show classes and types as square boxes, functions and methods as boxes with rounded corners and method names are prefixed with a punctuation mark.

Formatting of source code

Source code examples are presented throughout the report and may have been formatted differently in order to fit page width, compared to the original code. Source code references in this rapport are written with the following highlighting:

- `Namespace, Class or Type`
- `Function or Method`
- `Variable, Binding or Parameter`

Definitions

The following definitions are used in the rapport.

Clojure Clojure version 1.6, since it is the most recent stable release at the time of writing.

Clojure Runtime The Java archive produced by compiling Clojure, it contains a Clojure part accessible by Clojure developers, and a inaccessible Java part containing the scanner, compiler, data structures, etc.

Side-effect A function performs side-effects if it in addition to returning a value, also manipulates state outside of the function scope.

Transaction A STM transaction in the sense of Clojure's STM implementation.

Transactional Code Code that can be safely executed multiple time inside a transaction.

Non-Transactional Code Code that can be safely executed once inside a transaction.

Transaction Control Functionality allowing a developer to manually control a transaction for example to block, abort or terminate a transaction.

Abort A transaction that stop execution, for example due to conflicts with a another transaction and allows the implementation to re-execute.

Terminate A transaction that aborts and is prevented from being re-executed through some means.

Contents

CHAPTER 1

Introduction

The general development in CPUs goes in the direction of adding more computation threads to the processor instead of increasing the clock frequency. One could say that "the free lunch is over", referring to the article by the same name written by Herb Sutter [1] which foresaw this development in 2005 where the free performance gain from higher CPU frequencies halted and instead started adding more computational threads, making the job of the developer harder due to the need for managing multiple threads of execution with non-deterministic interleaving [2].

Functional programming has been on the rise in the recent years, with multiple functional programming languages being ranked increasingly higher on the Tiobe Index [3]. Functional languages simplify parallel programming by defaulting to immutable data structures, ensuring data can be shared between multiple threads without any risks. Most functional programming languages do however provide access to mutable state for use in for example communication between multiple threads [4, 5].

This chapter explains the motivation for this project in Section 1.1 and defines the problem we will try to solve in Section 1.2. This is followed by Section 1.3 that describes how the problem is approached in order to arrive at a solution for the specified problem.

1.1 Motivation

Our 9th semester project named "Performance and Usability Evaluation of Concurrency in Modern Functional Programming Languages" [6] showed through a usability evaluation, that the implementation of the Software Transactional Memory (STM) in Clojure lacks constructs for allowing side-effects such as API or database calls inside transactions as they cannot be rolled back if the transaction retries. Furthermore it showed that Clojure lacks constructs for controlling when transactions abort and blocking of threads from further execution. This is a problem because Clojure then depends on its interoperability with Java to use the monitors found on all Java `Objects` and the constructs from the `java.util.concurrent` package

to control the execution of threads. These constructs nearly all depend on side-effects to operate, making them incompatible with Clojure's implementation of STM and forces the use of exceptions to make transactions abort.

The limitations of Clojure's STM implementation are what is addressed in this project. In our earlier work [6] we identified several different use cases where we find the existing concurrency constructs and the STM implementation in Clojure insufficient:

- Any use of side-effects such as prints, file operations etc. cannot be performed transactionally as they are not supported by the STM implementation
- Interoperability with Java constructs through side-effects requires the use of locks to be synchronised as only changes to the `Ref` type is supported as part of Clojure's STM implementation
- It is not possible to use the return value of a side-effect. Side-effects in STM can be executed using `agent`s in Clojure, but `agent`s execute asynchronously after the transaction has completed.
- Synchronous side-effects are possible by waiting on an `agent` to finish processing using `await`, this leads to unnecessary blocking as it waits for all actions dispatched thus far, not just the one sent from the caller of `await`.
- The lack of synchronisation constructs including transaction control, makes it impossible to block threads until data is ready for processing, forcing the use of `agent`s or Java interoperability for blocking threads.

```
1 (def keys-ref (ref []))
2 (def rows-ref (ref vector-of-rows))
3
4 (dosync
5   (let [row (first (deref rows-ref))
6         next-key (database-insert row)]
7     (alter keys-ref conj next-key)
8     (alter rows-ref rest)))
```

Listing 1.1: Inserting database rows and returning keys

A simple example that shows the problems with combining side-effects with STM can be seen in Listing 1.1. The example performs parallel insertion of rows into a database and stores the returned keys. The transaction starts by inserting a row into a database in Line 6 and then appends the key to a `vector` in Line 7, before removing the inserted row from the `vector` on Line 8.

Exclusive access are acquired through `alter` to ensure both insertion of the row and updating the `vector`s are performed atomically, without other threads interleaving. A problem with calling `alter` in Line 7 and Line 8 is that it can cause the transaction to abort while having executed the database operation which cannot be rolled back by the transaction.

This is not acceptable behaviour because it means a row would be added to the database multiple times depending on when the transaction aborts, resulting in the key would be added multiple times to the `vector`. Locks would remove the problem of multiple executions because locks only allow for one thread inside a critical section ensuring conflicts are not possible. The use of locks would however also introduce the possibility of deadlocks, therefore the use of STM is favourable.

Restructuring the example seen in Listing 1.1 to use `agent`s to perform the side-effect would not be possible, as performing the database operation inside an `agent` would not allow the transaction to return the resulting key since messages to `agent`s are sent after a transaction commits. Creating another transaction for updating the `vector` with the key would allow operations to be performed, but would not ensure that the `vector` of rows and `vector` of keys were updated atomically together.

Furthermore, since `agent`s execute asynchronously it is necessary to wait for the `agent` to finish executing using the function `await` before reading the resulting value, however we can only wait until all functions sent to the `agent` have been executed, meaning that other threads might have overwritten the result of our original computation before we could read it.

1.2 Problem Statement

Clojure contains limited functionality for handling side-effects in STM transactions and depends on its interoperability with Java to control the execution of threads.

The existing Java constructs used to control the execution of threads are incompatible with the STM implementation of Clojure because they depend on side-effects that are not transactional safe, forcing the use of exceptions to make transactions abort before synchronisation is performed.

A solution to this is to allow side-effects to be executed inside transactions in a transactional safe manner and to introduce constructs that give more control over the execution of threads and better means to abort transactions. Such a solution can be used to supplement Clojure existing solution for controlling the execution of threads and forcing transactions to abort.

The following questions needs to be answered to take steps towards these solutions for allowing transactional safe side-effects and transaction control.

- How does the STM implementation of Clojure interact with the rest of the language and implementation?

- How does Clojure handle the use of side-effects in transactions and what are its limitations?
- How is it possible to introduce the use of side-effects in transactions in Clojure's STM implementation?
- How is it possible to introduce the use of transaction control in Clojure's STM implementation?
- How does the introduction of these concepts into Clojure's STM implementation affect the STM implementation in terms of functionality and usability?

1.3 Project Approach

The approach of this project is influenced by our 9^{th} semester project [6] that gave us general knowledge about concurrent programming in functional programming languages, enabling us to start prototyping and experimenting earlier.

The scope of this project is to enable the use of side-effects and transaction control in Clojures STM transactions, therefore we will not focus on the use of existing constructs not part of Clojure's STM implementation such as `agent`s and external libraries `core.async`.

We will start by looking at the Clojure runtime and the STM implementation to gain an understanding of what is possible in the existing structure and how the STM implementation interacts with the rest of the language. We will take advantage of the knowledge gained from our earlier work and perform experimental implementations. The idea with these experiments is to get a better understanding of Clojure's STM implementation. This understanding of the STM implementation will enable us to develop solutions to solve the presented problems, and how side-effects and transaction control can be supported by the current implementation. Based on the analysis of Clojure's STM implementation, we will compare our experimental implementation with existing solutions presented in the literature, and develop suitable constructs for handling side-effects and perform transaction control. We will then implement these constructs into Clojure's STM implementation and evaluate the added constructs in terms functionality, usability and influence of the constructs on Clojure's overall model of concurrency.

To summarise, the following items have been chosen as our approach to reach the two solutions.

- Analyse Clojure's runtime and STM implementation
- Experiment with possible solutions for side-effects and transaction control

- Compare the experiments to the literature about STM, side-effects and transaction control
- Implement constructs for the use of side-effects in transactions
- Implement constructs for transaction control
- Evaluate the implemented constructs in terms functionality and usability
- Compare the concurrency model of the existing STM implementation of Clojure to the model of the same implementation with these added constructs

We argue that this approach increases the chances that the solution is more fitting for Clojure by taking a starting point in the STM implementation. This means that we will not rely on existing solutions made for other platforms in the literature before we start these experiments. Because of this some of the experiments may overlap with existing solutions found in the literature.

The alterations made to the Clojure runtime will be made as non-intrusive as possible and with efficient data structures and algorithms in mind, therefore a performance evaluation of the STM implementation before and after the additions of the constructs will not be performed.

The report is structured as follows, Chapter 2 analyses, explains and explores the relevant parts of Clojure and its STM implementation. Chapter 3 documents the complete solution constructed based on the analysis and exploratory experiments. The implementation will be evaluated in Chapter 4, ending the report with a discussion in Chapter 5 and conclusion in Chapter 6 which also contains ideas for future work on the subject.

CHAPTER 2

Clojure

This chapter will provide the needed theory to read the rest of the report. We start by giving an in-depth description of Clojure's capabilities of Clojure's macros in Section 2.1. Then an overview of Clojure's runtime in Section 2.2 is provided. Last we given an overview of Clojure's STM implementation in Section 2.3, followed by an in-depth description of each of the components in Clojure's STM implementation in Section 2.4.

2.1 Clojure Macros

Clojure inherent a powerful macro system from it being a Lisp variant where data and code both are symbolic expressions. This allows the same syntax to be used for both functions and macros even though there are semantic differences between them. These differences will be described in this section. [7]

Function: defined using `fn`, `defn` or `defn-`, accepts values as input, returns values as output, and is executed at run time.

Macro: defined using `defmacro`, accepts code as input, returns code as output, and is executed at compile time.

Because macros are expanded at compile time they allow Clojure to be expanded with additional constructs without changing the runtime. Namespace lookup for symbols is automatic inside macros using Clojure's syntax quoting. This syntax quoting is an extended version of Lisp's traditional quoting that allows for additional reader macros. [8]

Similar functionality can be constructed using functions but the code that is used as input must be quoted by the developer. If no quoting was done then the code would be evaluated and the return value would be passed to the function.

Returning code from a function after it has been transformed is also problematic. A lambda function could be returned, which the developer then must execute manually. Alternatively the code could be executed by

the function instead of being returned, the code would then execute in the scope of the function. Some overhead is added by each code transformation because functions are executed at runtime. Macros, on the other hand, are expanded at compile time which removes the runtime cost entirely.

Clojure's macros provide two additional variables. The first variable is called `&env` and provides a `map` with all lexically scoped bindings. The second variable is called `&form` and provides the original form of the macro before it was expanded. Clojure does not provide any capabilities for capturing the current lexical scope on runtime which means that only non-local `vars` can be resolved by a function.

Function: provides dynamic lookup of non-local `var`s as they are stored by the `Namespace` class in the runtime.

Macro: automatically captures any lexically scoped bindings created as a parameter, through a `with-local-vars` or a `let` binding, and provides dynamic lookup of non-local `var`s.

In summary, macros provides access to all symbols in the scope where they are expanded, and allows new syntax to be created without changing the Clojure runtime. Macros however cannot be used to manipulate data provided at runtime, such as a text file or data from a database, as they are expanded before such data is provided to the program. Data manipulation is done by functions, which are executed at the programs runtime, and provides an interface unsuitable to manipulating code due to the additional quoting and lack of accesses to locally scoped variables.

2.2 The Clojure Runtime Overview

The source code of the Clojure runtime is divided into two parts, one part is written in Java the other is written in Clojure, each part is compiled in its own phase of the compilation of the runtime. The Java part is compiled first and contains code for the core of Clojure, for example reading Clojure source code and translating it into Java bytecode which is able to be run on the Java Virtual Machine (JVM). The second part of the compilation phase is to compile the Clojure part of the runtime with the now compiled Java part of the runtime. The Clojure part of the runtime contains all of the function implementations of the language, these serve as Clojure's standard library and provide an interface for the underlying Java parts of the runtime. Finally, both parts are compiled into a single Java Archive (JAR) file that is the Clojure runtime.

An overview of the runtime and the main parts of it can be seen in Figure 2.1. Here the two parts of the runtime are illustrated by the two larger blocks with their content shown inside.

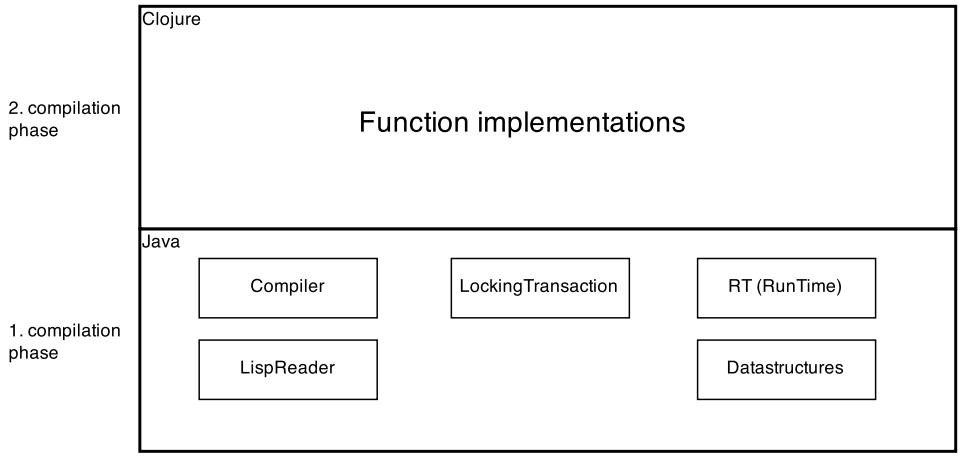

Figure 2.1: Overview of the Clojure runtime structure

2.3 Software Transactional Memory

The implementation of Clojure's STM ensures that operations on transactional data can be safely interleaved by providing guarantees of atomicity, consistency and isolation. Atomicity means that all or none of the operations in a transaction are executed. Consistency means that the transaction has the same view of the transactional data through the entire transaction. Isolation means that operations performed by a transaction is not visible to other transactions before it commits.

The STM implementation in Clojure is based on Multi Version Concurrency Control (MVCC) where each transaction sees its own version of the transactional variables used. MVCC is used by major databases and has been an active research area in more than 30 years and is the basis for the STM implementation found in Clojure. The use of transactions in databases and the notion of snapshot isolation is carried over from the world of databases into the world of concurrency control in programming languages with the use of MVCC. [9]

In Figure 2.2 an example of a MVCC system with multiple concurrently executing transactions is seen. The transaction T_1 is started at time t_1 and will therefore have precedence compared to both T_2 and T_3 if they all need a write lock to the same transactional variable. The three transactions T_1, T_2 and T_3 will all start with the same snapshot since no transaction will commit between t_1 and t_2 in this example. If T_1, T_2 and T_3 need to modify the same transactional variable they will need to be executed in serial whereas T_4 is able to run concurrently with T_1, T_2 and T_3 if either of them abort, given that it does not modify a transactional variable changed in any of T_1, T_2 or T_3. The notion of transactions and serialisability is also found in the domain of databases where it originates [9].

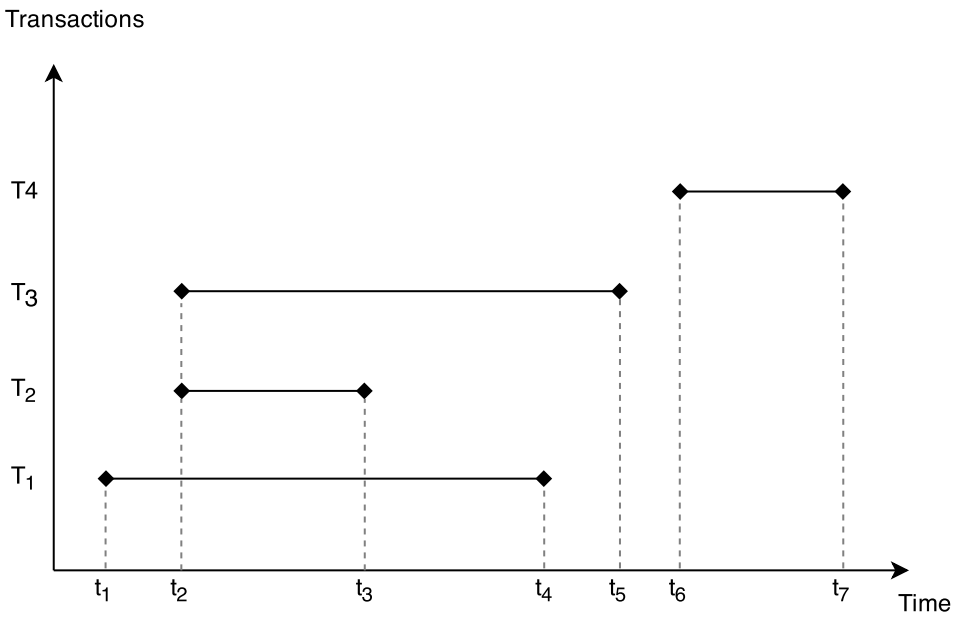

Figure 2.2: An example of MVCC with multiple concurrent transactions

All functionality related to STM is located in the `LockingTransaction` class found in the Java part of the runtime. It is responsible for holding transaction specific values of transactional variables which are instances of the class `Ref`. It is the responsibility of the `LockingTransaction` class to ensure serialisability using its implementation of MVCC. The MVCC implementation found in Clojure uses timestamps on transactions to determine which transaction is allowed to commit in case of a conflict between transactions.

A `Ref` contains information about the version of the value it holds for other transactions to know if they need to abort in order to maintain serialisability. This check of versions happens whenever a `Ref` is read or written for the first time in a transaction and on commit time of each transaction. When a transaction commits it will ensure that it holds the appropriate locks for writing its local transactional values to the global version of the `Ref`s.

The interface for Clojure's STM implementation is provided by six different macros and function, shown in Listing 2.1. To start a new transaction in Clojure the `dosync` macro is used. `dosync` takes an arbitrary number of expressions as input and executes all the expressions as one atomic transaction. Transactions should be short in execution time and the operations must be side-effect free because they might abort and re-execute [9]. This results in the side-effect being executed multiple times. Clojure's STM implementation provides the `io!` macro that can be added to a function to

```
(dosync & body)
(io! & body)

(deref ref)
(ensure ^clojure.lang.Ref ref)

(alter ^clojure.lang.Ref ref fun & args)
(ref-set ^clojure.lang.Ref ref val)
(commute ^clojure.lang.Ref ref fun & args)
```

Listing 2.1: Signatures for the Clojure STM functions

make it thrown an exception if the function is executed inside a transaction, this is to indicate that this part of the code is not transactional safe.

Since `Ref`s are transactional they are only allowed to be modified inside a transaction, otherwise the operation will cause an exception to be thrown. Based on the function executed on a `Ref` inside the transaction, the transaction will automatically acquire the correct type of lock needed. Clojure provides the following four functions for manipulating `Ref`s inside a transaction.

ensure: protects the `Ref` from being changed by another transaction, but leaves the current value intact.

alter: executes a function with the current value of the `Ref` as it's first argument, and assigns the `Ref` to the return value.

commute: equivalent to `alter` but expects the function to be commutative, and allows another transaction to change the same `Ref` using `commute`.

ref-set: assigns a `Ref` to a specific value, and has the same abort semantics as `alter` due to shared implementation.

Depending on how these function are interleaved, some transactions might need to abort due to conflicts with other transactions. The semantics for when and why a function forces a transaction to abort can be seen in Table 2.1. The tables show T_1 on the left and T_2 on the top, each field is annotated with nothing if the functions executed by both can be executed without forcing either to abort, an E if T_2 to forced to abort when it executes it function, and an C if T_2 will be forced to abort when it tries to commit.

T_1 \ T_2	Deref	Ensure	Commute	Alter	Ref-Set
Deref					
Ensure			C	E	E
Commute					
Alter		E	C	E	E
Ref-Set		E	C	E	E

Table 2.1: Overview of when transactions abort based on when the functions are executed, the function on the row (T_1) are executed before the function on the column (T_2), (E) is an abort when the function is executed and (C) when the transaction of the last function tries to commit

2.4 LockingTransaction and Ref

In this section the details of the `LockingTransaction` class and the `Ref` class, mentioned in Section 2.3, will be described to give a better understanding of the STM implementation found in Clojure. The architecture tecture of Clojure's STM implementation, can be seen in Figure 2.3.

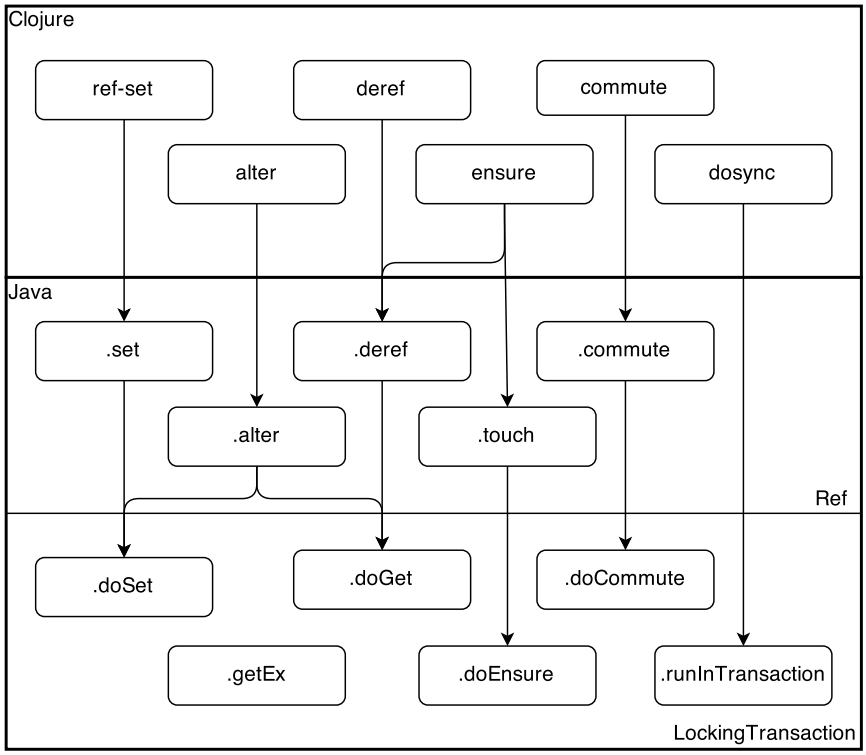

Figure 2.3: The architecture of the STM implementation in Clojure

2.4.1 Metadata

Both `LockingTransaction` and `Ref` contains meta-data to help the STM implementation in handling transactional values, transaction precedence etc. An instance of `LockingTransaction` corresponds to a transaction, therefore the meta-data belonging to a transaction is set on the transactions instance of `LockingTransaction`. A transaction contains the following meta-data

`Info info` Instance of the `Info` type which contains the status of the transaction, the starting point of a transaction represented as a globally unique number and a Java `CountDownLatch`

`long startPoint` A globally unique number to represent at what point in time the transaction started. A higher value for `startPoint` indicates a newer transaction.

`long startTime` A time stamp created by `System.nanoTime`, indicating at what time the transaction was started

`long readPoint` A globally unique number to represent at what point in time the current transaction started. It is equal to `startPoint` if the transaction have not aborted at least once

`ArrayList<Agent.Action> actions` The actions which will be dispatched when the transaction has committed. An action is a function dispatched to an `agent`.

`HashMap<Ref, Object> vals` The transactional values for all `Ref`s in the transaction

`HashSet<Ref> sets` The set of `Ref` instances that `ref-set` or `alter` have been called on

`TreeMap<Ref, ArrayList<CFn» commutes` Commutative operations for `Ref`s in the transaction added by using `commute`

`HashSet<Ref> ensures` The set of `Ref` instances that `ensure` have been called on to make sure that the value of the `Ref` instances will not change as long as the transaction is running

The `Info` type contains meta-data for ordering transactions. This is used as a reference in `Ref` as well as for comparing the age of transactions. `Info` contains the following meta-data:

`AtomicInteger status` Contains the numerical representation of the following states: `RUNNING`, `COMMITTING`, `RETRY`, `KILLED`, `COMMITTED`

`long startPoint` Contains the globally unique numerical identifier of transactions, the higher the `startPoint` the newer the transaction, it is equal to `startPoint` on `LockingTransaction`

`CountDownLatch latch` The `CountDownLatch` is used for allowing transactions to block while waiting for another transaction to execute, thereby unlocking the write lock for a `Ref`

`Ref`s also contains meta-data to handle transactional values and to enable locking. The `tvals` meta-data on a `Ref` is a doubly linked list containing the history of the `Ref`, the maximum size of the linked list is defined by `maxHistory`, hereafter the oldest entry in the linked list is replaced by a new entry, maintaining the size of the linked list to `maxHistory`. The type `TVal` represents an entry in a doubly linked list containing a transactional value and its committed point in time.The meta-data in `Ref` is described below:

`TVal tvals` Contains a doubly linked list of historic values for the `Ref`. `tvals` points to the newest transactional value

`AtomicInteger faults` Contains the number of read faults happened to the `Ref`. A fault happens if no value of a `Ref` is valid for a given transaction. For example is no value for the `Ref` exists in the transaction's snapshot.

`ReentrantReadWriteLock lock` Contains a lock to be used requiring a write lock on a `Ref`

`LockingTransaction.Info tinfo` Contains a reference to the transactional info of the transaction currently holding the lock for the `Ref`

`long id` : Contains a globally unique numerical identifier of the `Ref`, this is used comparing `Ref`s

`int minHistory` Contains the minimum number of elements in `tvals`

`int maxHistory` Contains the maximum number of elements in `tvals`

2.4.2 dosync

The `dosync` macro is the transactional block in Clojure. All code inside a `dosync` block is executed together as a transaction.

The `dosync` method, seen in Listing 2.2 on Line 1, takes a list of expressions as argument and calls the macro `sync` with the same expressions as argument. The `sync` macro then calls the method `runInTransaction` on the `LockingTransaction` class with the expressions it was given as the argument `body` inside a function on Line 6. The `runInTransaction` method creates

```
1 (defmacro dosync [& exprs]
2   `(sync nil ~@exprs))
3
4 (defmacro sync [flags-ignored-for-now & body]
5   `(. clojure.lang.LockingTransaction
6     (runInTransaction (fn [] ~@body))))
```

Listing 2.2: Signatures for the Clojure `dosync` and `sync`

a new transaction by creating a new instance of `LockingTransaction` and starts executing the expressions in the transaction by calling the `run` method with the function containing all the expressions.

```
1  for(int i = 0; ! done && i < RETRY_LIMIT; i++) {
2    try {
3      ...
4    } catch(RetryEx ex) {
5      // Ignore this so we retry rather than fall out
6    } finally {
7      ...
8    }
9  }
10 if( ! done)
11   throw Util.runtimeException("Transaction failed after reaching retry limit");
```

Listing 2.3: The structure of the `run` method in `LockingTransaction`

The `run` method on `LockingTransaction` is used for the execution of transactions and interacts with almost all other parts of Clojure's STM implementation. The `run` method contains a large `for`-loop with a `try` block inside which takes care of the retry functionality, this structure can be seen in Listing 2.3, and the flow between the major parts of `run` can be seen in Figure 2.4. Figure 2.4 will be revisited in Chapter 3 with the added constructs. In Line 1 the variable `done` is checked if true and the variable `i` is checked if below the maximum number of retries allowed for a transaction. `done` is a boolean indicating whether or not the transaction is done, the variable `i` contains the number of transaction retries. The `try` block started in Line 2 is used for the execution of the expressions inside a transaction. If a transaction encounters an error or some state which requires a retry an instance of the `RetryEx` error is thrown, this error is caught in Line 4.

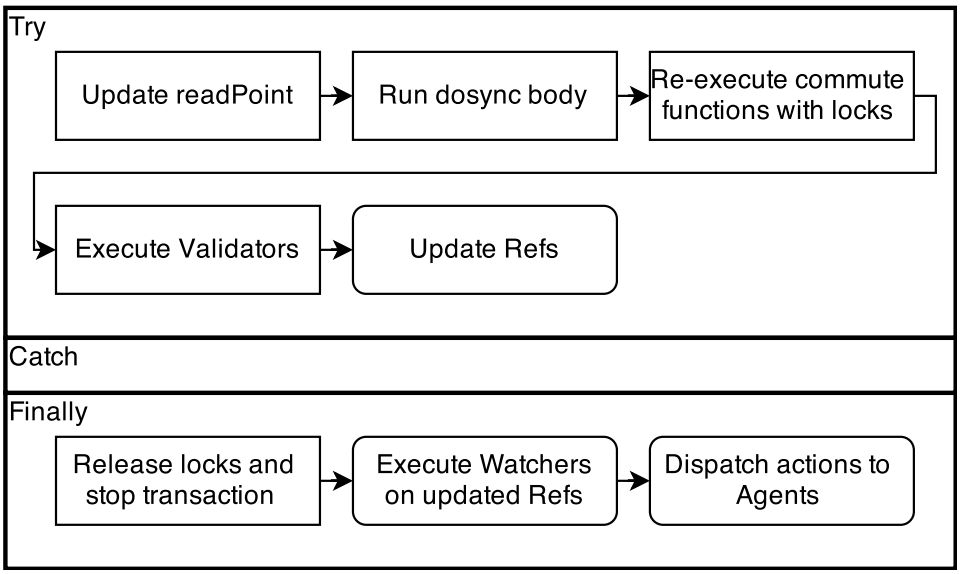

Figure 2.4: The major parts of the `run` method, squares show parts that can be re-executed, while rounded squares cannot

The `finally` block seen in Line 6 takes care of unlocking any locks acquired while executing the transaction and if the transaction was executed successfully `Watchers` are notified and `Agent` actions are dispatched. If the `for`-loop reaches the limit of retries allowed without executing successfully a `RuntimeException` is thrown in Line 11. The code inside the `try`-block in Listing 2.3 on Line 3 is replaced with "..." to highlight and explain some of the code in their own Listings in the following.

```
1 getReadPoint();
2 if (i == 0) {
3   startPoint = readPoint;
4   startTime = System.nanoTime();
5 }
6 info = new Info(RUNNING, startPoint);
7
8 ret = fn.call();
```

Listing 2.4: Starting time and point is assigned in the `run` method

In Listing 2.4 the transaction metadata about starting time and point is assigned, this can be seen in Lines 1 to 6. In Line 8 the transaction is executed and the return value from the last expression is assigned to `ret`. When executing Line 8 the underlying code can throw a `RetryEx` which will result in a retry of the transaction. The next step for the execution

of the transaction is to commit the transaction, making the changes of the transactional values global.

First commutative operations are handled by looping through the Java `HashMap` `commutes` and ensures that no other transaction has any exclusive lock on the `Ref` after the transaction committing was started. If `ensure` was called after the transaction was started the transaction has to be restarted since calling `ensure` on a `Ref` guarantees that the `Ref` will not change as long as the transaction is still running. If another transaction T_2 has started and taken a lock on the `Ref` which T_1 is trying to commit, T_1 will try to barge in on T_2.

Next the write locks needed for modification of the `Ref` instances which have been called `ref-set` or `alter` on are acquired if possible or the transaction aborts. Before committing the transaction values to the global state, assigned `Validator`s for `Ref`s are run. When committing the transaction all the transactional values in the transaction are made global by altering the metadata of the `Ref`s both in terms of the value and the point of committing the value. After the transaction has committed the `done` variable is set to `true` and the status of the transaction is set to `COMMITTED`.

At this point all that is left is to clear up after the transaction in the `finally` block also seen in the structure of the `run` method earlier. In the finally block all locks are unlocked in order for other transactions to acquire the locks needed for committing themselves. If executing the `finally` block after the transaction committed successfully then the `Watcher`s are notified of the update to `Ref` being watched. The actions sent to `Agent`s inside the transaction will also be dispatched, this first happens at commit time to avoid actions to be dispatched multiple times by transactions retrying.

2.4.3 deref

The function `deref` takes one argument as seen in Listing 2.1 in Line 4. The argument can be either a `Ref`, `Var`, `Atom`, or `Agent`, in this section we will describe how `deref` operates the function is applied to a `Ref`, the implementation for the other data types are not described as these are not transactional value and therefore outside the scope of the project. If `deref` is called within a transaction it returns the in-transactional value of the `Ref`, otherwise it returns the most recently committed value. When `deref` is called it checks if the given argument is of type `Ref` before calling the Java method `deref` on the `Ref` instance.

```
1  public Object deref(){
2    LockingTransaction t = LockingTransaction.getRunning();
3    if(t == null)
4      return currentVal();
5    return t.doGet(this);
6  }
```

Listing 2.5: The `deref` Java method in the `Ref` class

The method `deref` is found in the `Ref` instance and can be seen in Listing 2.5. It tries to get the currently running `LockingTransaction` instance by calling `getRunning` on Line 2. If there is no running transaction then it calls `currentVal` in Line 4 which returns the latest committed value of the `Ref` instance because it was not called from a transaction and therefore has no transactional values. But if it was called from a transaction then it calls `doGet` through the `LockingTransaction` instance with the `Ref` instance as argument in Line 5. In case of `ensure` the method `deref` will always be called from a transaction which means it will always call `doGet`.

```
1  Object doGet(Ref ref){
2    if ( ! info.running())
3      throw retryex;
4    if (vals.containsKey(ref))
5      return vals.get(ref);
6    try {
7      ref.lock.readLock().lock();
8      if (ref.tvals == null)
9        throw new IllegalStateException(ref.toString() + " is unbound.");
10     Ref.TVal ver = ref.tvals;
11     do {
12       if(ver.point <= readPoint)
13         return ver.val;
14     } while ((ver = ver.prior) != ref.tvals);
15   } finally {
16     ref.lock.readLock().unlock();
17   }
18   ref.faults.incrementAndGet();
19   throw retryex;
20 }
```

Listing 2.6: The `doGet` Java method in the `LockingTransaction` class

The method `doGet` can be seen in Listing 2.6. The first thing the method does is to check if the transaction is in a running state in Line 2. If the `Ref` instance is in the set of in-transactional values, `vals`, on Line 4 then this value is returned. If this is not the case then the method starts a `try` block

and tries to get the read lock of the `Ref` instance in Line 7. On line 8 the method checks if the `Ref` instance itself contains a value, before it goes through all values in the `Ref`'s internal history in a loop on Line 11, where it will return the first committed value contained in the `Ref` instance that was earlier later or at the same time as the current transaction was started. The `finally` block on Line 15, ensures that the read lock for the `Ref` is removed from the it. This means that when a value is found and returned in Line 13 a committed value for the `Ref` instance has been returned by `deref`. If no value was returned a fault counter will be incremented in Line 18 and the transaction will abort. Faults are used by `LockingTransaction` to determine when a `Ref` should increase the size of its history. Increasing the history for a `Ref`, increases the window where snapshot isolation can be provided, as transactions started before the oldest committed value will be forced to abort due to them breaking consistency when `deref` is called for the `Ref`.

2.4.4 alter and ref-set

The two functions `alter` and `ref-set` will be described together because they are both implemented in terms of `doSet` and have the same abort semantics as seen in Table 2.1.

The `alter` function takes three arguments as seen in Listing 2.1 on Line 7 where the signature for the function is shown. The first argument is the `ref` argument, the second argument is the function which is executed on the value of the `Ref` with the arguments specified.

```
1 public Object alter(IFn fn, ISeq args) {
2   LockingTransaction t = LockingTransaction.getEx();
3   return t.doSet(this, fn.applyTo(RT.cons(t.doGet(this), args)));
4 }
5 public Object set(Object val){
6   return LockingTransaction.getEx().doSet(this, val);
7 }
```

Listing 2.7: The `alter` and `set` Java method in the `Ref` class

The `alter` function invokes the `alter` method on the instance of `Ref` passed as the argument `ref`. The source code for the `alter` method in the `Ref` class can be seen in Listing 2.7. The method takes two arguments, an `IFn` which is an interface representing a Clojure function in the Java part of the runtime and an `ISeq` which contains the arguments for the `IFn`. The instance of the currently running transaction is obtained by calling `getEx` on `LockingTransaction` on Line 2. Hereafter `doSet` is called with the `Ref` and the return value of `fn` as arguments. The function `fn` is called with the

current value of the `Ref`, then followed by the arguments specified by the developer. In Listing 2.7 on Line 5 the method `set` found in `Ref` can be seen, it is called by `ref-set`. It it very similar to `alter` but instead of using the return value from a function as argument to `doSet`, `set` uses the value found in the argument `val`.

```
1  Object doSet(Ref ref, Object val) {
2    if ( ! info.running())
3      throw retryex;
4    if (commutes.containsKey(ref))
5      throw new IllegalStateException("Can't set after commute");
6    if ( ! sets.contains(ref)) {
7      lock(ref);
8      sets.add(ref);
9    }
10   vals.put(ref, val);
11   return val;
12 }
```

Listing 2.8: The `doSet` Java method in the `LockingTransaction` class

In Listing 2.8 the method `doSet` can be seen which handles setting the value of a `Ref` in a specific transaction. `doSet` checks in Line 2 that transaction is still running, if not the transaction is forced to retry. This check is necessary due to possible interleaving problems. The method then checks if the `Ref` has already been used in a `commute` operation, if so an `IllegalStateException` is thrown in Line 4 since an `alter` or a `ref-set` operation is not allowed after a `commute` operation has been executed on the same `Ref`. This is because `commute` breaks the consistency of the transaction to provide a higher degree of concurrency compared to `alter` and `ref-set` as described in Section 2.4.5. On Line 6 the instance property `sets` is checked whether or not it already contains the `Ref` found in the argument to `doSet` to check that the transaction have the necessary locks. In Line 7 the transaction will try to get the write lock for the `Ref` that is added to the `LockingTransaction` instances `sets` property. If the write lock is not acquired then the transaction aborts. Finally the new value of the `Ref` in the transaction is set by adding the `Ref` and the value `val` to the instance property `vals` which is a Java `HashMap` containing tuples of `Ref`s and `Objects`. The new value of the `Ref` is finally returned by `doSet`.

2.4.5 commute

The `commute` function takes the same three arguments as `alter` as seen in Listing 2.1 on Line 9, as it performs the same operations but allows commutative updates as described in Section 2.3.

```
1 public Object commute(IFn fn, ISeq args) {
2   return LockingTransaction.getEx().doCommute(this, fn, args);
3 }
```

Listing 2.9: The `commute` Java method in the `Ref` class

The `commute` function invokes the `commute` method on the instance of `Ref` passed as the argument `ref`. The source code for the `commute` method in the `Ref` class can be seen in Listing 2.9. The method takes two arguments, a function `IFn` and an `ISeq` which are arguments for the function passed as `IFn`. The method gets the currently running transaction using `getEx` on `LockingTransaction`, and then calls `doCommute` on the returned transaction with the `Ref` instance, and the function and arguments passed by the developer.

```
1  Object doCommute(Ref ref, IFn fn, ISeq args) {
2    if ( ! info.running())
3      throw retryex;
4    if ( ! vals.containsKey(ref)) {
5      Object val = null;
6      try {
7        ref.lock.readLock().lock();
8        val = ref.tvals == null ? null : ref.tvals.val;
9      } finally {
10       ref.lock.readLock().unlock();
11     }
12     vals.put(ref, val);
13   }
14   ArrayList<CFn> fns = commutes.get(ref);
15   if (fns == null) {
16     commutes.put(ref, fns = new ArrayList<CFn>());
17   }
18   fns.add(new CFn(fn, args));
19   Object ret = fn.applyTo(RT.cons(vals.get(ref), args));
20   vals.put(ref, ret);
21   return ret;
22 }
```

Listing 2.10: The `doCommute` Java method in the `LockingTransaction` class

In Listing 2.10 the method `doCommute` can be seen which executes the `commute` function, and sets the resulting value as the transactional value for the `Ref` in the currently running transaction. Like `doSet`, `doCommute` checks on Line 2, that the transaction is currently running. The method then checks if the `Ref` has an existing transactional value in the `vals HashMap` as described in Section 2.4.4, this is on Line 4. If no transactional value exists

then the method tries to get read locks to read and set the most recent value, or `null` if the `Ref` does not contain a value, as the transactional value. This means that `commute` will break consistency by using the most recent value if it is newer than the transaction.

The method then checks if any other function have been executed on the `Ref` using `commute` on Line 14. If other functions have been called with the `Ref` as input then this `commute`s function and arguments is added to the existing list. These `commute` functions are executed again with the most recently committed value when the transaction enters commit state. This enables commutative modifications of the `Ref` through `commute`.

Finally the transactional value is updated by executing the function passed to commute with the `Ref` and any additional arguments, this can be seen on Line 19. The resulting value is stored as the new transactional value in the `vals HashMap`, overwriting the value extracted from the `Ref` earlier in the method. The updated value is finally returned to the caller, allowing it to be returned by `commute` inside the `dosync` block.

2.4.6 ensure

The function `ensure` takes one argument shown by its signature seen in Listing 2.1 on Line 5. First `ensure` calls the Java method `touch` on the `Ref` instance to protect it from modification by other transactions. `ensure` then it calls the Java method `deref` to return the transactional value of the same `Ref`, `deref` was described in Section 2.4.3. The purpose of `ensure` is to protect `Ref`s from writes by other transactions without modifying them while allowing more transactions to call `ensure` on the same `Ref` instance concurrently. This gives a higher degree of concurrency compared to using `alter` to protect `Ref`s as more transactions can call `ensure` on the same `Ref`s at the same time where `alter` gives one transaction exclusive access.

```
1 public void touch(){
2   LockingTransaction.getEx().doEnsure(this);
3 }
```

Listing 2.11: The `touch` Java method in the `Ref` class

The method `touch` can be seen in Listing 2.11. It calls `doEnsure` with the `Ref` instance as argument. The currently running `LockingTransaction` instance is acquired by calling the method `getEx` on Line 2. `doEnsure` is then called through the `LockingTransaction` instance with the `Ref` instance as argument.

```
1  void doEnsure(Ref ref) {
2    if ( ! info.running())
3      throw retryex;
4    if (ensures.contains(ref))
5      return;
6    ref.lock.readLock().lock();
7
8    if (ref.tvals != null && ref.tvals.point > readPoint) {
9      ref.lock.readLock().unlock();
10     throw retryex;
11   }
12
13   Info refinfo = ref.tinfo;
14
15   if (refinfo != null && refinfo.running()) {
16     ref.lock.readLock().unlock();
17
18     if (refinfo != info) {
19       blockAndBail(refinfo);
20     }
21   } else {
22     ensures.add(ref);
23   }
24 }
```

Listing 2.12: The `doEnsure` Java method in the `LockingTransaction` class

The method `doEnsure` can be seen in Listing 2.12. The first thing the method does is to check if the transaction is running in Line 2. After that it checks if the `Ref` instance already is in the set `ensures` that contains ensured `Ref`s in Line 4. If the instance is found in the set then the `Ref` instance is already ensured the function returns. Otherwise a read lock is acquired on the `Ref` instance as seen in Line 6. In Line 8 the `Ref` is checked if it already contains a transactional value and that the value is older than the start point of the transaction, otherwise the transaction is forced to retry. The transactional information of the `Ref` instance is then retrieved in Line 13. After that the function checks if the `Ref` instance exists and checks if its already used in another transaction as seen in Line 15. If used by another transaction the transaction will throw its read lock and call `blockAndBail` in Line 19. If the `Ref` is not used by another transaction the instance of the `Ref` is added to the set of ensured `Ref`s in Line 22. This means that the `Ref` is now protected against modifications from other transactions by the STM implementation.

2.4.7 blockAndBail

```
1 private Object blockAndBail(Info refinfo){
2   stop(RETRY);
3   try {
4     refinfo.latch.await(LOCK_WAIT_MSECS, TimeUnit.MILLISECONDS);
5   } catch(InterruptedException e) {
6
7   }
8   throw retryex;
9 }
```

Listing 2.13: The `blockAndBail` Java method in the `LockingTransaction` class

The method `blockAndBail` is called from the method `doEnsure` described in Section 2.4.6 and the method `lock`. This method is called when the current transaction has to wait on another older transaction because it needs a specific lock of one of its `Ref` instances. It will either wait until the transaction that is blocking has stopped or waits a predefined amount of time defined by Clojure's STM implementation before aborting. The current transaction will remain blocked until its `latch` to reaches 0 in Line 4.

2.4.8 barge

```
1  private boolean bargeTimeElapsed(){
2    return System.nanoTime() - startTime > BARGE_WAIT_NANOS;
3  }
4
5  private boolean barge(Info refinfo){
6    boolean barged = false;
7    if(bargeTimeElapsed() && startPoint < refinfo.startPoint){
8      barged = refinfo.status.compareAndSet(RUNNING, KILLED);
9        if(barged)
10         refinfo.latch.countDown();
11   }
12   return barged;
13 }
```

Listing 2.14: The `barge` Java method in the `LockingTransaction` class

The method `barge` is called in the method `run` described in 2.4.2 and the method `lock`. This method is called when the current transaction needs

a `Ref` instance that currently is locked by another transaction. `barge` forces the other transaction to abort if the current transaction is older and have been alive for more then a static threshold defined by Clojure, the static threshold is checked by the method `bargeTimeElapsed` which can be seen as part of Line 7. It aborts the other transaction by setting its status to `KILLED` as seen in Line 8 and then signaling to other waiting transactions that it has been killed by counting down its `latch` to zero in Line 10. The call to `countDown` is needed to ensure that the barged transaction is killed even though it is blocked.

2.5 Exploration

The analysis of Clojure's STM implementation, described in this chapter, served as the base of a series of experiments that were conducted to determine how side-effects can be used in a transactional safe manner and how transaction control can be used to avoid to depend on synchronisation constructs such as Java mutexes or semaphores. This series of experiments, further described in Appendix A and B, were carried out without relying on the literature as this would allow for ideas that favoured Clojure's STM implementation and were compared with solutions presented in the literature to relate the experiments to the literature. This section start by describing three examples used to evaluate each experiment, followed by a description of each of the general approaches, found in the literature and a summary of the experiments that were carried out using the described approach. At the end of the section it will be discussed if it is possible to combine the experiments into a unified solution and thereby gaining all of the advantages of the individual experiments.

2.5.1 Examples

To evaluate and compare the different methods for performing side-effects in transactions we created three examples, one representing each of the first three limitations of Clojure STM implementation as described in Section 1.1. Each example is created to be the smallest possible showcase of each problem. The last two limitations are not covered by our examples, as they refer to `Agent`s which is not used in any experiment and refer to transaction control which cannot be compared to methods for performing side-effects in transactions. The merit of adding transaction control to Clojure will instead be evaluated as part of our entire implementation in Section 4.

The first example in Listing 2.15 describes a transaction with a counter that must be printed in Line 4 before it is increased or set to zero if it reaches ten. In Clojure, this example will always print the value of the counter each time the transaction executes even if it retries, this is not the

```
1 (def counter-ref (ref 0))
2
3 (dosync
4   (println (deref counter-ref))
5   (if (< counter-ref 10)
6     (alter counter-ref inc)
7     (ref-set counter-ref 0)))
```

Listing 2.15: Printing counter to standard output stream

desired behaviour as it should only print the value if the counter is increased or set to zero.

```
1 (def arraylist-ref (ref (java.util.ArrayList.)))
2 (def log-ref (ref (writer  "log.txt" :append true)))
3
4 (dosync
5   (alter arraylist-ref .add 0)
6   (alter log-ref .write "Added 0 to arraylist-ref"))
```

Listing 2.16: Modifying ArrayList and logs

The second example in Listing 2.16 describes a transaction that first adds an element to an `ArrayList` in Line 5 and then logs that it did it in Line 6. These two actions are side-effects that should only happen together or not at all. In Clojure, this example will try to execute the first `alter` in Line 5 and then the next in Line 6. This is not acceptable behaviour because an element could be added to the `ArrayList` without the logging taking place.

```
1 (def keys-ref (ref []))
2 (def rows-ref (ref vector-of-rows))
3
4 (dosync
5   (let [row (first (deref rows-ref))
6         next-key (database-insert row)]
7     (alter keys-ref conj next-key)
8     (alter rows-ref rest)))
```

Listing 2.17: Inserting database rows and returning keys

The third example in Listing 2.17 describes a transaction where a `vector` of database rows are inserted into a database on Line 6, the function returns the key assigned to the row from the database which is inserted a `vector` on

Line 7, and last the row inserted are removed from the list of rows on Line 8. The primary aspect of the example is that it both consumes elements from a `Ref`, performs a side-effect, and then adds the result to a `Ref` with the result requiring the same order as produced from the input. This is not possible with STM in Clojure, as either of two `alter` function can cause the transaction to abort, which in turn would execute the insertion of the same row into the database multiple times.

2.5.2 Defer

Defer [10] is an approach which postpones the execution of code with side-effects to take place after the transaction is ensured to commit. This approach has been studied in [11, 12, 13, 14]. The Defer approach has been achieved by implementing a two-phase commit where the transaction's commit operation is split into a validate and a finalise step. The developer is able to specify code that is executed in between the two steps. The defer experiments have been described in Section A.1. Two defer experiments were performed, called After-Commit in Section A.1.1 and Lazy Evaluation in Section A.1.2.

After-Commit is a macro named `dosync-ac` that is capable of executing code synchronously after a transaction commits, with additional earlier versions shown in Appendix C. The solution does not allow additional operations compared to using Clojure's `agent`s for side-effects as both executes outside a transaction, but is synchronous while `agent`s are asynchronous. Furthermore an `agent` require any function it executes to accept the `agent`'s current value as the first argument. Overall `dosync-ac` simplifies Clojure's existing approach to side-effects, but it does not solve all problems as all side-effects must be packaged and executed after the transaction has finished, meaning that transactional values cannot depend on the result of side-effects.

After-Commit is different compared to other defer approaches found in [11, 12, 13] because these approaches only have their code execution postponed until before the finalisation step whereas After-commit is executed after the finalisation step. This is a limitation because other concurrent transactions can be interleaved and commit a value to the `Ref` before After-commit is executed.

The second defer experiment is to use lazy evaluation to delay the evaluation of expressions that perform side-effects until the transaction is guaranteed to commit. The use of lazy evaluation is found in other languages like Haskell [15] whose use of thunks was the inspiration for our own implementation. The solution allows the developer to mark expressions for execution just before the transaction commits, at a time where it is certain that the transaction will commit. This allows the developer to use side-effects, and write to any `Ref`s where the write-lock already has been acquired, which

in most cases is performed automatically by the macro `le`. However the solution is not without problems since the developer must manually specify `Ref`s that are modified within functions called by a lazy expression. This is due to the fact that changing the value of a `Ref` without acquiring the write-locks before the execution could make the transaction abort, removing the advantage of lazy expressions. While both lazily and strictly evaluated expression can be interleaved in the code, it can be complex to reason about the execution order when some expressions are delayed and some are not, a problem made more clear when expressions use data computed by other expressions.

Lazy evaluation is similar compared to the defer approaches found in [11, 12, 13] where the is executed in-between the validation and finalize step. It is similar because we make sure we have all necessary locks before executing code.

2.5.3 Compensate

The approach to side-effect handling called Compensate [10] reverts a side-effect which just has been done by the transaction. Compensate does this by letting the developer specify compensation code, that is then executed when a transaction fails to commit. This approach is described in [11, 12, 13, 14]. The compensate experiment named Undo for allowing side-effects is presented in Section A.2.

Undo allows the developer to specify code that will be executed if the transaction aborts. The idea is that this code should be able to compensate for side-effects executed by the aborting transaction. The experiment provides macros for adding expressions and functions to be executed if the transaction aborts. The functions are executed as part of the transaction so `Ref`s can be modified if needed. The problem of acquiring write-locks is not present in this solution, as `Ref`s must have been changed by the transaction before there is anything to undo, so the needed write-locks have already been acquired for removing previous changes. However not all side-effects can be removed through additional code, such as for example a statement printed to the screen. Other operations like a call to the database could be too expensive to apply and remove multiple times.

Undo is similar to [11, 12, 13] as they all allow the execution of code when a transaction aborts. This is in all cases done to compensate for a previously executed side-effect. [11] is slightly different because they additionally gives the possibility to not just compensate but to make an aborted transaction commit successfully by modifying the state of the transaction.

2.5.4 Irrevocability

A third approach to controlling side-effects in transactions is to introduce Irrevocability [10] which can be seen as a promise to the developer that the transaction will commit without retrying. This is a widely known approach and has been researched in [16, 17, 18, 19, 20]. In Section A.3 the irrevocable experiment named Check-run is presented.

The idea behind Check-Run is to use a lazy expression for an entire `dosync` block. The experiment was implemented by utilising a single lazy evaluation expression containing the content of the `body` argument sent to the `dosync-checked` or the `dosync-checked-ref` macro. This hides the problem of reasoning about interleaved lazily and strictly evaluated expressions, by providing a uniform abstraction where every expression is evaluated lazy. This allows the developer to both use side-effects and modify transactional data, without concern for the transaction aborting. Check-run inherits the problem from lazy expressions, of acquiring locks for functions that modify `Ref` instances not directly referenced in the transaction. The solution is implemented with the same utility to extract `Ref`s as the macro `le`, which is unable to extract `Ref`s not available directly in the Clojure code, for example a `Ref` changed by a function call must declared manually.

Check-Run is similar to what has been researched in [16] as it allows more than one irrevocable transaction to run at a time if they are not conflicting. Check-run is therefore different compared to [17, 18, 19, 20] because they only allow one irrevocable transaction to run at a time. Multiple irrevocable transactions are possible concurrently with Check-Run. This is quite different compared to relying on a global lock to see if any irrevocable transaction is currently running as done in [18, 20]. A more aggressive approach compared to Check-Run is presented in [19] where the authors abort all other transactions if they conflict with the irrevocable transaction. A more passive approach compared to Check-Run is presented in [17] that waits until all conflicting transactions have finished and blocks the creation of new transactions before the irrevocable transaction has been executed.

2.5.5 Transactional Control

The concept of Transaction Control makes it possible to control a transaction's behaviour. A transaction can be forced to have a specific behaviour when it encounters an explicit construct. These behaviours include that a transaction can be forced to abort by using the construct `retry` [11, 16, 21], to wait for a condition [22] or to executive a different operation `or-else` if the first would force the transaction to aborts [21].

The experiment is presented in Section A.4 and allows the developer to explicitly control the execution of the transaction. This solution is known from Haskell [21] where the `retry` and `orElse` constructs are used. The con-

structs and their utilisation are known so our experimentation focused on how the two constructs could be implemented in Clojure. We implemented both the `retry` and `or-else` function for use in Clojure, in addition to implementing `retry-all` which allows the transaction to block until all `Ref`s have been written to, instead of just one of them like in Haskell. We also implemented `terminate` which aborts a transaction, and prevents it from retrying. These constructs do not allow for side-effects to be used with transactions, but gives the developer direct control over the transaction, removing the need for synchronising through side-effects such as exceptions and Java monitors which is currently needed in Clojure, if a transaction should abort before it completes. The experiment complements a solution for allowing side-effects in transactions by removing the need for some side-effects.

This `retry` approach is similar to what have been researched in [11, 16, 21]. In [21] the retry construct blocks the transaction from further execution until one of the references used in the transaction changes. This is similar to what `retry` can do if its called with no arguments. In [16] the retry construct causes the transaction to retry without blocking on variables or looking for changes in-transactional variables. `retry` deliberately avoids this to only execute the transaction again if changes in-transactional variables have happened. The approach in [11] gives the developer the opportunity to execute code in between the validation and finalisation step where it is possible to specify that the transaction should retry. The `or-else` approach that was taken here is similar to what was presented in [21] where the implementation could choose one of two functions, instead of a list of expressions as in our experiment.

2.5.6 Discussion and Insight

All of the described experiments allowing side-effects to be used in a transaction have pros and cons. The defer experiment After-commit is not executed as part of the transaction, a problem solved in the Lazy expressions experiment, but using Lazy expressions requires that locks for `Ref`s are acquired beforehand. The irrevocable experiment Check-run allows both side-effects and modification to transactional code, but requires the developer to specify what `Ref`s are used in some cases. The compensation based solution is not capable of undoing all side-effects and could be computationally expensive, but cannot fail due to the developer not specifying the correct `Ref`s. While transactional control supplements the other experiments, it does not help to execute side-effects inside a transaction.

It has been shown that the approaches can be combined to give the best of both worlds, as side-effects in a transaction can both be deferred but also compensated as has been done in [12, 13, 14]. In [16] the authors showed that the combination of irrevocable transactions and explicit transaction `retry` construct is counter intuitive because if a `retry` operation occurred

in an irrevocable transaction it would tell the system to abort a transaction that cannot abort.

Based on this insight and our experimental implementations, a generic event handling system will be implemented and described in Section 3.1. The event handling system will combine the three above mentioned experiments to allow side-effects in transactions. Especially the implementation of lazy evaluation and the introduction of `Thunk`s made it clear that a generic event handling system would be able to satisfy our needs since a `Thunk` is nothing more than a subroutine with memorisation executed by a specific event, in this case at the time of commit for a transaction. The use of `Thunk`s from lazy evaluation can be reused when executing events in the more generic event handling system. What then needs to be added in the Java part of the Clojure runtime is storage and execution of events in a generic way, in addition to a way to add events to Clojure programs. Such facilities would allow the developer to create his/her own events and the runtime to have some built-in events for when for example a transaction commits. The experiments were needed to arrive at the generic event handling system as a solution to the handling of side-effects as the literature on the subject do not suggest a similar generic approach.

CHAPTER 3

Implementation

The implementation of the more generalised event handling system idea, seen in Section 2.5, will be described in detail in this chapter. Information on how to obtain the code is described in the preface.

3.1 Event Handling System

This section will describe the implementation of the event handling system starting by describing the interface available to the developer, followed by a detailed explanation of its implementation. The event handling system is based on the generalisation of the experiments described in Section A.1, A.2 and A.3. The main idea is to allow the developer to specify code that will be executed when a defined event happens. The event handling system allows the developer to create events that can be used to cause code to be executed. Examples of useful events in an STM implementation could be when the transaction commits or aborts.

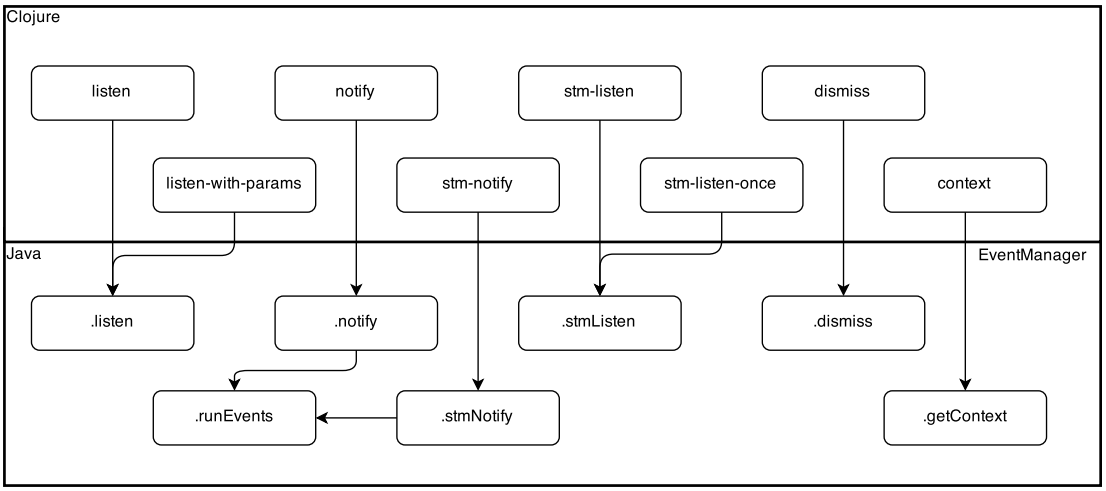

Figure 3.1: The architecture of the event handling system

The following notion will be used for the event handling system, as it matches the terminology used in Java; When an event happens during execution time it is said that the event is notified since the event listeners, specified code waiting to be executed, are notified that they should execute. References to data structures must be stored inside `Ref`s and accessed via `alter`, `commute`, `ensure` or `ret-set` if the developer wishes to modify the data structures despite the reference itself never being modified. This allows Clojure's STM implementation to synchronise access to them between threads.

The interface is written in Clojure as an extension to the Clojure part of the runtime, while the implementation is developed as an extension to the Java part of the runtime. An overview of the event handling system architecture is shown in Figure 3.1 where methods in the Clojure part is displayed in the upper rectangle and the lower rectangle contains Java methods encapsulated in the class `EventManager`.

3.1.1 Clojure implementation

The interface of the event handling system consists of three parts, an interface for the general system, an interface for the STM event handling system and an interface for the events added to Clojure's STM implementation. The STM event handling system allows the use of events inside of transactions in a transactional safe manner, these events are private to each transaction and therefor also removed when a transaction has committed. Notifying and creating listeners for non-transactional safe global and thread local events are not allowed inside of transactions.

The three parts of the interfaces can be seen in Listing 3.1. The generalised event handling system supports almost the same functionality inside and outside transactions, but the functionality are differs since the STM versions of the functions are the only ones allowed inside of transactions to ensure events are used in transactional safe manner. Using one of the two functions `listen` and `stmListen` shown on Line 2 and 12, it is possible to listen for an event to be triggered, resulting in the code given to the listener as argument being executed. The two functions take three arguments; `event-key` which is the event key to listen for, `event-fn` which is the function to execute when event is triggered and `event-args` which is the arguments for `event-fn`. The function `listen-with-params` on Line 3 provides the developer with more control over the different semantics for the event, compared to using `listen`. `listen-with-params` has two additional arguments; `thread-local` which specifies whether or not the event should be thread local if `true` or global if `false`, and `delete-after-run` which specifies if the event listeners should be deleted after the event was triggered for the first time. The function `stm-listen-once` on Line 13 enables the developer to delete the event listeners after the event has been triggered

```
1  ; Interface for use of events outside transaction
2  (listen event-key event-fn & event-args)
3  (listen-with-params event-key thread-local delete-after-run event-fn & event-args)
4
5  (notify event-key)
6  (notify event-key context)
7
8  (dismiss event-key event-fn dismiss-from)
9  (context)
10
11 ; Interface for use of events inside transaction
12 (stmListen event-key event-fn & event-args)
13 (stm-listen-once event-key event-fn & event-args)
14
15 (stm-notify event-key)
16 (stm-notify event-key context)
17
18 (lock-refs func & body)
19
20 ; Interface for the events in the STM implementation
21 (on-abort & body)
22 (on-abort-fn event-fn & event-args)
23
24 (on-commit & body)
25 (on-commit-fn event-fn & event-args)
26
27 (after-commit & body)
28 (after-commit-fn event-fn & event-args)
```

Listing 3.1: The interfaces for the event handling system

once. Events created inside a STM transaction cannot be defined as non-transactional to ensure the same event listener is not created multiple times due to the transaction aborting.

For triggering an event and notifying any listeners to a specific event the functions `notify` and `stm-notify` on Lines 5 and 15 are used. Both functions are overloaded on the number of arguments. If the functions are called with only the `event-key` argument the event is triggered with no additional context, if also the `context` argument is given this context will be available inside the functions listening. The `context` argument can be any data, and this data is then made available for use inside the event listeners. In an event listener the `context` is retrieved through a function named `context`, as seen on Line 9, instead of being added as an extra argument to the listener as done by Clojure for `Agent`'s current value, to not make developers add an extra argument to a function for when it is used as a event listener. To remove a function from a specific event the function `dismiss` seen on Line 8 is available. `dismiss` takes three arguments, first

the `event-key` and `event-fn` are specified identify what function should be removed from which event. Thirdly the argument `dismiss-from` specifies whether the `event-fn` is to be removed from the thread local event, the global event or both. `dismiss` cannot be used to remove a transactional event and no `stm-dismiss` is implemented to encourage small transactions, further discussion of this decision is found in Section 5.5.

The events added to Clojure's STM implementation are a way to listen for events triggered by the STM implementation itself when a transaction is guaranteed to commit, but still running, by using `on-commit` and `on-commit-fn`, after it has been committed by using `after-commit` and `after-commit-fn` or it have aborted by using `on-abort` and `on-abort-fn`. Each pair differs only in which events they listens for, and the only difference between the macro and the function in each set is that the macro takes a list of expressions as argument, while the function takes another function and arguments for it. The macros `on-commit`, `after-commit` and `on-abort` all take a single argument which is the body of expressions to be run when the event is triggered. The functions `on-commit-fn`, `after-commit-fn` and `on-abort-fn` all take two arguments; `event-fn` specifies which event to call when the event is triggered and `event-args` the arguments for the function to be called. Both are provided to allow the developer to use the most appropriate syntax for the problem at hand.

```
1  (defmacro lock-refs
2    [func & body]
3    ; Extracts the lexical scoped symbols from the environment
4    (let [lexically-scoped-bindings (keys &env)
5          locking-fn (case func
6                       ensure #(ensure %)
7                       commute #(commute % identity)
8                       alter #(alter % identity)
9                       (throw (IllegalArgumentException. "func must be ..."))))]
10     `(do
11        (doseq [r# (extract-refs '() '~body ~@lexically-scoped-bindings)]
12          (~locking-fn r#))
13        ~@body)))
```

Listing 3.2: The `lock-refs` function

The macro `lock-refs` seen in Listing 3.2 enables the developer to get the access type specified in argument `func`, `alter`, `commute`, or `ensure`, on all extractable `Ref`s inside the argument `body`. In Line 4 the keys of the `&env` `Map` are extracted and are bound to the symbol `lexically-scoped-bindings`. `&env` are provided as a special symbol inside `defmacro`, and provides access to lexically scoped symbols from where the macro are expanded as these symbols are not available through `resolve`. The function defining what accesses

should be acquired is bound to the variable `locking-fn` in Line 5. In Line 11 the `Ref`s are extracted from `body` using the helper function `extract-refs` which iterates through the expressions in `body` to resolve and extract any additional `Ref`s not lexically bound. `lexically-scoped-bindings` are also passed to the function for the lexically scoped `Ref`s to be combined with those extracted by `extract-refs`, so the resulting list contains all `Ref`s directly accessible in the code. Finally in Line 12 the `locking-fn` function is called on all the `Ref`s found, both lexically bound and extracted by the function `extract-refs`.

3.1.2 Java implementation

All of the functions described above calls directly into the Java class `EventManager` as can be seen in Figure 3.1.

```
1  public static EventFn listen(Keyword key, IFn fn, ISeq args,
2    boolean threadLocal, boolean deleteAfterRun) {
3
4    if (LockingTransaction.isRunning()) {
5      throw new IllegalStateException("Listen is not allowed in
6      a transaction, use stmListen");
7    }
8
9    Map<Keyword, ArrayList<EventFn>> eventMap = (threadLocal) ?
10     EventManager.threadlocalEvents.get() :
11     EventManager.globalEvents;
12
13   EventFn listenerEventFn = new EventFn(fn, args, deleteAfterRun);
14
15   synchronized (eventMap) {
16     if ( ! eventMap.containsKey(key)) {
17       eventMap.put(key, new ArrayList<EventFn>());
18     }
19     eventMap.get(key).add(listenerEventFn);
20   }
21   return listenerEventFn;
22 }
```

Listing 3.3: The `listen` Java method in the `EventManager` class

The functions `listen` and `listen-with-params` both call the static method `listen` on the `EventManager` which can be seen in Listing 3.3. If the method is called when executing a transaction the method will throw an `IllegalStateException` as seen in Line 6 since `listen` is not transactional safe, due to it not preventing the same event being added multiple times if the transaction abort. Next the correct Java `Map` is selected based on whether or not the event listened for is thread local or a global event, this

happens in Line 9. The key of the `Map` is an instance of the class `Keyword` and the value for each entry is a Java `ArrayList` containing instances of the type `EventFn` which encapsulates an event listener function with arguments. A new instance of `EventFn` is initialised in Line 13.

The function then enters a block `synchronized` on the `Map`, this is to synchronise access to the `Map` with global events, as using an internally synchronised data structure would allow a race conditions between the check on Line 16 for if the `Map` already contain an `ArrayList` for the given `key` and the time an empty list is added. The `synchronized` block is also used for the thread local `Map`, as it allows for a simpler implementation then using additional branching and performance benchmarks out of the scope of this project would be needed to determine which is preferable in terms of performance. The function then adds the `EventFn` to the `ArrayList` indicated by `key`, this is done on Line 19, before returning the `EventFn` on Line 21.

```
1  public static void notify(Keyword key, Object context) {
2
3    if (LockingTransaction.isRunning()) {
4      throw new IllegalStateException("Notify is not allowed in
5      a transaction, use stmNotify");
6    }
7    synchronized (EventManager.globalEvents) {
8      EventManager.runEvents(key, EventManager.globalEvents, context);
9    }
10   EventManager.runEvents(key, EventManager.threadlocalEvents.get(), context);
11 }
```

Listing 3.4: The `notify` Java method in the `EventManager` class

When calling the function `notify` the static method `notify` on the `EventManager` class will be called, this method can be seen in Listing 3.4. This method takes only two arguments, the `key` which specifies which event's listeners should be notified, and `context` which is made available through the function `context` described earlier. As with `listen`, `notify` is not safe to use in transactions, therefore an `IllegalStateException` is thrown in Line 5 if the method is called while executing a transaction. This is a design choice to ensure only events added doing the transaction are executed, and the developer not accidental executes a non transactional safe global event. If not in a transaction, all listeners both thread locally and globally will be notified in Line 8 and 10.

The functions `stm-listen` and `stm-listen-once` call the static method `stmListen` in the `EventManager` class. The `stmListen` method is very similar to the `listen` method, therefore its source code will not be shown here. The difference between `stmListen` and `listen` is that the `Map` from the

event key to the `ArrayList` of `EventFn`s are stored on the currently running transaction instead of in the `EventManager` itself. The function `stm-notify` which calls the static method `stmNotify` in the `EventManager` class is similar to `notify` which is why the source is not shown here. The difference between the two is similar to the difference between `listen` and `stmListen` where the `Map` from the event key to the `ArrayList` of `EventFn`s, are stored as an instance variable on `LockingTransaction` and not on `EventManager` itself, binding the event listeners to the currently running transaction and only to that single transaction.

```
1  static void runEvents(Keyword key, Map<Keyword,
2    ArrayList<EventFn>> events, Object context) {
3
4    if (events.containsKey(key)) {
5      ArrayList<EventFn> toDeleteAfterRun = new ArrayList<EventFn>();
6      EventManager.context.set(context);
7
8      for (EventFn fn : events.get(key)) {
9        fn.run();
10
11       if(fn.deleteAfterRun()) {
12         toDeleteAfterRun.add(fn);
13       }
14     }
15
16     EventManager.context.set(null);
17     events.get(key).removeAll(toDeleteAfterRun);
18   }
19 }
```

Listing 3.5: The `runEvents` Java method in the `EventManager` class

In Listing 3.5 the method `runEvents` can be seen. `runEvents` is called by the methods `notify` and `stmNotify` in `EventManager` and it sets the context and executes the event listener functions. The methods checks if there are any events to be executed by checking if the argument `events` contains an entry for the `key` specified as seen in Line 4. In Line 5 an empty `ArrayList` is initialised for holding `EventFn`s that should be removed from the event after it has been executed. The class property `context` which is set in Line 6 is thread local to ensure that triggering events concurrently does not result in faulty behaviour where event listeners would see the context of another event. On Line 8 a `for`-loop iterates through the list of `EventFn` instances, the body of the loop executes each listener as shown on Line 9 and adds them to an `ArrayList` of `EventFn`s if they should be removed from the list of events after all event listeners have been run. The reason for the `toDeleteAfterRun ArrayList` and first removing the `EventFn`s after

all event listeners have been run is that trying to alter an `ArrayList` while looping through it will cause an `ConcurrentModificationException`. After all event listener functions have been run the thread local class property `context` are set to `null` in Line 16 to ensure the information is not accessible after the events have been run, and finally event listeners to be deleted are removed in Line 17.

```
1  public static void dismiss(Keyword key, EventFn eventFn,
2    Keyword dismissFrom) {
3
4    if (LockingTransaction.isRunning()) {
5      throw new IllegalStateException("Dismiss is not allowed in
6      a transaction, events are dismissed with the transaction");
7    }
8
9    if (dismissFrom != DISMISSALL &&
10       dismissFrom != DISMISSGLOBAL &&
11       dismissFrom != DISMISSLOCAL) {
12     throw new IllegalArgumentException("The dismissFrom keyword
13     must be either :all, :global or :local");
14   }
15
16   if (dismissFrom == DISMISSALL  dismissFrom == DISMISSGLOBAL) {
17     synchronized (EventManager.globalEvents) {
18       if (EventManager.globalEvents.containsKey(key)) {
19         EventManager.globalEvents.get(key).remove(eventFn);
20       }
21     }
22   }
23
24   if (dismissFrom == DISMISSALL  dismissFrom == DISMISSLOCAL) {
25     if (EventManager.threadlocalEvents.get().containsKey(key)) {
26       EventManager.threadlocalEvents.get().get(key).remove(eventFn);
27     }
28   }
29 }
```

Listing 3.6: The `dismiss` Java method in the `EventManager` class

The static method `dismiss` in the `EventManager` class can be seen in Listing 3.6. Just like the methods `listen` and `notify` in the `EventManager` class `dismiss` is not safe to use in transactions. Therefore an `IllegalStateException` will be thrown in Line 6 if the method is called when executing a transaction. In Line 9 the validity of the `dismissFrom` argument is checked to one of the three valid values; `DISMISSALL`, `DISMISSGLOBAL` or `DISMISSLOCAL`. The `dismissFrom` argument specifies if the value of the `eventFn` argument should be removed from both the local thread event and/or the global event.

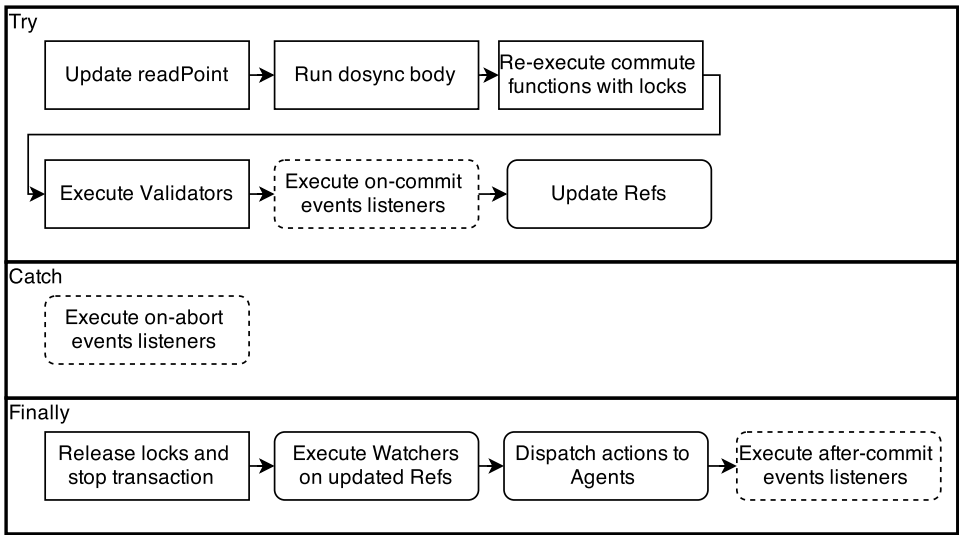

Figure 3.2: The major parts of the `run` method with the addition of the event handling system, squares show parts that can be re-executed, while rounded squares cannot, dashed lines indicate the added events

Events are added to the STM implementation in the `run` method on `LockingTransaction`, the updated flow `run` can be seen in Figure 3.2. The dashed, rounded squares are the events added to the STM implementation. The on-commit event is executed right before the `Ref`s are updated and the transaction commits its changes to the transactional values. The on-abort event is executed when the transaction is forced to abort and the after-commit event is executed when actions to `Agent`s are dispatched.

3.2 Transaction Control

This section will describe the implementation of the transaction control system starting by describing the interface available to the developer, followed by a detailed explanation of its implementation. An overview of the implementation can be seen in Figure 3.3. The design and implementation is a continuation of the experimental design shown in Section A.4, and our experimental implementation described in Section B.5. The implementation consists of three parts. The first part is a set of Clojure functions in the Clojure part of the runtime described in Section 3.2.1 and can be seen in the figure as the upper rectangle named Clojure. The second part is an the interface to the Java part of the runtime in the class `RT`, it is described in Section 3.2.2 and can be seen in the figure as a rectangle named Runtime (RT). The third part is the Java methods in the `LockingTransaction` class and the `STMBlockingBehavior` class found in the Java part of the runtime described in Section 3.2.4 and can be seen in the figure as the rectangles

named LockingTransaction and STMBlockingBehavior.

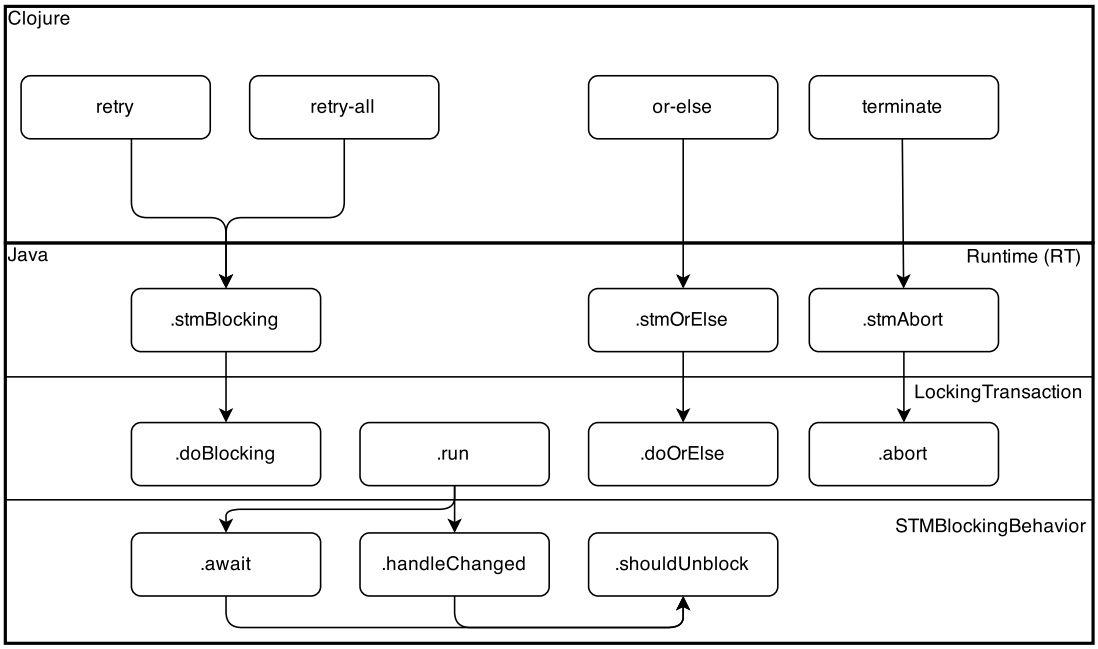

Figure 3.3: The architecture of the transactional control system

3.2.1 Clojure Implementation

The implementation of the Clojure interface for transaction control be seen in Listing 3.7.

The functions `retry` and `retry-all` can be seen in Listing 3.7 on Line 1 and Line 6 respectively. They provides the same interface but have different semantics. Both block until a set of specified `Ref`s are written to by another transaction, `retry` block until any of the `Ref`s are written to, while `retry-all` block until all the `Ref`s in the set are updated. Both functions have three overloads based on the number of arguments given to the function. When no arguments are given the functions will block based on the `Ref`s read in the transaction. When the function is given one argument it must be a `Ref` or a list of `Ref`s and the function will block on the specified `Ref` or `Ref`s. The last overload takes both `Ref`s, a function returning a boolean and arguments for the function. The function is executed when either one or all of the `Ref`s are written to, depending on if `retry` or `retry-all` are used. If the function evaluates to `True` the transaction is unblocked, and if the function returns `False` the system waits until one or all `Ref`s have been written to again.

The `or-else` function is implemented in Clojure in Listing 3.7 on Line 11. The function takes a list of expressions where it will try to execute

```
1  (defn retry
2    ([] (RT/stmBlocking nil nil nil false))
3    ([refs] (RT/stmBlocking refs nil nil false))
4    ([refs func & args] (RT/stmBlocking refs func args false)))
5
6  (defn retry-all
7    ([] (RT/stmBlocking nil nil nil true))
8    ([refs] (RT/stmBlocking refs nil nil true))
9    ([refs func & args] (RT/stmBlocking refs func args true)))
10
11 (defn or-else
12   [& body] (RT/stmOrElse body))
13
14 (defn terminate
15   [] (RT/stmAbort))
```

Listing 3.7: Clojure functions for the `retry`, `retry-all`, `or-else` and `terminate` constructs

them in the given order until one of them executes without forcing the transaction to retry. The function allows the developer to specify a fallback for operations that do not succeed, instead of continuously aborting the transaction until one specific statement succeeds.

Finally `terminate` is implemented on Line 14 in Listing 3.7. This function makes it possible to terminate a transaction by aborting it without any re-executing. One use of this function is to combine `terminate` with `or-else` to skip a STM transaction if a resource is unavailable and using it would force the transaction to abort.

3.2.2 Runtime Implementation

The class `RT` was extended with transactional control methods for constructing the necessary data structures, `RT` was extended as the class serves as a interface for the Clojure part to the Java part of the code. The `RT` class functions as a facade pattern and serves to hide the internal implementation of Clojure runtime from the standard library, providing operations such as the conversions of data structures between Clojure and Java.

```
1  static public void stmBlocking(Object refs, IFn fn, ISeq args,
2    boolean blockOnAll) throws InterruptedException {
3
4    HashSet<Ref> convertedRefs;
5    if (refs instanceof Ref) {
6      convertedRefs = new HashSet<Ref>();
7      convertedRefs.add((Ref) refs);
8    } else {
9      convertedRefs = new HashSet<Ref>((Collection) refs);
10   }
11   LockingTransaction transaction = LockingTransaction.getEx();
12   transaction.doBlocking(convertedRefs, fn, args, blockOnAll);
13 }
```

Listing 3.8: The `stmBlocking` Java method in the Runtime `RT` class

The method `stmBlocking` implemented in the `RT` class is used by `retry` and `retry-all`. It can be seen in Listing 3.8. The method takes four arguments as seen in Line 1. The method checks if the `refs` argument is an instance of `Ref` in Line 5. If this is true then a new `HashSet` is created in Line 6 and the `Ref` instance is added to the `HashSet` in Line 7. If the `refs` argument is not an instance of `Ref` then it must be a `Collection` of `Ref` instances. A new `HashSet` is created from this `Collection` as seen in Line 9. Having a `HashSet` of `Ref` instances we get the currently running transaction on Line 11 before calling the method `doBlocking` on the `LockingTransaction` object on Line 12. Using a `HashSet` ensures that no duplicate `Ref`s are present in the collection used to check if a thread should be unblocked.

```
1  static public Object stmOrElse(ISeq body) {
2    ArrayList<IFn> fns = new ArrayList<IFn>((Collection) body);
3    if (fns.isEmpty()) {
4      return null;
5    }
6    LockingTransaction transaction = LockingTransaction.getEx();
7    return transaction.doOrElse(fns);
8  }
```

Listing 3.9: The `stmOrElse` Java method in the Runtime `RT` class

The implementation of `stmOrElse` can be seen in Listing 3.9, and takes a single argument called `body`. The method `or-else` must be called with a list of functions as argument. The currently running transaction is acquired in Line 6 before calling the method `doOrElse` with the list of functions as argument on the acquired `LockingTransaction` instance in Line 7.

```
1 static public void stmAbort() throws Exception {
2   LockingTransaction transaction = LockingTransaction.getEx();
3   transaction.abort();
4 }
```

Listing 3.10: The `stmAbort` Java method in the Runtime `RT` class

The `stmAbort` function takes no arguments and can be seen in Listing 3.10. It gets the currently running transaction on Line 2 before calling the method `abort` on the acquired `LockingTransaction` instance in Line 3.

3.2.3 STMBlockingBehavior Implementation

The concept of blocking threads based on a set of `Refs` is introduced in the abstract class called `STMBlockingBehavior`. A blocking behavior is used to block a thread based on a set of `Refs`. Each behavior has different semantics but they all extend the `STMBlockingBehavior`. We have defined four different blocking behaviors which have the following semantics:

STMBlockingBehaviorAny Blocks the thread until any `Refs` defined have been written to by other transactions

STMBlockingBehaviorFnAny Blocks the thread until any `Refs` defined have been written to by other transactions, then the defined function is executed and if it returns `true` the thread unblocks

STMBlockingBehaviorAll Blocks the thread until all `Refs` defined have been written to by other transactions

STMBlockingBehaviorFnAll Blocks the thread until all `Refs` defined have been written to by other transactions, then the defined function is executed and if it returns `true` the thread unblocks

All blocking behaviors are child classes of the abstract super class `STMBlockingBehavior` which defines the interface of the child classes. The constructor of the class is found in Listing 3.11 on Line 6. The constructor takes two arguments `refSet` which is a `Set` of `Refs`, and `blockPoint` which is the `readPoint` of the transaction where `retry` was called, providing a relative timestamp to other transactions. Both arguments are assigned to instance variables on Line 7 and Line 8 respectively. Lastly the initialisation of a `CountDownLatch` on Line 9 is performed. A `CountDownLatch` is a Java class which has the same functionality as a barrier but cannot be reset, it is to block the thread if necessary as it depends on how threads are scheduled.

The class contain three methods `await` on Line 12, `handleChanged` method is seen 18, and the abstract method `shouldUnblock` on Line 24.

```
1  abstract class STMBlockingBehavior {
2    protected Set<Ref> refSet;
3    protected CountDownLatch cdl;
4    protected long blockPoint;
5
6    STMBlockingBehavior(Set<Ref> refSet, long blockPoint) {
7      this.refSet = refSet;
8      this.blockPoint = blockPoint;
9      this.cdl = new CountDownLatch(1);
10   }
11
12   void await() throws InterruptedException {
13     if ( ! shouldUnblock()) {
14       this.cdl.await();
15     }
16   }
17
18   void handleChanged() {
19     if (shouldUnblock()) {
20       this.cdl.countDown();
21     }
22   }
23
24   abstract protected boolean shouldUnblock();
25 }
```

Listing 3.11: The abstract super class for the specific blocking behaviors

The abstract method `shouldUnblock` is overridden by subclasses to return a boolean indicating if the instance of `STMBlockingBehavior` should unblock now. `await` is called by `LockingTransaction` to maybe block the transaction, as changes to the `Ref`s passed to `retry` might have been changed between the time the `STMBlockingBehavior` was created and `await` is invoked. `handleChanged` is called by other transaction after they have committed, checking if the instance of `STMBlockingBehavior` should unblock because of the changes caused by the calling transaction.

The `shouldUnblock` function is implemented by subclasses since their blocking semantics are different. Each implementation of `STMBlockingBehavior` check if a `Ref` have been changed by comparing `blockPoint` to `tvals.point`, which indicates at what time the `Ref` was written to last. The implementation iterates through each `Ref` it is blocked on. This is a difference from the experimental implementation Section B.5, as it iterates through the set of `Ref`s changed by the transaction trying to unblock `STMBlockingBehavior`s.

This change was made as the experimental implementation could deadlock, as `await` did not check for updates to `Ref`s between the instance of `STMBlockingBehavior` being created and the time `await` is called. Instead

of checking for changes using both `point` on `Ref`, and the set of `Refs` provided by a transaction when it commits, is `point` used for both. The change also tunes the implementation for use where each `STMBlockingBehavior` only blocks on a single or a few `Refs`, which we see as the most common use case.

3.2.4 LockingTransaction Implementation

The four subclasses of `STMBlockingBehavior` are used by methods defined in `LockingTransaction` which will be described now. First we will go through the three parts of `retry` and `retry-all`, then the implementation of `or-else` and last the `terminate` implementation is described.

```
1  void doBlocking(HashSet<Ref> refs, IFn fn, ISeq args,
2    boolean blockOnAll) throws InterruptedException, RetryEx {
3
4    if ( ! info.running()) {
5      throw retryex;
6    }
7
8    if (refs == null) {
9      refs = new HashSet();
10     refs.addAll(this.gets);
11   }
12
13   if (blockOnAll) {
14     if (fn != null) {
15       this.blockingBehavior =
16         new STMBlockingBehaviorFnAll(refs, fn, args, this.readPoint);
17     } else {
18       this.blockingBehavior =
19         new STMBlockingBehaviorAll(refs, this.readPoint);
20     }
21   } else {
22     if (fn != null) {
23       this.blockingBehavior =
24         new STMBlockingBehaviorFnAny(refs, fn, args, this.readPoint);
25     } else {
26       this.blockingBehavior =
27         new STMBlockingBehaviorAny(refs, this.readPoint);
28     }
29   }
30   LockingTransaction.blockingBehaviors.add(this.blockingBehavior);
31   throw retryex;
32 }
```

Listing 3.12: The `doBlocking` Java method in the transaction `LockingTransaction` class

The method `doBlocking`, seen in Listing 3.12, adds the correct blocking behavior to the transaction which makes the transaction block when it retries. On Line 4 it checks whether the transaction is in a running state, if not the transaction is forced to retry by throwing `retryex` which is an instance of the `RetryEx` exception, this can happen if a transaction is barged. If the `refs` argument is `null` it means that it should block on all `Ref` instances dereferenced in the transaction, therefore `refs` is assigned to a new `HashSet` containing all `gets`, the set of dereferenced `Ref` instances. The argument `blockOnAll`, in Line 13, decides if the `doBlocking` method must wait for changes in any or all `Ref` instances given in `refs`. If a function is given to `doBlocking` a blocking behavior based on the function is initialised, this is checked on Line 14. The Lines from 13 to 29 initialises the correct blocking behavior based on the arguments `blockOnAll` and `fn`. On Line 30 the blocking behavior is added to the global `Collection` of blocking behaviors across all transactions that is used to unblock the correct transactions when a transaction commits.

The implementation uses a `Set` backed by a `ConcurrentHashMap` as an instance implementing the `Collection` interface, this guarantees efficient thread-safe addition and removal of `STMBlockingBehavior`s from the `Collection`. Using a `Set` however does not guarantee fairness when iterating through the entries of the `Collection` this could lead to starvation of blocked threads. Alternatively a `ConcurrentLinkedQueue` could be used as it also implements the `Collection` interface, `ConcurrentLinkedQueue` guarantees that the oldest entry is handled first and allows thread safe removal of any elements in the queue.

Last the transaction is forced to retry by throwing the instance of `RetryEx` in `retryex` on Line 31. The act of blocking and unblocking based on a `STMBlockingBehavior` is implemented as part of the `run` method on `LockingTransaction`, these changes to the flow are shown at the end of the section in Figure 3.4

```
1 if (this.blockingBehavior != null) {
2   this.blockingBehavior.await();
3   LockingTransaction.blockingBehaviors.remove(this.blockingBehavior);
4   this.blockingBehavior = null;
5 }
6 gets.clear();
```

Listing 3.13: If the transaction has a blocking behavior then it must wait for it to resolve.

In Listing 3.13 code from `LockingTransaction` is shown. The code is the first part of the transaction's `run` method and checks if the transaction

has a blocking behavior set on Line 1. If so, the transaction is on Line 2 forced to block until the `CountDownLatch` in the blocking behavior is counted down to zero where it will unblock. After the transaction has been unblocked it will remove its blocking behavior from the global set of blocking behaviors across all transactions and then set the blocking behavior of this transaction to `null` on Line 3 and 4. Before executing the transaction the set of dereferenced `Ref`s will be cleared on Line 6.

```
1 for (STMBlockingBehavior blockingBehavior :
2       LockingTransaction.blockingBehaviors) {
3   blockingBehavior.handleChanged();
4 }
```

Listing 3.14: Notify all blocking behaviors in other transactions with which `Ref`s have been written to by the committed transaction

When the transaction has committed it notifies all blocking behaviors in the global set of blocking behaviors that `Ref`s was written to by calling `handleChanged` on each `STMBlockingBehavior` on Line 3 in Listing 3.14.

Iterating through the `Collection` and unblocking `STMBlockingBehavior`s is purposely not done atomically, in order to avoid unnecessary delay when a thread is unblocked. This is important for instances of the classes `STMBlockingBehaviorFnAny` and `STMBlockingBehaviorFnAll` as they execute a user defined function to test if the should unblock, this function can contain references to global state that might be changed between each call to `handleChanged`.

```
1  void doOrElse(ArrayList<IFn> fns) {
2
3    this.orElseRunning = true;
4    for (IFn fn : fns) {
5      try {
6        fn.invoke();
7        return;
8      } catch (RetryEx ex) {
9        // We ignore the exception to allow the next function to execute
10     }
11   }
12   this.orElseRunning = false;
13   throw retryex;
14 }
```

Listing 3.15: The `doOrElse` Java method in the transaction `LockingTransaction` class

The `doOrElse` method, seen in Listing 3.15, takes a single argument `fns` that is an `ArrayList` of `IFn` instances. An object that implements the interface `IFn` is a function in Clojure. The methods starts by setting the variable `orElseRunning` to true, forcing calls to `blockAndBail` to simply throw a `RetryEx` exception instead of blocking the transaction, as the transaction does not need to wait for a resource to be available when `or-else` is used.

The method then iterates through the functions given as `fns`, this is shown on Line 4 where each function at hand is executed using the `invoke` method on Line 6. If the function is executed without throwing a `RetryEx` exception then `doOrElse` will return. Otherwise the `RetryEx` exception will be caught and ignored which will allow the next function in `fns` to be executed. If no functions in `fns` is able to execute without throwing a `RetryEx` exception then `doOrElse` will throw the `RetryEx` instance in `retryex` to force the transaction to retry.

```
1 void abort() throws AbortException{
2   EventManager.stmNotify(LockingTransaction.ONABORTKEYWORD, null);
3   stop(KILLED);
4   throw new AbortException();
5 }
```

Listing 3.16: The `abort` Java method in the transaction `LockingTransaction` class

The `abort` method, seen in Listing 3.16, takes no arguments and terminates the currently running transaction. When a transaction terminates it needs to trigger the event handling system to execute the functions waiting to be notified on the `LockingTransaction.ONABORTKEYWORD` keyword on Line 2. Stopping a transaction in Clojure is done by calling the `stop` method on Line 3 with the constant `KILLED`. Finally the method throws an `AbortException` which forces the transaction to not execute again after the transaction have ended.

The changes to the flow of the `run` method on `LockingTransaction` due to the addition of transaction control can be seen in Figure 3.4. If the transaction contains a blocking behavior it should be blocked, this is checked in the first part of `run`. When the transaction has committed, other blocked transactions are notified with the changes of the committed transaction and possibly unblocked. If `terminate` is executed the `AbortException` is caught in the added `catch` block.

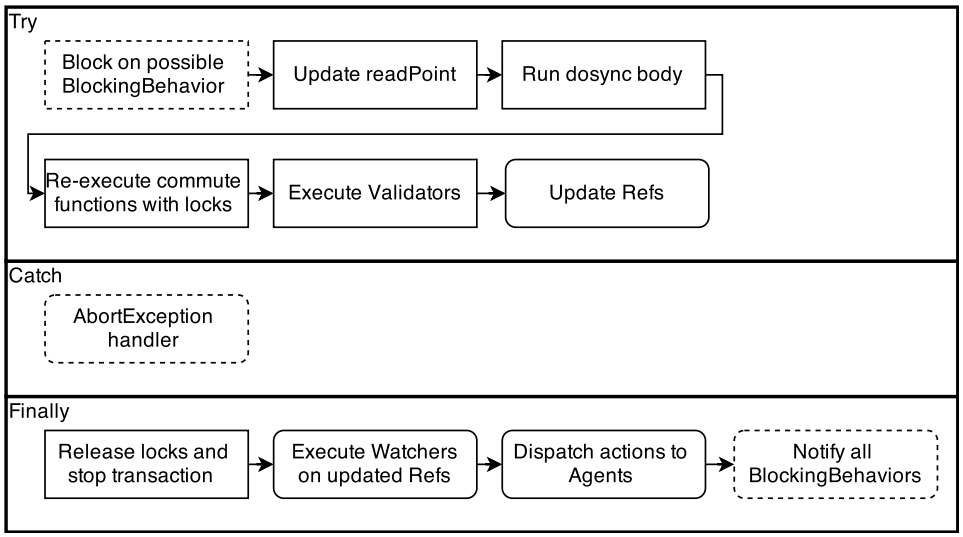

Figure 3.4: The major parts of the `run` method with the addition of transaction control squares show parts that can be re-executed, while rounded squares cannot, dashed lines indicate the added parts

3.3 Summary

In this chapter the implementation of a generic event handling system as well as constructs for transaction control into Clojure has been presented. This altered version of Clojure will be from here on be referred to as dptClojure.

This is done to easily differentiate between the two versions in the following sections where Clojure and dptClojure will be evaluated and compared to each other.

CHAPTER 4

Evaluation

An evaluation of our implementation seen in Chapter 3, will be described in this chapter. The event handling system described in Section 3.1, will be evaluated based on functionality in Section 4.1 by presenting cases where the added constructs of the event handling system allow for operations in dptClojure that was not possible in Clojure's STM implementation as the system have no equivalent in Clojure. The transactional control constructs described in Section 3.2, will be evaluated both on functionality and usability in Section 4.2 by first presenting cases where the added constructs allow operations not possible in Clojure's STM implementation. This evaluation will continue with a usability evaluation of the `retry` constructs based on the Santa Claus problem [23], as Clojure already contains constructs that allow for comparable functionality the benefits of dptClojure were not trivial to see. This will be followed by a summary in Section 4.3 that also evaluates the abstract concurrency models underlying Clojure's STM implementation by comparing the difference in this model between Clojure and dptClojure.

4.1 Event Handling System

The event handling system introduces the constructs described in Section 3.1. These constructs allow the developer to create events, to trigger events to notify listeners, and to listen for created events and events built into Clojure's STM implementation. By listening for events triggered by the STM implementation itself, it is possible to add code which will be executed when a transaction either commits, after it commits or when a transaction aborts.

Because the event handling system provide the same functionality as the experimental designs summarised in Section 2.5, will we use the same examples to showcase how the event handling system enables the use of side-effects in transactions. The first example prints the value of a `Ref` and before it tries to update its value which may force the transaction to abort. The second example updates the contents of an `ArrayList` and then writes a line to a log, both operations might make the transaction abort. The third

example inserts rows into a database and add the keys of the rows to a list of keys.

The three examples presented in Section 2.5.1, are implemented here using the event handling system instead of the specialised functions used in the experiments. They show how it is possible to use the event handling system in dptClojure to execute side-effects inside transactions, using STM as the synchronisation mechanism for mutable data structures and external effects, functionality not possible in a transactional safe way in Clojure's STM implementation.

```
1 (def counter-ref (ref 0))
2
3 (dosync
4   (after-commit (println (deref counter-ref)))
5   (if (< (deref counter-ref 10))
6     (alter counter-ref inc)
7     (ref-set counter-ref 0)))
```

Listing 4.1: Use of the event handling system for printing to the standard output stream after the transaction has committed

In Listing 4.1 the example of printing the value of the variable `counter-ref` is written using the `after-commit` function from the event handling system, `after-commit` is used as the value of the `Ref` is extracted, and there is no need to ensure that the `Ref` does not change between the call to `alter` and the value `Ref` being printed.

```
1 (def arraylist-ref (ref (java.util.ArrayList.)))
2 (def log-ref (ref (writer  "log.txt" :append true)))
3
4 (dosync
5   (on-commit (lock-refs alter
6     (alter arraylist-ref .add arraylist-ref 0)
7     (alter log-ref .write "Added 0 to arraylist-ref"))))
```

Listing 4.2: Use of the event handling system for modifying `ArrayList` and logs

The second example is shown as Listing 4.2 where the use of the `on-commit` function can be seen together with the `lock-refs` function. The function passed to `on-commit` is executed as part of the transaction, making it necessary to use `lock-refs` to ensure that the necessary locks are acquired. The function `lock-refs` is implemented as a separate feature to ensure the developer has full control of when automatic locking of `Refs` are performed.

```
1 (def keys-ref (ref []))
2 (def rows-ref (ref []))
3
4 (dosync
5   (on-commit (lock-refs alter
6     (let [row (first (deref rows-ref))
7           next-key (database-insert row)]
8             (alter keys-ref conj next-key)
9             (alter rows-ref rest)))))
```

Listing 4.3: Insert rows into a database and return the key

The third example is shown in Listing 4.3, similar to the second example the `on-commit` function is used together with the `lock-refs` function. Locking the `Ref`s ensure that the database insert operation is only executed if we can guarantee write access to the locks needed, otherwise the transaction will abort before the row is inserted.

The functionality of listening for events in the STM implementation is not possible without these added constructs and gives the developer a way to use side-effects in a transactional safe manner. Furthermore, the introduction of a generic event handling system enables the developer to write event listeners for events both in the Clojure runtime and in Clojure outside of the runtime created by the developer. The implementation allows a context to be provided for an event listener. As an example, the `on-commit` event provides the set of `Ref`s that the transaction had written to, this context is accessible through the function `context` in the scope of an event listener.

The last special event provided by the event handling system is `on-abort`, which is not shown with an example since the three examples shown above were simpler to implement using `after-commit` and `on-commit`. The `on-abort` event allows code to be executed when a transaction aborts, this functionality is currently not present in Clojure's STM implementation. The `on-abort` event makes it possible to roll-back side-effects on for example a mutable data structure.

The three examples all use the special event functions directly inside of the transactions, but they could also be embedded into data structures to make the data structures STM aware. For example by making an extended version of the Java data structure `ArrayList` that undoes operations if the transaction aborts using `on-abort` or waits with operations until commit time with `on-commit`.

The primary reason for implementing the event handling system was to allow Clojure's STM implementation to provide built-in events, but it also provides a platform for Clojure developers to use events in their own

programs, a functionality not present in Clojure. A usability evaluation of the event handling system has not been performed since the system has no counterpart in Clojure, and only adds additional functionality.

4.2 Transactional Control

The transactional control constructs described in Section 3.2, allows the developer to control the behaviour of transactions. The methods `or-else` and `terminate` will be evaluated in Section 4.2.1, while `retry` will be evaluated in Section 4.2.2.

4.2.1 Or-else and Terminate

The function `or-else` allows the developer to specify multiple functions to execute sequentially and return the value of the first function that completed without trying to abort the transaction. If no functions are able to complete execution without the transaction trying to abort, the transaction aborts. The `terminate` function allows the developer to abort a transaction without re-executing it, letting the executing thread continue instead of re-executing the transaction. An example that uses both functions can be seen in in Listing 4.4.

```
1 (dosync
2   (execute-db-operation
3     (or-else
4       (alter dbc-one identity)
5       (alter dbc-two identity)
6       (alter dbc-three identity)
7       (terminate))))
```

Listing 4.4: Or-else and terminate example

In this example a thread tries to execute an operation on a database if a database is available, otherwise the thread continues its execution. The method `or-else` in Line 3 is used to `alter` one of three database connections. If no database connections could be acquired it terminates the transaction by calling `terminate` in Line 7 and the thread will continue. If a database connection is acquired its value is returned to `execute-db-operation`. As the function `or-else` introduces functionality that do not exist in Clojure's STM implementation, it will be evaluated as no constructs in Clojure that it can be compared with.

The `terminate` function can be emulated with the use of exceptions as shown in Listing 4.5. Here a `try/catch` block was put around the `dosync` block in order to catch an exception named `TerminateException`. This

construction allows code inside the transaction to throw this exception which then terminates the transaction, Clojure aborts transactions if an exception is thrown, and the use of a specific exception ensures that another exception does not trigger this behaviour.

```
(dosync
  (execute-db-operation
    (or-else
      (alter dbc-one identity)
      (alter dbc-two identity)
      (alter dbc-three identity)
      (terminate))))
```

```
(try
  (dosync
    (execute-db-operation
      (or-else
        (alter dbc-one identity)
        (alter dbc-two identity)
        (alter dbc-three identity)
        (throw TerminateException))))
  (catch TerminateException te))
```

Listing 4.5: Or-else using dptClojure (left), and Clojure (right)

Based on Listing 4.5, it is trivial to see that the use of `terminate` simplifies the operation of forcing a transaction to abort without allowing it to retry. This is due to `terminate` removing the need for an exception handler around the `dosync` block, and the need for defining a unique exception to prevent conflict with other Java exceptions.

4.2.2 Retry

The methods `retry` and `retry-all` allows the developer to specify that a transaction must abort and first re-execute on change in either one or all of the specified `Ref` instances.

```
(def updated-ref (ref 0))
(def static-ref (ref 10))

(dosync
  (if (== (deref updated-ref) (deref static-ref))
    (ref-set updated-ref 0)
    (retry updated-ref)))
```

Listing 4.6: Retry example

An example of using `retry` is shown in Listing 4.6. Here `updated-ref` is set to zero when it reaches ten. The incremental changes to `updated-ref` are done by other transactions. `retry` is used in Line 7, to block the transaction until `updated-ref` is modified. Since `static-ref` in this scenario is never updated outside the shown transaction, waiting on both `Ref`s will result in a deadlock. Therefore we only call `retry` with `updated-ref`.

The `retry` function provides functionality that can be replicated by a developer in the current STM implementation of Clojure. The transac-

tion can be terminated with an exception like and then the executed again through recursion. The synchronisation aspects of `retry` and `retry-all` can be achieved using constructs from `java.util.Concurrency`, which contains synchronisation constructs such as semaphores and synchronised data structures.

As emulating `retry` and `retry-all` is more complex and requires the use of Java synchronisation constructs, it is not trivial to see that it provides a simpler way for performing synchronisation of threads. Instead we perform a usability evaluation, described in Section 4.2.2 to evaluate which of the two synchronisation methods is simplest to use.

Usability evaluation

The usability evaluation consists of two implementations of the Santa Claus Problem [23], as it is a well known problem for working with concurrency and synchronisation. The Santa Claus problem is seen as a representation of a real world concurrent program because of the amount of synchronisation needed for a correct implementation. Furthermore we have developed solution for the problem before in our 9^{th} semester project [6]. The Santa Claus Problem is described in Appendix D. The first implementation will use dptClojure's STM implementation and the second implementation will be created using only Clojure's constructs.

The two implementations will be compared based on the following metrics.

Lines of code (LOC) How many LOC the implementation consists of while ignoring empty lines and comments.

Development time How much time was spend on creating the implementation.

LOC will be used as a measure as it will make the resulting code comparable to existing research [24, 25, 26] even tough LOC in itself is not that informative it does give an indication of how much code is needed to use the constructs of the language. We have chosen to ignore comments and empty lines as LOC because they are not a direct part of the program because there is no requirements to their use and its entirely up to the developer where they are used.

A short program on its own is not guaranteed to have been easy to write which is why development time is also included because it shows how efficiently the developer can create solutions with the constructs available. A combination of LOC and development time is therefore a good indicator on programming languages usability because a language that allow for short development time is efficient to develop, and a small program gives less

to reason about for modification and that the language support expressive constructs. [26]

However it have been shown that metrics alone is unreliable in determining how challenging language constructs are to use [25]. Therefore a subjective discussion about disadvantages and advantages of the two implementations will help give a more complete picture of the usability of the added language constructs.

To ensure a fair comparison between the results of the two implementations in regards to LOC and development time the two implementations were developed by the same developer. The developer already knew the Santa Claus Problem and had implemented solutions for it before in Clojure as well as other functional programming languages. The developer had experience using the constructs added in this project as well as using similar constructs found in Haskell.

The implementation with the added constructs was developed first to not give this implementation any upper hand compared to the implementation of the solution without the added constructs. Additionally the developer had earlier developed a solution for the Santa Claus problem using locks and semaphores giving that implementation somewhat of an advantage in terms of development time since the developer had experience with writing the exact same solution.

Usability evaluation outcome

The results of the usability evaluation are presented in this section, with both metrics and the subjective discussion of the implementations with direct comparison between the two implementations.

As explained the implementations were evaluated using two metrics namely, LOC and development time. The results for development time are rounded to the nearest quoter to make comparison simpler by removing insignificant detail, are shown below in Table 4.1.

	Lines of Code	Development time
With extensions	56	2 hours
Without extensions	74	4 hours
Percentage Difference	24.3 %	50 %

Table 4.1: The results of the usability evaluation based on the two chosen metrics, lines of code and development time rounded to the nearest quoter

The implementation of dptClojure performed better both in terms of LOC and in terms of development time compared to Clojure. The reduced number of LOC is the result of the addition of the constructs for `retry` in dptClojure, instead of throwing exceptions and synchronising with a `Semaphore` and `CyclicBarrier` .

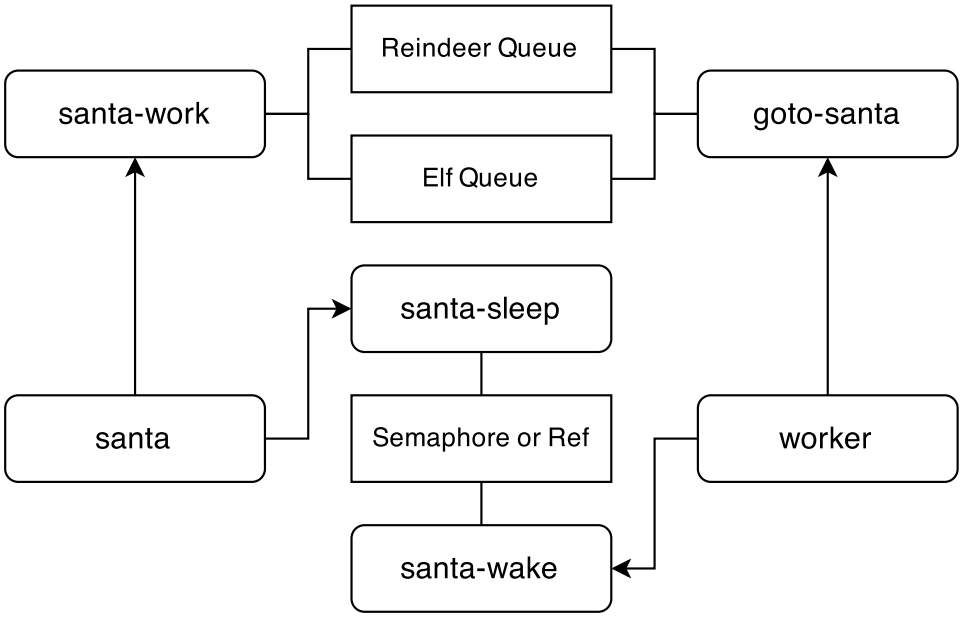

Figure 4.1: Architecture of Santa Claus Problem implementation. Arrows indicate function calls. Lines indicate data structure access.

The two implementations are quite similar but differ in how the concurrency is done. An architectural overview of the implementations can be seen in Figure 4.1. The implementation consists of a single thread, santa, that infinitely executes the `santa` function. Nine threads representing the reindeers executes the `worker` function and waits in the `reindeer-queue`, while thirty threads representing the elfs executes the `worker` function and three at a time waits in the `elf-queue`. With the added constructs there is no need for semaphores or barriers for ensuring the correct behaviour, instead the `retry` functionality is used. Below selected parts of the implementations are compared to each other to show they are similar in behaviour but differ in syntax as a result of the added constructs.

```
1 (defn worker [wid queue-ref max-sleep max-queue sleep-ref]
2   (sleep-random-interval max-sleep)
3   (dosync
4     (let [queue-length (count (deref queue-ref))]
5       (if (< queue-length max-queue) (goto-santa wid queue-ref) (retry))
6       (when (== (inc queue-length) max-queue)
7         (santa-wake sleep-ref))))
8   (recur wid queue-ref max-sleep max-queue sleep-ref))
```

Listing 4.7: The `worker` function with extensions

In Listing 4.7 the `worker` function of the implementation with the added constructs is shown. In Line 1 the function takes five arguments; `wid` is the worker ID, `queue-ref` is the `Ref` for the appropriate queue based on the worker type, `max-sleep` defines the maximum number of seconds the worker is allowed to sleep, this is also defined by the worker type. `max-queue` defines the maximum number of elements allowed on the queue in `queue-ref` before waking up Santa. Finally the `sleep-ref` argument defines the `Ref` holding the state whether or not Santa is asleep. In Line 2 the function will sleep a random number of seconds with a maximum defined by `max-sleep`. The transaction used by the worker is started in Line 3 and the length of the appropriate queue for the worker type is bound to `queue-length` in Line 4. In Line 5, based on the length of the queue the worker will either call the function `goto-santa` if the queue is not yet full or call `retry` if the queue is full. By calling `retry` the worker will block until the queue altered, otherwise the function `goto-santa` is called which will add the worker to the queue and block the thread using `retry` until the queue is cleared by Santa if the worker is already in the queue. If the queue fills up as a result of the worker calling `goto-santa`, the worker will call the function `santa-wake` in Line 7. The `santa-wake` function will wake up Santa by altering the `sleep-ref Ref` if he is asleep, otherwise worker will call `retry` and block until either `max-queue` or `sleep-ref` is altered. Finally the function will call itself recursively in Line 8.

```
1  (defn worker [wid queue-ref max-sleep max-queue sleep-sem
2                worker-sem worker-barrier]
3    (sleep-random-interval max-sleep)
4    (try
5      (dosync
6        (let [queue-length (count (deref queue-ref))]
7          (if (< queue-length max-queue)
8            (goto-santa wid queue-ref)
9            (throw (java.lang.IllegalStateException.)))))
10     (catch java.lang.IllegalStateException ise
11       (.acquire worker-sem 1)
12       (.await worker-barrier)))
13   (recur wid queue-ref max-sleep max-queue sleep-sem
14    worker-sem worker-barrier))
```

Listing 4.8: The `worker` function without extensions

In Listing 4.8 the implementation of the same `worker` function implemented using Clojure is shown. The function takes more arguments compared to the implementation with the added constructs because of the need for additional concurrency constructs to achieve the same behaviour. Instead of using a `Ref` for the state of whether or not Santa is sleeping a Java

`Semaphore` is used. A `Semaphore` is used for blocking the workers together with a Java `CyclicBarrier`. Both are needed to prevent a race condition, where one worker decrements the `Semaphore` multiple times leading to a deadlock.

```
1 (defn start-workers [queue-ref max-sleep max-queue sleep-sem
2                      worker-sem worker-barrier number]
3   (add-watch queue-ref :key (fn [_key _ref old-state new-state]
4                               (when (== (count new-state) max-queue)
5                                 (santa-wake sleep-sem))))
6   (doseq [wid (range 0 number)]
7     (.start (Thread.
8               (fn [] (worker wid queue-ref max-sleep max-queue sleep-sem
9                              worker-sem worker-barrier))))))
```

Listing 4.9: The interfaces for the event handling system

In the Clojure implementation watchers are used to call `santa-wake` when the size of one of the queues are altered and maybe filled. The addition of the watchers are done in the function `start-workers` which is a helper function for starting all workers. The `start-workers` function of the implementation without the added constructs can be seen in Listing 4.9, the function exists in both implementation to start the worker threads, in the Clojure version a watcher is also set to execute code when a transaction commits. Setting the watcher can be seen on Line 3 where a watcher is set for the `Ref queue-ref`, which is the queue for either reindeers or elfs depending on what set of threads are being started, the function executed by the watcher checks on Line 4 the size of the queue and if the queue is full `santa-wake` will be called in Line 5.

Implementing the overall structure of the two version proved to be very similar, and the time spent for each was nearly the same. However the time spent ensuring that the Clojure implementation did not cause a deadlock were higher. Primarily correcting the race condition caused by the lack of a `Barrier`, increased the development time for the Clojure version. In general reasoning about when threads reached a specific place in the code to interact with a `Semaphore` or `Barrier` and the current state of each thread proved to be a challenge, which required the assistance of a debugger to ensure that the implementation was correct. Synchronisation in terms of data proved much simpler to comprehend, since the synchronisation constructs were added to ensure that data was in a specific state before another operation was executed, which could be expressed directly using `retry`.

4.3 Summary

The underlying concurrency model of Clojure's and dptClojure's STM implementation is evaluated based on the approach taken in [27]. By model we mean the functionality provided by each implementation, without concern for any implementation details. The approach evaluates the model by looking at the following different characteristics and is useful to reason about how the overall behaviour of the underlying concurrency model has changed by implementing these new constructs.

Implicit or Explicit Concurrency A model provides implicit concurrency if concurrency is provided as part of the model itself and requires no additional work by the developer. A model provides explicit concurrency if it requires the developer to manually add additional constructs such as locks and semaphores.

Fault Restricted or Expressive Model A fault restricted models prevents the developer from making errors by only providing high level access to the constructs of the model. An expressive model gives low level access to each part of the model, but expects the developer to ensure each part is used correct.

Pessimistic or Optimistic Model A pessimistic model only allows the amount of concurrency that can be performed without conflicting reads or writes. An optimistic model maximizes the amount of concurrency, at the cost of requiring a method for handling conflicts.

Automatic or Manual Parallelisation An automatic model uses the compiler or runtime to automate the parallelisation of the code with no additional work by the developer. A manual model requires the developer to add additional constructs and to create additional threads to execute code concurrently.

The two models are weighted on a five step scale that spans from one side of the contrasting concurrency criteria to the other. By the addition of the constructs described in this project, the concurrency of the underlying model has become more explicit and expressive as can be seen in Figure 4.2. The evaluation of the concurrency model of Clojure before the addition of constructs was done in [6] were the individual scorings are explained.

As described in Section 4.1 the addition of the generic event handling system enables the developer to listen for events in for example the STM implementation. This gives the developer a lower level of access to the STM implementation, thereby increasing the expressiveness of the model. The introduction of functions for automatic locking of `Ref`s if needed also increases the degree of explicit concurrency in an indirect way.

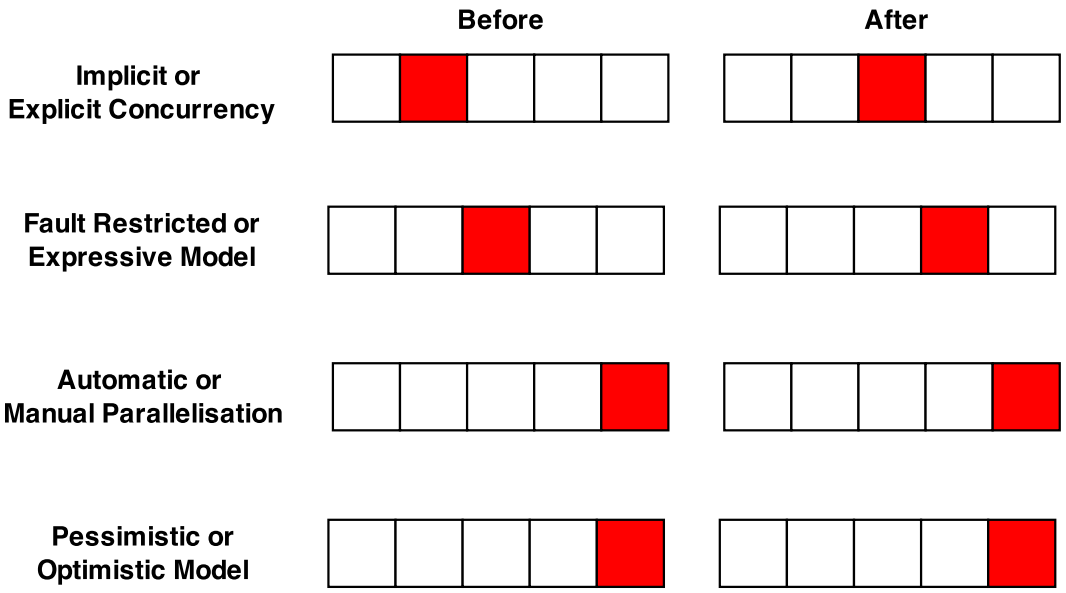

Figure 4.2: Evaluation of the model underlaying Clojure's and dptClojures's STM implementation, the evaluation model was presented in [27]

The addition of the constructs for transactional control described in Section 4.2 makes the underlying model more expressive by allowing the developer to manually control transactions instead of controlling transactions by for example altering `Ref`s. Besides forcing a transaction to abort and block until the alteration of one or more `Ref`s with different semantics it is now possible to terminate a transaction explicitly, and it is possible to execute the first possible of multiple functions.

Both of the additions adds some explicitness and some expressiveness to the model, but are described in the way above to ease the explanation of why the model of dptClojure is different compared to the model for Clojure.

CHAPTER 5

Reflection

This chapter will discuss and reflect on the choices made in this project. First the effect of adding constructs for handling side-effects in transactions and constructs for transaction control will be discussed. Hereafter the usability evaluation will be discussed followed by considerations about how a performance evaluation of dptClojure could be done. The chapter is closed of with a discussion of the project approach as well as some considerations for the implementation of dptClojure.

5.1 Effect of the added constructs

dptClojure extends Clojure with support for transactionally safe side-effects and transactional control. Allowing side-effects in transactions make STM similar in capabilities to locks, as a transaction can synchronise both access and external effects. We argue that this makes using transactions a simpler task for developers already knowledgeable about lock based concurrency, as the same capabilities are provided by the synchronisation mechanism, with only the underlying semantics changed. A negative aspect of this is that developers might use STM as a direct composable substitute for locks. Which can lead to waste of CPU cycles if a developer expects other threads to wait until a thread is finished with a critical section as this is not the case with STM.

The addition of transactional control introduces a completely alternative way to think about synchronisation, as the synchronisation is now based on changes to transactional data instead of a position in the code. While the evaluation shows that the use of transactional control can lead to smaller programs and shorter development times, it does require a change in how the developer thinks about performing synchronisation which could be difficult if the developer already is experienced in developing programs using the classic synchronisation constructs.

5.2 Usability Evaluation

The usability evaluation is performed using a single test person with extensive knowledge about both the language constructs and how to use them for synchronisation. This prevents having to learn to use the constructs from influencing the evaluation, and lets us evaluate how the constructs perform for building actual applications. However the evaluation does not give an indication of how much time is needed to acquire such familiarity with the constructs, and if the time spent learning to use this alternative method of synchronisation is acceptable compared to the decrease in development time, especially if a developer already is familiar with synchronisation using locks. Also as the usability evaluation is based on only one developer and one test problem, the results could, when compared to other evaluations be an seen as an outlier.

5.3 Performance Evaluation

We decided that performance was of secondary priority in the project, and that the addition of functionality was in focus and not its optimisation. However if one were to perform a performance evaluation of the implementation, multiple aspects would have to be evaluated.

- Does the additional functionality impact performance for programs where it is not used?
- Does the use of the added functionality add overhead compared to the alternatives?
- Does the added overhead make the added functionality unaffordable in terms of running time

The first question could be answered by implementing two test programs; one synchronising a large set of transactional values in a few transactions and the second synchronising a very small test of transactional values in a lot of transactions and then comparing if there are a performance difference between Clojure and dptClojure. The second question would require the same programs with side-effects and need for synchronisation between threads to be implemented. dptClojure should only use STM for both synchronising data, side-effects and threads, while the Clojure version would have to use a combination of STM and components from `java.util.Concurrency` to implement equivalent functionality.

5.4 Approach

The approach of the project has been different than our other projects as it succeeds the work from our 9th semester project [6]. The knowledge gained from our previous project [6] made it possible for us to start experimenting with solutions to the problems of the project, without the need for familiarising us with the domain beforehand. We therefore decided to take an experimental approach to the problem in order to get an understanding of Clojure's internal implementation.

These experiments and their implementation helped give an overview of Clojure's runtime and STM implementation. Instead of starting the project with an extensive literature study of the subject we were able to gain an overview of the problems and how possible solutions relate to each other and to Clojure's STM implementation. This overview were easier to gain by looking at the connection between each solution's implementation compared to only reading the literature which only explains the concepts of the solutions.

Starting with an in-depth analysis of literature about methods for combining side-effects and STM would probably have removed the need for the experimental design and implementation phase, as the method developed as part of this could be found in the literature. However this would probably also have made the time spent analysing Clojure's implementation and `LockingTransaction` much longer, and made the final solution implemented of lesser quality, as the experimental implementation helped to gain knowledge about subtleties in Clojure's STM implementation.

5.5 Implementation

The use of `context` in the event handling system is limited in the events added for `on-commit`, `after-commit` and `on-abort`. Context is only provided for the event listeners registered for the `on-commit` event, the context provided is a Clojure `PersistentHashSet` containing all the `Ref`s is used in the transaction. By providing a `PersistentHashSet` the system is guaranteed that the `Ref`s are not altered by the event listener function. However it would be possible to provide context to the event listener function that enables behaviour like the one presented in [11] where context can be used to change the behaviour of a transaction inside an event listener function. In general the use of context in event listener functions show a great potential for giving the developer more power over and information about the STM implementation. Because of time constraints and the relative simplicity of the concept of context we have not added context for more events and leave this to be explored in future work.

We have identified a problem with `retry` when the `Ref` being blocked on

by the transaction is not updated by another transaction, then the transaction being blocked will never be unblocked. A solution to the problem would be to let the developer specify a maximum time a transaction may be blocked by a specific blocking behavior. We have however decided not to add this functionality as we see the problem as a result of missing updates to the `Ref` being blocked on and thereby something up to the developer to fix.

The function `dismiss` is used to remove an event listener function from a specific event and can by choice not be used to remove event listener functions added to STM events. Furthermore we have chosen not to implement an STM version of `dismiss` as we encourage small transactions. Small transactions have a smaller possibility to conflict with other transactions as their running time as well as the number of locks needed to commit are smaller, therefore we encourage small transactions to avoid conflicts [9]. With smaller transactions the need for both adding and then removing event listener functions as very little therefore we have chosen not to implement this functionality.

The implementation of the constructs in dptClojure have been tested using unit testing. The tests have been written to fully cover the code with both a single and multiple transactions. Besides unit testing we have also been using pair programming as well code reviews to ensure the implementation contains as few bugs as possible from the start.

CHAPTER 6

Conclusion

STM in Clojure provides a composable and deadlock free alternative to locks for synchronising concurrent programs, but does not provide functionality for handling side-effects, and allows no manual control over each transaction for synchronisation of threads. The general problems with Clojure's lack of this functionality was determined through an extensive evaluation of both performance and usability of parallel functional programming languages and their concurrency models in our earlier work [6].

In this project we have implemented constructs allowing the use of side-effects and transaction control in STM transactions into the programming language Clojure, the extended language we have dubbed dptClojure. The constructs were developed based on an extensive analysis of the Clojure runtime and STM implementation, and multiple experiments about how methods for handling side-effects in transactions could be performed.

The experimental designs were implemented in Clojure's STM implementation, together with constructs for transaction control inspired from Haskell, to understand how such functionality could be enabled in Clojure's STM implementation. Based on the experimental design and their implementation a unified event handling system was developed and implemented in Clojure with minimal changes to the existing STM implementation, enabling the use of side-effects in transactions by ensuring the side-effects are executed only if a transaction is in a specific phase. Constructs for transaction control were added and extending those inspired by Haskell, allowing the developer to not only block a transaction based on transactional data, but also specify a function for determining the state of said data and check if the transaction should unblock.

We evaluated the capabilities of the event handling system using the three minimal examples, and we showed the event handling system could be used to solve all problems represented by the examples. Transaction control was evaluated by demonstrating the effect of the added functionality in Clojure as the concepts and what they solve were known from Haskell. For the constructs that provided alternatives to existing Clojure functionality the constructs were evaluated using a usability evaluation based on the

Santa Claus problem, with two metrics and a subjective discussion. This evaluation showed a decrease both in terms of Lines Of Code (LOC) and in terms of development time, and subjectively the added constructs helped to create a simpler implementation.

To summarise the project we revisit the problem statement found in Section 1.2, describing the questions of the project and defining tasks we need to do to answer the questions.

How does the STM implementation of Clojure interact with the rest of the language and implementation?

In Chapter 2 Clojure's runtime as well as its STM implementation has been described in detail. This in-depth description gives an overview of how Clojure interacts with the STM implementation through the language's functions, constructs and types.

How does Clojure handle the use of side-effects in transactions and what are its limitations?

The way to use side-effects in a transactional safe manner in Clojure is through the use of `agent`s. `agent`s have multiple limitations since the execution of `agent`s are asynchronous and the return value of side-effects are therefore not accessible inside the transaction. These limitations are described in greater detail in Section 2.3 and 2.4.

How is it possible to introduce the use of side-effects in transactions in Clojure's STM implementation?

Through an exploration of Clojure we have experimented with multiple solutions to introduce the use of side-effects in transactions in Clojure's STM implementation. Together with this exploration we have studied the literature of the subject and generalised our experiments to a event handling system for introducing the use of side-effects in transactions in Clojure's STM implementation. A summary of the these explorations can be found in Section 2.5.

How is it possible to introduce the use of transaction control in Clojure's STM implementation?

We implemented constructs for transaction control into Clojure inspired from Haskell, first as a part of the experiments early in the project. This experimental implementation was then extended in dptClojure. Transactional control was introduced by the addition of multiple constructs into Clojure and blocking behaviors into the STM implementation. A description of the implementation can be found in Section 3.2.

How does the introduction of these concepts into Clojure's STM implementation affect the usability of the STM implementation?

The usability of the addition of transaction control is evaluated in Section 4.2. Only the addition of transaction control was usability evaluated since transaction control is possible in Clojure but only by using abuse existing constructs in Clojure. We evaluated the usability by implementing the Santa Claus problem and two metrics, LOC and development time. The usability evaluation found a decrease both in terms of LOC and in terms of development time, furthermore a subjective evaluation of the implementations found that the added constructs for transaction control helped create simpler implementation of the Santa Claus problem. The effect of adding the event handling system in terms of usability has not been evaluated in this project as the event handling system introduces new functionality, therefore it is impossible to compare the new constructs to any existing ones in Clojure.

Concluding Remarks

By answering the questions stated in the problem statement we have created an extended version of Clojure called dptClojure which contains new constructs for handling the use of side-effects in transactions in Clojure's STM implementation as well as constructs for more explicit transaction control.

6.1 Future Work

This section will discuss possibilities for future avenues of research in relation to dptClojure.

Twilight STM The event handling system of dptClojure introduced the use of contexts in event handling functions. A context is information provided by the thread notifying an event. This functionality is currently only used to inform the `on-commit` event about which `Ref`s that can be changed while being guaranteed that the transaction will not abort. However we see the possibility of adding more detailed control over the transaction, similar to what is provided in Twilight STM [11] by the use of contexts. This could be achieved by creating an API for interacting with the transaction through an interface into `LockingTransaction` made accessible as an object instance through the context provided to the `on-commit` and `on-abort` events.

Evaluation The evaluation of dptClojure could be extended with a more detailed evaluation of the usability of dptClojure as mentioned in the discussion in Section 5. A larger usability evaluation using multiple examples and more developers could be conducted to evaluate whether the transaction

control constructs are preferable in general and not just for developers with extensive knowledge about the constructs and the Santa Claus problem.

Additionally an evaluation of how much overhead and how the performance characteristics change when comparing dptClojure to Clojure's STM implementation could be useful to see if dptClojure is be a relevant alternative to Clojure. How a performance evaluation could be performed is described in Section 5.3.

Bibliography

[1] Sutter, Herb. The free lunch is over: A fundamental turn toward concurrency in software. `http://www.gotw.ca/publications/concurrency-ddj.htm`.

[2] Lee, Edward A. The Problem with Threads. Technical Report UCB/EECS-2006-1, EECS Department, University of California, Berkeley, Jan 2006. The published version of this paper is in IEEE Computer 39(5):33-42, May 2006.

[3] TIOBE Software: Tiobe Index. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`, 2015. Accessed: 05-03-2015.

[4] Totoo, Prabhat and Deligiannis, Pantazis and Loidl, Hans-Wolfgang. Haskell vs. F# vs. scala: a high-level language features and parallelism support comparison. In *Proceedings of the 1st ACM SIGPLAN workshop on Functional high-performance computing*, pages 49–60. ACM, 2012.

[5] Clojure Concurrent Programming. `http://clojure.org/concurrent_programming`. Accessed: 05-03-2015.

[6] Jensen, Daniel Rune AND Jensen, Søren Kejser AND Jacobsen, Thomas Stig. Performance and Usability Evaluation of Concurrency in Modern Functional Programming Languages. Technical report, Department of Computer Science, Aalborg University, 2015. `https://github.com/eXeDK/dpt908e14/blob/master/Report.pdf`.

[7] Clojure.Org Macros. `http://clojure.org/macros`. Accessed: 04-05-2015.

[8] Clojure.Org Macro Characters. `http://clojure.org/reader#The%20Reader--Macro%20characters`. Accessed: 04-05-2015.

[9] Kalin, M. and Miller, D. Clojure for Number Crunching on Multicore Machines. *Computing in Science Engineering*, 14(6):12–23, Nov 2012.

[10] Baugh, Lee and Zilles, Craig. An analysis of I/O and syscalls in critical sections and their implications for transactional memory. In *Performance Analysis of Systems and software, 2008. ISPASS 2008. IEEE International Symposium on*, pages 54–62. IEEE, 2008.

[11] Bieniusa, Annette and Middelkoop, Arie and Thiemann, Peter. Actions in the Twilight: Concurrent irrevocable transactions and inconsistency repair (extended version). Technical report, Technical Report 257, Institut für Informatik, Universität Freiburg, 2010.

[12] Harris, Tim. Exceptions and side-effects in atomic blocks. *Science of Computer Programming*, 58(3):325–343, 2005.

[13] McDonald, Austen and Chung, JaeWoong and Carlstrom, Brian D and Minh, Chi Cao and Chafi, Hassan and Kozyrakis, Christos and Olukotun, Kunle. Architectural semantics for practical transactional memory. *ACM SIGARCH Computer Architecture News*, 34(2):53–65, 2006.

[14] Volos, Haris and Tack, Andres Jaan and Goyal, Neelam and Swift, Michael M and Welc, Adam. xCalls: safe I/O in memory transactions. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 247–260. ACM, 2009.

[15] Haskell Community. Thunk. `https://wiki.haskell.org/Thunk`.

[16] Welc, Adam and Saha, Bratin and Adl-Tabatabai, Ali-Reza. Irrevocable transactions and their applications. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 285–296. ACM, 2008.

[17] Blundell, Colin and Lewis, E Christopher and Martin, Milo. Unrestricted transactional memory: Supporting I/O and system calls within transactions. 2006.

[18] Spear, Michael and Michael, Maged and Scott, Michael. Inevitability mechanisms for software transactional memory. In *3rd ACM SIGPLAN Workshop on Transactional Computing, New York, NY, USA*, 2008.

[19] Olszewski, Marek and Cutler, Jeremy and Steffan, J Gregory. JudoSTM: A dynamic binary-rewriting approach to software transactional memory. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 365–375. IEEE Computer Society, 2007.

[20] Rossbach, Christopher J and Hofmann, Owen S and Porter, Donald E and Ramadan, Hany E and Aditya, Bhandari and Witchel, Emmett. TxLinux: Using and managing hardware transactional memory in an operating system. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 87–102. ACM, 2007.

[21] Harris, Tim and Marlow, Simon and Peyton-Jones, Simon and Herlihy, Maurice. Composable Memory Transactions. In *Proceedings of*

the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '05, pages 48–60, New York, NY, USA, 2005. ACM.

[22] Smaragdakis, Yannis and Kay, Anthony and Behrends, Reimer and Young, Michal. Transactions with isolation and cooperation. In *ACM SIGPLAN Notices*, volume 42, pages 191–210. ACM, 2007.

[23] Trono, John A. A new exercise in concurrency. *ACM SIGCSE Bulletin*, 26(3):8–10, 1994.

[24] Markstrum, Shane. Staking claims: a history of programming language design claims and evidence: a positional work in progress. In *Evaluation and Usability of Programming Languages and Tools*, page 7. ACM, 2010.

[25] Luff, Meredydd. Empirically investigating parallel programming paradigms: A null result. In *Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU)*, 2009.

[26] Nanz, Sebastian and West, Scott and Soares da Silveira, Kaue and Meyer, Bertrand. Benchmarking usability and performance of multicore languages. In *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*, pages 183–192. IEEE, 2013.

[27] Damborg, B. and Hansen, A.M. *A Study in Concurrency.* Aalborg University. Department of Computer Science, 2006.

APPENDIX A

Experimental Designs

This appendix will present experiments that were performed to explore and investigate Clojure's STM implementation. The experiments either focus on allowing the use of side-effects in transactions or introducing control of the transactions in order to avoid execution of side-effects. The sections A.1, A.2 and A.3 will focus on the use of side-effects in transactions whereas Section A.4 will focus on transaction control. Transaction control is a means to synchronise based on the state of data instead of through function side-effects. References to data structures must be stored inside `Ref`s and accessed via `alter`, `commute`, `ensure` or `ret-set` if the developer wishes to modify the data structures despite the reference itself never being modified. This allows Clojure's STM implementation to synchronise access to them between threads.

We created three examples to illustrate how the different side-effects are handled by the before mentioned experiments, the examples are documented in Section 2.5.1. These examples will be discussed in correlation with each experiment. Implementation details of these experiments have been documented in Appendix B.

A.1 Defer

Defer is an approach which postpones the execution of code with side-effects to take place when the transaction no longer can abort. We have performed two experiments from this category.

A.1.1 After-commit

The `dosync-ac` macro extends the `dosync` macro found in Clojure with a block of code to execute synchronously after the transaction has completed. This allows side-effects to be executed in the same manner as using an `agent` but synchronously and without having to take the `agent`'s current state into account. Implementation details of After-commit can be seen in Appendix B.1. The entire implementation is created using macros allowing it to be used as a library, without changes to the Clojure runtime.

```
1 (send a f & args)
```

Listing A.1: Clojure `agent` send example

Multiple design ideas for After-commit are discussed in Appendix C. We chose to implement the macros based on the structure of the existing `send` function in Clojure. The `send` function is used to send a function to an `agent` and can be seen in Listing A.1. The function `send` sends an agent `a` a function `f` with arguments `args`. This is very similar to what we want to achieve as we want to "send" functions with arguments to be executed after the transaction has committed. We will apply the same idea for the other experimental implementations of this project as it is most similar to Clojure's own functions in general. The implemented After-commit macro seen in Listing A.2.

```
1 (dosync-ac & body)
2
3 (ac & body)
4 (ac-fn func & args)
5 (ac-fn-set ref-set-map func & args)
```

Listing A.2: After-commit signatures

The solution consist of an extended version of the `dosync` macro which can be seen on Line 1 in Listing A.2 and is named `dosync-ac`. The `dosync-ac` macro has the same signature as the original `dosync` macro, however it can store functions and expressions for later execution. The other three macros shown in Listing A.2 all allows the developer to store code for executing synchronously after the transaction commits. The first macro `ac` takes a list of expressions as argument, it can be seen on Line 3. The second macro `ac-fn` takes as its first argument a function and a list of arguments as its second argument as seen in Line 4. The signature for the last macro named `ac-fn-set` can be found in Line 5 and like `ac-fn` it accepts a function and arguments but makes it easier to call `ref-set` on any `Ref`s passed arguments, allowing both consumption and resetting of `Ref`s in the same expression. The three examples described in Secion 2.5.1, are each implemented using `dosync-ac` to evaluate it's capabilities.

An implementation of the first example using `dosync-ac` can be found in Listing A.3. The example shows the use of `ac-fn` on Line 6, to print the value of a `Ref` after the transaction commits. Simply using `deref` on the `Ref` after the transaction commits does not guarantee the in-transactional value, as other transactions might have written to the `Ref` after the trans-

```
1 (def counter-ref (ref 0))
2
3 (dosync-ac
4   (if (< (deref counter-ref) 10)
5     (ac-fn #(println "counter-ref:" %)
6            (alter counter-ref inc))
7     (ac-fn-set {counter-ref 0}
8                #(println  "counter-ref:" %) counter-ref)))
```

Listing A.3: After-commit printing to standard output stream

action committed but before the print. If the value reaches ten the value is printed and afterwards set to zero by `ac-fn-set` in Line 8. The value is first extracted so it can be printed, afterwards the value is set to zero by indexing the hash-map given as argument with any `Ref`s in the list of arguments. The macro `ac` is not used in this version of our running example, as it performs the same functionality as `ac-fn` but takes a list of expressions instead of a function, which was subjectively simpler as it allowed the result of `alter` to be passed directly as an argument instead of having to be bound to a symbol using a `let` binding. Using `ac` reduces the amount of boilerplate code necessary compared to `ac-fn` since the expressions do not have to be encapsulated in a function.

```
1  (def arraylist-ref (ref (java.util.ArrayList.)))
2  (def log-ref (ref (writer  "log.txt" :append true)))
3
4  (dosync-ac
5    (alter arraylist-ref identity)
6    (alter log-ref identity)
7
8    (ac
9      (.add (deref arraylist-ref) 0)
10     (.write (deref log-ref) "Added 0 to arraylist-ref")))
```

Listing A.4: After-commit modifying `ArrayList` and logs with possibility to give inconsistent snapshots

The second example is found in Listing A.4. This example also shows how `ac` can be used to execute code after the transaction have committed. First references to an `ArrayList` in Line 1 and a `BufferedWriter` in Line 2 are created. The After-commit transaction is started on Line 4 by `dosync-ac` followed by calling `alter` on the two before mentioned references and therefore getting write locks for the two `Ref` instances. An After-commit block defined by `ac` in Line 8 executes its content that consists of adding

an element to `arraylist-ref` and a write to `log-ref` after the transactions commits. There are two problems with the example in Listing A.4. The first problem is that the write locks of the two `Ref` instances are dropped when the transaction commits and the second problem is that the After-commit block is executed after the transaction have committed which means that the `alter`, `ref-set` and `commute` functions cannot be used. This means that other transactions risk modifying the same references concurrently as the After-commit block.

The third example is not implemented as it requires multiple transactions, one to first extract the row and call the database, and a second to append the key to the result list. This creates the same problems already shown by the second example, where the ordering of the resulting list of keys are not guaranteed to match the original list of rows due to the possibility of inserts being interleaved.

The pros and cons of After-commit can briefly listed below:

Pro: Can be added as a library without any modification to Clojure

Pro: Alternative to existing `agent` model, executes synchronous

Con: Side-effects does not execute in a transaction

A.1.2 Lazy Evaluation

Another method for controlling side-effects in transactions is to introduce lazy evaluation. By lazy evaluation we mean that specified expressions will wait until just before the commit step before being executed. This is done by making sure we have the correct set of locks before we execute the lazy evaluated expressions right before the transaction commits, this ensures that no abort will occur as all necessary locks already have been gathered. This allows a combination of non-transactional and transactional expressions in the same transaction. The result of a lazily evaluated expression can only be used in other lazily evaluated expressions since they are evaluated when the transaction commits. Strictly evaluated expressions return values can be used in both strictly and lazily evaluated expressions. The use of lazy evaluation is done through the interface shown in Listing A.5.

```
1 (le & body)
2 (ler refs & body)
```

Listing A.5: Lazy-evaluation interface

These two macros take a body of expressions and create a `Thunk` for later execution. This is similar to how Haskell implements lazy evaluation

[15]. The macro `le` only takes a list of expressions as argument as it scans the expressions and extracts all `Ref`s directly in the expressions, in addition to the functions used to altering their state. It is necessary as any locks on `Ref`s needs to be acquired by the system when the `Thunk` is created, this ensures that all changes to the `Ref`s can be performed without problems when the lazy expression is evaluated. The `le` macro extract the `Ref`s using the access to `var`s and lexical bindings described in Section 2.1.

The macro `ler` creates the same lazy expression as `le` but allows the developer to pass `Ref`s as an argument. Passing a `Ref` as an argument is required the `Ref` is modified from within a function that is called from the lazy expression block. This is necessary because the macro does not have access to the code of the function. No alternate version of the `dosync` macro is necessary because executing expressions before a transaction commits, requires changes to the Clojure runtime so the expression is stored by the runtime instead of a list created by an alternate `dosync` macro. Lazy expressions are executed as part of the transaction and therefore have access to the in-transactional values of `Ref`s. This makes versions of `le` and `ler` that takes a function and corresponding arguments not necessary as the ability to store values are covered by the access to the in-transactional values.

```
1 (def counter-ref (ref 0))
2
3 (dosync
4   (le (println (deref counter-ref)))
5   (if (< counter-ref 10)
6     (alter counter-ref inc)
7     (ref-set counter-ref 0)))
```

Listing A.6: Lazy evaluation printing to standard output stream

An example of using `le` to implement the first example can be seen in Listing A.6 where a `Ref` is changed and then printed if the transaction commits. All `Ref`s write locks are extracted by `le` to ensure that the lazy expression can be executed without the transaction is aborted.

The second example can be seen in Listing A.7. Two references to an `ArrayList` and a `BufferedWriter` are created in Line 1 and 2. The lazy-evaluated block defined by `le` in Line 5 executes its content that adds an element to `arraylist-ref` and writes a string to `log-ref` when the transaction has all the necessary write locks.

The third example in Listing A.12 describes a transaction where a `vector` of database rows are inserted into a database on Line 6, the function returns the key assigned to the row from the database which is inserted into a `vector` on Line 10, and last the row inserted are removed from the list of rows on Line 11. The example ensures that the side-effect is executed transactionally

```
1 (def arraylist-ref (ref (java.util.ArrayList.)))
2 (def log-ref (ref (writer  "log.txt" :append true)))
3
4 (dosync
5   (le
6     (alter arraylist-ref .add 0)
7     (alter log-ref .write "Added 0 to arraylist-ref")))
```

Listing A.7: Lazy evaluation modifying `ArrayList` and logs

```
1 (def keys-ref (ref []))
2 (def rows-ref (ref []))
3
4 (dosync
5   (le (let [row (first (deref rows-ref))
6             next-key (database-insert row)]
7         (alter keys-ref conj next-key)
8         (alter rows-ref rest))))
```

Listing A.8: Inserting database rows and returning keys

safe, by taking exclusive locks for both `Ref`s before executing the side-effect on Line 6, thereby ensuring that the transaction will not abort when the `Ref`s are updated.

A short list as pros and cons for the experiment can be seen below.

Pro: Safely execute side-effects inside transactions after getting the necessary locks

Con: Modification of Clojure is necessary

Con: The developer must be aware of what `Ref`s are changed

Con: Separates execution into two stages, with no standard rules for passing of data

Lazy Evaluation Exception Handling

Using the lazy evaluated expressions documented in Section A.1.2, allows functions with side-effects to be executed safely inside a transaction. However exceptions need to be handled explicitly as they would otherwise cause the transaction to abort, we experimented with a few methods for handling exceptions each with their own strength and weaknesses.

Suppressed Exceptions The first method was to simply catch and store exceptions until the `le` block was queried for its result by `deref`, a behaviour already used by Clojure for `future`s and `agent`s. The system could be ported directly for use with `le`, but have a few drawbacks despite ensuring that the `le` function executes without allowing the transaction to abort.

Suppressing exceptions terminates the execution of the `le` function that threw it, which allows execution of the next `le` function to start despite the first having failed. If a `le` has thrown an exception and is dereferenced by another `le` function the exception is propagated and terminates this function as well. This allows for situations where only some side-effects are executed without any notification to the developer about the not executed side-effects. If a `le` function is dereferenced outside a lazy expression the entire transaction is terminated due to the value not having been computed yet, and exceptions are allowed to propagate outside a `dosync` block in Clojure.

Checked Exceptions Another solution was to enforce exception handlers to be created for all exceptions declared by a `le` function, similar to checked exceptions in Java. The solution would guarantee all `le` to execute unless the developer purposely re-threw an exception, which would allow the developer to force a transaction to terminate with some side-effects already executed.

Clojure however wraps checked exceptions thrown by Java code with unchecked `java.lang.RuntimeException` and does not utilise checked exceptions for Clojure code, so Clojure functions do not declare what exceptions they might throw by default. A system to check for possible exceptions thrown by a function would have to be built on top of Clojure for this method of exception handling to be used. An alternative implementation could remove the need for such a framework by requiring the developer to declare an exception handler general enough to catch all exceptions around all `le` functions, it would then be the developer's task to know which exception a function can throw and how each should be handled. Using checked exceptions or a forced exception handler would force developers to implement appropriate exception handlers for all possible kind of exceptions a given body of code can throw, adding additional complexity to the code.

Wrap in LazyEvalException The last solution we evaluated was to catch all exceptions thrown by an `le` function and wrap the exception in a `LazyEvalException`. This makes it clear to the developer that the exception was raised inside the `le` function and will have to be handled for the code to execute correctly. Any instances of `RetryEx` are caught in `le` and wrapped to prevent the transaction from aborting, this lets the developer know the that the execution of the `le` block failed. This prevents multiple executions of `le` functions implicitly, and makes it clear to the developer that an undeclared

`Ref` was encountered.

Wrapping exceptions does still allow a `le` function to execute some of its statements before being terminated, creating a situation with only some side-effects executed. This is similar to the problems with suppressing exceptions, but with the opposite result. Instead of hiding the problem from the developer, is it made clear that changes must be made to the `le` function for the code to operate correctly.

This provides a compromise between the other two solutions. It forces the least amount of work from the developer using `le`, no extra work is needed if no exceptions are thrown, matching exception handling in Clojure in general. At the same time this solution makes it clear to the developer if any exceptions were raised inside the `le` function, instead of hiding the problem. Due to this compromise, we decided subjectively this solution were the best.

A.2 Compensate

Compensate reverts a side-effect which just has been executed by the transaction if the transaction abort. This ensures that when a transaction commits, the side-effects have only been executed once effectively.

A.2.1 Undo

Undo allows the developer to specify a function or expression to ensure that in case of an abort, the changes performed by the transaction are rolled back before aborting the transaction. Undo does thus not require any changes to the existing code executed as part of a transaction, but requires the programmer to specify the needed additional code for rolling back side-effects. A drawback of this approach is that it can add additional execution time to the program, if expensive operations have to be done and undone multiple times, if they can be undone at all as with for example database queries. The signature for the experimental implementation of the idea can be seen as Listing A.9.

```
1 (dosync-undo & body)
2
3 (undo & body)
4 (undo-fn func & args)
```

Listing A.9: Undo signature

Undo consists of three macros. First an extended version of `dosync` named `dosync-undo`, seen on Line 1, which allows for the specification of

code that will be executed when the transaction aborts. The two other macros allow the developer to specify code to be executed on abort, these can be seen on Line 3 and 4. The macro `undo` takes a list of expressions while `undo-fn` takes a function and corresponding arguments.

```
1 (def counter-ref (ref 0))
2
3 (dosync-undo
4   (println (deref counter-ref))
5   (undo (println "Ignore previous println"))
6   (if (< counter-ref 10)
7     (alter counter-ref inc)
8     (ref-set counter-ref 0)))
```

Listing A.10: Undo printing to standard output stream

An example of the Undo is shown as Listing A.10. Undo is not suitable for handling printing inside a transaction, as the printed statement cannot be undone. Instead the developer can compensate for the print action as seen in Line 5. Any code to be executed when the transaction aborts needs to be registered before the transaction actually aborts, which is why the `undo` macro is placed on Line 5, before the two branches calling `alter` starting on Line 6.

```
1  (def arraylist-ref (ref (java.util.ArrayList.)))
2  (def log-ref (ref (writer  "log.txt" :append true)))
3
4  (dosync-undo
5    (let [element (Integer. 0)]
6      (alter arraylist-ref (.add element))
7
8      (undo
9        (alter arraylist-ref #(.remove % element)))
10
11     (alter log-ref (.write log-ref "Added 0 to arraylist-ref"))))
```

Listing A.11: Undo modifying `ArrayList` and logs

A second example is shown as Listing A.11 which shows that it is possible to compensate for the side-effect of adding an element to the `ArrayList` that can be seen in Line 6. This is compensated by `undo` in Line 8 as it removes the just added element if the transaction aborts. The expression captures the `Integer` object defined by symbol `element`, which allows it to use reference equality to remove it from the list without risk of deleting elements added by other transactions. The last line in the transaction that writes to a log

does not need to be supported by a roll-back because `alter` aborts before executing the function and not when the transaction commits like `commute`.

```
1  (def keys-ref (ref []))
2  (def rows-ref (ref vector-of-rows))
3
4  (dosync
5    (let [row (first (deref rows-ref))
6          next-key (database-insert row)]
7            (undo
8              (database-remove next-key))
9
10             (alter keys-ref conj next-key)
11             (alter rows-ref rest)))
```

Listing A.12: Inserting database rows and returning keys

The third example in Listing A.12 describes a transaction where a `vector` of database rows are inserted into a database on Line 6, the function returns the key assigned to the row from the database, a function for removing the inserted row is then registered on Line 8 to ensure the transaction does not leave side-effects if updating the `Ref`'s fail. The key is then inserted into a `vector` on Line 10, and last the row inserted are removed from the list of rows on Line 11. Implementing the example using Undo create some undesired behavior, first if the update and removal operations are not executed in the same database transaction would it allow other queries to see a row that the STM transaction might delete. Also applying multiple insert and remove operations on the database adds an unnecessary additional load to the database.

A short list advantages and disadvantages for the experiment is presented below.

Pro: Minimal changes to the transactional code

Pro: Can be added as a library without any modification to Clojure

Con: The state of the side-effects needs to be recorded in order to only compensate for executed side-effects

Con: Some operations can be expensive to compensate for

Con: Some operation cannot not be compensated for

A.3 Irrevocability

Irrevocability can be seen as a promise to the developer that the transaction will commit without aborting, therefore side-effects can be executed safely.

A.3.1 Check-Run

This experiment of introducing irrevocability in Clojure is dubbed Check-Run. The idea is to simulate the execution of the code in the transaction without performing any operations, checking for any operation that could potentially force the transaction to abort before actually executing it. In the check phase, any needed write locks are taken by the transaction. If some write locks cannot be taken the transaction will follow the normal abort semantics for a transaction, continuing to check the code until all write locks needed are captured by the transaction and may abort during the check phase. When all locks are acquired the transaction code is executed. The transaction is guaranteed to commit as all locks have been taken and thereby avoiding a possibility for a abort when actually executing the transaction code. The interface of Check-Run is shown in Listing A.13.

```
1 (dosync-checked & body)
2 (dosync-checked-ref refs & body)
```

Listing A.13: Check-Run signature

The two macros use Clojure's `dosync` macro and performs the check step before executing the transaction. The difference between the two presented macros is that `dosync-checked` extracts all `Ref`s from inside the code block. This operation adds additional runtime cost, and `ref`'s modified by functions cannot be extracted as Clojure does not provides a means to get the Clojure source code of functions in macros or other functions. This problem is solved by providing another macro `dosync-checked-refs` that allows the developer to pass a list of `Ref`s or a dictionary of a `Ref` as key and a function as the value. If a list is passed then `alter` is used to get a write lock on all `Ref`s. A dictionary can be given with `alter` or `commute` depending on what level of write protection the developer wants.

```
1 (def counter-ref (ref 0))
2
3 (dosync-checked
4   (println (deref counter-ref))
5   (if (< counter-ref 10)
6     (alter counter-ref inc)
7     (ref-set counter-ref 0)))
```

Listing A.14: Check-run printing to standard output stream

An example of using `dosync-checked` to implement the first example can be seen in Listing A.14 where a `var` containing a `Ref` is changed and

then printed inside a transaction. The `dosync-checked` macro extracts any `Ref`s as described above and takes a write lock on these `Ref`s to ensure that the transaction can be executed without retrying.

```
1 (def arraylist-ref (ref (java.util.ArrayList.)))
2 (def log-ref (ref (writer  "log.txt" :append true)))
3
4 (dosync-checked
5   (alter arraylist-ref (.add 0))
6   (alter (.write log-ref "Added 0 to arraylist-ref")))
```

Listing A.15: Check-run modifying `ArrayList` and logs

Another example of Check-Run is shown in Listing A.15, here the second example is implemented. On Line 1 and 2 two references are created to an `ArrayList` and a `BufferedWriter`. The Check-Run block is defined by `dosync-checked` in Line 4 which carries out the addition of an element to the referenced `java.util.ArrayList` and writes a string to a log file.

```
1 (def keys-ref (ref []))
2 (def rows-ref (ref vector-of-rows))
3
4 (dosync-checked
5   (let [row (first (deref rows-ref))
6         next-key (database-insert row)]
7             (alter keys-ref conj next-key)
8             (alter rows-ref rest)))
```

Listing A.16: Cehck-run inserting database rows and returning keys

The third example in Listing A.12 describes a transaction where a `vector` of database rows are inserted into a database on Line 6, the function returns the key assigned to the row from the database which is inserted a `vector` on Line 7, and last the row inserted are removed from the list of rows on Line 8. The example ensures that the side effect is executed transactionally safe, by taking locks for both `Ref`'s before executing the side-effect on Line 6, thereby ensuring that the transaction will not abort when the `Ref`'s are updated.

Pro: Works like a normal `dosync` block but allows side-effects by getting the necessary locks before execution of the transaction code

Con: The developer must be aware of what `Ref`s are changed

Con: Modification of Clojure is necessary

A.4 Transaction Control

The concept of Transaction Control makes it possible to control a transaction's behaviour by the use of explicit constructs. We have performed two experiments from this category.

A.4.1 Retry, Or-else and Terminate

As a supplement to implementing ways to support the use of side-effects in STM transactions, some of the side-effects used for synchronisation can be removed by implementing constructs for synchronising threads based directly on the state of the data in `Ref`s. Two such functions named `retry` and `orElse` already exist in Haskell [21]. The `retry` functions allow the developer to abort a transaction and block the thread executing the transaction based on the `Ref`s in the transaction. The `or-else` function allows multiple STM operations to be chained together. The execution of this chain will only go to the next operation if the current operation would make the transaction abort. The signatures for the two functions and their overloaded versions can be seen in Listing A.17.

```
1 (retry)
2 (retry refs)
3
4 (retry-all)
5 (retry-all refs)
6
7 (or-else & body)
8
9 (terminate)
```

Listing A.17: Retry and or-else signatures

Two different functions are created for blocking the execution of a transaction, the first is named `retry` and the second `retry-all`. `retry` blocks a transaction until a single `Ref` is changed, while `retry-all` waits until all `Ref`s given as argument have been written to by other transactions. The versions of the two functions without any arguments, blocks the thread based on the `Ref`s that were dereferenced during the transaction, a behaviour that matches the implementation in Haskell at the time of writing. The second version takes a sequence of `Ref`s as argument, and blocks until either one or all of them have been written to. This allows the developer to wait on changes to a subset of `Ref`s dereferenced in the transaction, or `Ref`s that had not been dereferenced in the transaction at all. This extension of `retry` and `retry-all` is natural since the methods in the Java part of the runtime

implementing `retry` and `retry-all` takes a set of `Ref`s, making it trivial to allow the developer using `retry` or `retry-all` to specify a set of `Ref`s.

The `or-else` function combines multiple expressions inside a STM transaction and executes them one at a time until one executes without aborting the transaction, the result of this expression is then returned. If all expressions forces the transaction to abort, then the entire transaction aborts. The function allows the developer to specify a fallback for operations that may not succeed, instead of continuously aborting the transaction until one specific statement succeeds.

Lastly the `terminate` construct allows the developer to terminate the transaction, preventing it from retrying. One use of this function is to combine `terminate` with `or-else` to skip a STM transaction if a resource is unavailable and using it would force the transaction to abort.

```
1 (def updated-ref (ref 0))
2 (def static-ref (ref 10))
3
4 (dosync
5   (if (== (deref updated-ref) (deref static-ref))
6     (ref-set updated-ref 0)
7     (retry updated-ref)))
```

Listing A.18: Retry example

As transaction control does not allow the use of side-effects directly can it not be used to implement the examples described in Section 2.15, instead examples specific for transaction control was created. An example of using `retry` can be seen in Listing A.18. Here `updated-ref` is set to zero when it reaches ten. The incremental changes to `updated-ref` is done by other transactions. `retry` is used to block the transaction until `updated-ref` is modified. `static-ref` in this scenario is never updated outside the shown transaction, therefore waiting on both `Ref`s would result in a deadlock. Which is why `retry` is called with only `updated-ref`.

```
1 (dosync
2   (execute-db-operation
3     (or-else
4       (alter db-connection-one identity)
5       (alter db-connection-two identity)
6       (alter db-connection-three identity)
7       (terminate))))
```

Listing A.19: Or-else and terminate example

The second example is shown as Listing A.19. Here `or-else` and `terminate` are used by a thread to execute an operation on a database if a database is available, otherwise the thread continues its operation. On Line 3 `or-else` is used to secure one of three database connections against other transactions by using the function `alter`. If one of the three database connections is successfully secured then the value of the expression is returned which for this example is the database connection it received as input. If all database connections are currently in use by other transactions then it calls `terminate` and the thread will terminate the transaction and continue without waiting for a database connection.

A short summary of this experiment in comparison to Clojure's existing control over transaction can be seen in the following itemize as advantages and disadvantages.

Pro: Allow synchronisation without side-effects

Con: Cannot compensate for any side-effect

Con: Modification of Clojure is necessary

APPENDIX B

Experimental Implementations

This appendix contains implementation details of the experiments described in Appendix A. The experiments were used to see how each approach were compatible with Clojure's STM implementation while gaining better general understanding of the implementation. For more information about how to obtain the code see the preface.

B.1 After-Commit

The After-Commit experiment described in A.1.1 was implemented as three macros and two private functions. The private functions are used to verify input and perform updates to `Ref`s.

```
1 (defmacro dosync-ac
2   [& body]
3   `(let [~'&ac-funcs-and-args (java.util.ArrayList.)
4          dosync-return# (dosync
5                           (.clear ~'&ac-funcs-and-args)
6                           ~@body)
7          after-commit-return# (map #(apply (first %) (second %))
8                                    ~'&ac-funcs-and-args)]
9      (vec (conj after-commit-return# dosync-return#))))
```

Listing B.1: The `dosync-ac` macro

The `dosync-ac` shown in Listing B.1 enhances Clojure's existing `dosync` macro with the ability to store functions and arguments for later execution. This is achieved by a `let` binding named `&ac-funcs-and-args` in Line 3 that binds an instance of a `java.util.ArrayList` and executes the current transaction inside its scope. This gives any function executed as part of the transaction access to a mutable storage allowing easy aggregation of functions and arguments without any of the downsides of using a mutable

collection because it is only accessible from inside the `let` binding and it is emptied every time the transaction is executed. This causes retries to be irrelevant. Each After-Commit function is added to `&ac-funcs-and-args` as the transaction executes. The `map` function in Line 7 is lazily evaluated in Clojure and executed all After-Commit functions. The value returned from the transaction itself is the first value in the list of returned values. The other returned values in the list comes from the After-Commit functions in the executed order. This list is converted to a `vector` to force executing of the lazy `map` sequence and returned to the developer.

The name of the binding, in this case `&ac-funcs-and-args`, must however be static for it to be accessible by the other macros using it, to do so is the symbol prefixed with `~'` which first quotes and then unquotes the symbol resulting in a unqualified symbol. An unqualified symbol in Clojure binds to the first instance it encounters doing lookup with the same name when ignoring namespaces, qualified symbols contain namespace and access the data directly by ignoring any unqualified symbols. This makes `&ac-funcs-and-args` inaccessible for the developer when using the macro, unless the developer purposely creates an unqualified symbol with the name `&ac-funcs-and-args` as all symbol are fully qualified by default in Clojure. It would be possible to use dynamic bindings for `&ac-funcs-and-args` like it was done in the earlier experiments described in Appendix C. However this would require a `def` to be created outside the scope of the macro in which the macro could store the `java.lang.ArrayList` during execution of the transaction. While this method would allow functions executed in the transaction to add functions to be executed after commit, as bindings are dynamically scoped instead of lexically scoped like `let`, it would also increase the chance of hiding the developer's symbols as the binding would be fully qualified and visible in a much wider scope, so we decided to use a `let` binding.

```
1 (defmacro ac
2   [& body]
3   `(dosync-ac-helper ~'&ac-funcs-and-args nil (fn [_#] ~@body) nil))
```

Listing B.2: The `ac` macro

```
1 (defmacro ac-fn
2   [func & args]
3   `(dosync-ac-helper ~'&ac-funcs-and-args nil ~func ~@args))
```

Listing B.3: The `ac-fn` macro

```
(defmacro ac-fn-set
  [ref-set-map func & args]
  `(dosync-ac-helper ~'&ac-funcs-and-args ~ref-set-map ~func ~@args))
```

Listing B.4: The `ac-fn-set` macro

The three other macros used directly by the developers are `ac`, `ac-fn` and `ac-fn-set` shown respectively in Listing B.2, B.3, B.4. These macros add a function and optional arguments to `&ac-funcs-and-args`. The `ac` macro converts the expressions passed as it's argument to a function that is passed to the `dosync-ac-helper` function. The `ac-fn` macro allows the developer to specify functions with arguments. The `ac-fn-set` macro allows a dictionary to be passed with a `Ref` as key. Any `Ref` contained in the dictionary that are passed as argument to `ac-fn-set` will be dereferenced so its value can be used by the function to be run on commit, and then set to the value stored in the dictionary. This is intended to allow multiple `Ref`'s to be reset to a starting value in one expression instead of one per reference, reducing the chance of one being changed without the other if the function to be run on commit consumes the current value of the `Ref`'s.

```
(defn dosync-ac-helper
  [^java.util.ArrayList array ref-set-map func & args]
  (when-not (fn? func)
    (throw (IllegalArgumentException. "argument (fun) must be fn")))
  (if (nil? ref-set-map)
    (.add array [func args])
    (let [updated-args (map #(update-arg-by-ref-map ref-set-map %) args)]
      (doall updated-args)
      (.add array [func updated-args])))
  nil)
```

Listing B.5: The `dosync-ac-helper` function

The first of the two private helper functions are `dosync-ac-helper` shown in Listing B.5. Its job is to verify input to `ac`, `ac-fn`, `ac-fn-set` and update `Ref`s if necessary before adding everything to `&ac-funcs-and-args`. First the function argument is verified on Line 3 before the function checks if a `map` is passed on Line 5. If a `map` is passed the second helper function `update-arg-by-ref-map` is called on every entry in the map to update the value of any `Ref`s specified in the map.

The second helper function `update-arg-by-ref-map` seen in Listing B.6 verifies that its arguments is a `Ref` and that it is contained in the map passed as `ref-set-map`. If so, the `Ref` is set to the value stored in the `map` and the

```
(defn- update-arg-by-ref-map
  [ref-set-map arg]
  (if (and (instance? clojure.lang.Ref arg) (contains? ref-set-map arg))
    (let [arg-val @arg]
      (ref-set arg (ref-set-map arg))
     arg-val)
    arg))
```

Listing B.6: The `update-arg-by-ref-map` function

original value of the `Ref` is returned, otherwise the argument simply passes through.

B.2 Lazy Evaluation

The implementation of Lazy Evaluation, described in A.1.2, required changes to Clojure's runtime and new constructs for creating lazy expressions. An overview of the architecture of the implementation can be see in Figure B.1.

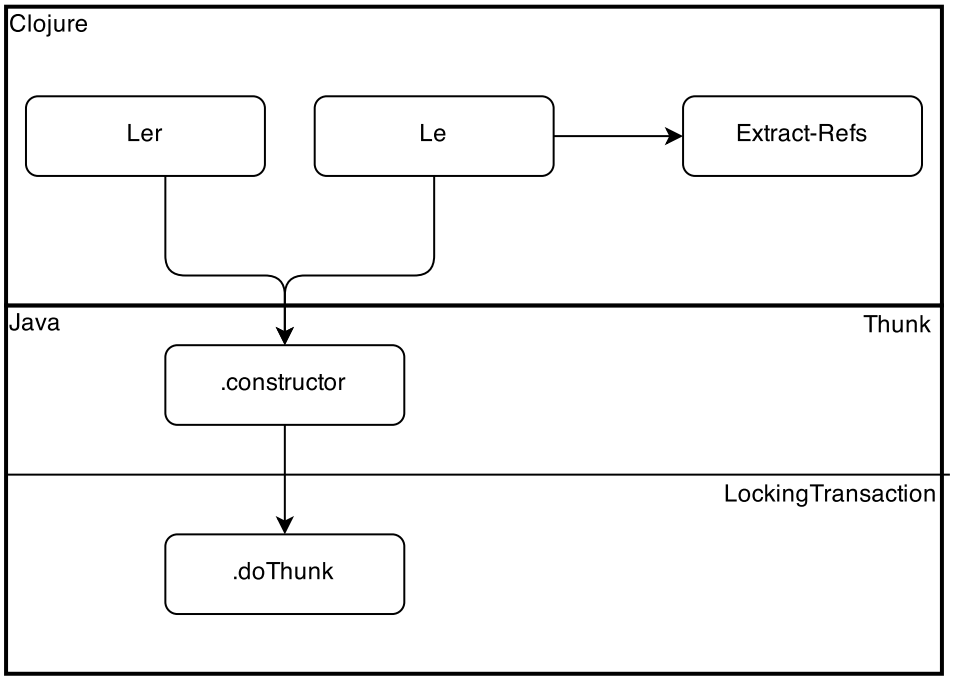

Figure B.1: Architecture of lazy expressions

The interface for lazy expressions consists of the macros `le` and `ler` that was described in Section A.1.2. The `le` macro supports the automatic

extraction of `Refs` by calling the function `extract-refs` with the body of the lazy expression as input. After the requested locks have been taken for each `Ref` then an instance of the class `Thunk` is created that holds the body of the lazy expression and delays execution until the STM transaction is guaranteed to commit. The `Thunk` object is then passed to the current instance of `LockingTransaction`.

```
(defmacro le
  [& body]
  (let [lexically-scoped-bindings (keys &env)]
    `(do
       (doseq [r# (extract-refs '() '~body ~@lexically-scoped-bindings)]
         (alter r# identity))
       (Thunk. (fn [] ~@body)))))
```

Listing B.7: The `le` macro

In Listing B.7 `le` is shown which automatically extracts all `Refs` from the `body` passed as the only argument, lexically scoped `Ref`'s cannot be look up by `resolve` and are extracted from `&env` a symbol available inside `defmacro`. After the `Refs` are extracted, write locks are taken for all `Refs` using `alter` as it ensures that changes are synchronised. The semantics provided by `commute` is less strict and require the operation performed to be commutative. To specify a `Ref` to be commutative, the developer needs to use `ler` instead of the more automatic `le`.

```
(defmacro ler [refs & body]
  `(do
     (cond
           (map? ~refs)
           (doseq [[le-ref# func#] ~refs]
             (if (or (= func# commute) (= func# alter))
               (func# le-ref# identity)
               (throw (IllegalArgumentException.
                        "map value must be alter or commute"))))
           (vector? ~refs) (doseq [r# ~refs] (alter r# identity))
           :else (throw (IllegalArgumentException.
                          "(refs) is neither a vector or a map")))
     (Thunk. (fn [] ~@body))))
```

Listing B.8: The `ler` macro

The `ler` macro shown as Listing B.8 depends on the developer to indicate which locks are needed for the `Refs` in the lazy expression. It expects either a `vector` or a `map` as an extra argument in addition to the body of the

expression. The type of argument `refs` is checked in Line 3. If `refs` is a map then the key of the entry is used as an argument to the function found in the value of the entry. If `refs` is a `vector` then write locks is implied and `alter` is executed for each `Ref`. The `identity` functions is used as the function for both `commute` and `alter` to get necessary locks without changing the value of the `Ref`. Finally a `Thunk` is created on Line 13 with the expression to be executed on commit.

```
1  (defn extract-refs [acc & body]
2    (distinct (reduce
3      (fn [acc elem]
4        (cond
5          (and (ref? elem) (var? elem)) (conj acc @elem)
6          (ref? elem) (conj acc elem)
7          (symbol? elem) (do
8                           (let [elem-var (some-> elem resolve var-get)]
9                             (if (ref? elem-var)
10                              (conj acc elem-var)
11                              (if (or (seq? elem-var) (vector? elem-var))
12                                (extract-refs acc elem-var)
13                                acc))))))
14     acc (flatten body))))
```

Listing B.9: The `extract-refs` function

The function `extract-refs` shown as Listing B.9 goes through the expression passed to `le` and extracts any `Ref`s directly in the code as seen in Line 5 or accessible through a `var` as seen in Line 8. If any symbols resolve to a sequence the function is called recursively as shown on Line 11, as `flatten` only removes explicit nesting and not collections accessed through symbols.

The changes to `LockingTransaction` are shown as Listing B.10. The class is extended with a method for adding `Thunk`s shown on Line 3, the method verifies that the transaction is running before adding the `Thunk`. The `run` method shown on Line 10 is executed by `LockingTransaction` to evaluate the STM transaction, a loop was added shown on Line 12 to execute the function stored in each `Thunk` and store the return value so the developer can retrieve it through `deref`.

```
1  public class LockingTransaction{
2  ...
3    void doThunk(Thunk thunk) {
4      if ( ! info.running()) {
5          throw retryex;
6      }
7      thunks.add(thunk);
8    }
9    ...
10   Object run(Callable fn) throws Exception{
11   ...
12     for (Thunk thunk : thunks) {
13         thunk.run();
14     }
15   ...
16   }
17 ...
18 }
```

Listing B.10: Additions to the `LockingTransaction` class

B.3 Undo

The Undo experiment described in A.2 was implemented as three macros.

```
1  (defmacro dosync-undo
2    [& body]
3    `(let [~'&undo-funcs (java.util.ArrayList.)]
4       (dosync
5         (try
6           ~@body
7           (catch Error e#
8             (doall (map #(apply (first %) (second %)) ~'&undo-funcs))
9             (.clear ~'&undo-funcs)
10            (throw e#))))))
```

Listing B.11: The `dosync-undo` macro

The `dosync-undo` macro seen in Listing B.11 extends the normal transaction with undo capabilities. This is accomplished by placing the normal transaction in a `let` binding that binds an instance of a `java.util.ArrayList` to the symbol `&undo-funcs` as seen in Line 3. `&undo-funcs` is used by the other two macros to store functions that needs to be executed when the transaction aborts. The transaction is placed in the `try` part of a try-catch block because a transaction aborts by throwing an error. If an error is thrown during execution of the transaction it is caught in Line 7 and then

the `catch` part is executed. This part executes all the functions stored in `&undo-funcs` in Line 8. Then it clears `&undo-funcs` in Line 9 and throws the error once again which aborts the transaction.

The two other macros allows the developer to specify expressions or functions to be run if the transaction aborts.

```
1 (defmacro undo
2   [& body]
3   `(.add  ~'&undo-funcs [#(do ~@body) nil]))
```

Listing B.12: The `undo` macro

```
1 (defmacro undo-fn
2   [func & args]
3   `(.add  ~'&undo-funcs [~func '~args]))
```

Listing B.13: The `undo-fn` macro

For adding function to be executed does the Undo experiment also implemented the macros `undo`, that can be seen in Listing B.12, and `undo-fn`, which can be seen in Listing B.13, as a means to store functions in `&undo-funcs`. The two macro are very similar because `undo` allows the developer to pass a body and store it as a function without any arguments where as `undo-fn` will store a function with specified arguments.

B.4 Check-Run

Check-Run experiment, described in A.3, is implemented on top of lazy evaluation described in Section B.2. Check-Run is sharing the same concept as Lazy Evaluation of verifying a list of expressions that can be executed before performing the actual execution. Check-Run however hides this implementation detail providing the developer with the same syntax as the existing `dosync` macro.

```
1 (defmacro dosync-checked
2   [& body]
3   `@(dosync
4       (le ~@body)))
```

Listing B.14: The `dosync-checked` macro

Check-Run is implemented as two macros, `dosync-checked` shown as Listing B.14 and `dosync-checked-ref` shown as Listing B.15. The two macros are nearly identical and passes the entire contents of the `dosync` block given, as the argument to a lazy expression. As described in Section B.2 lazy expressions takes the needed read and write locks using Clojure's existing locking semantics before executing, ensuring a retry is impossible when the expression is actually executed. When the body of the `Thunk` has been executed, the `Thunk` returns the result of the execution of the body when dereferenced instead of returning the `Thunk` itself, making `dosync-checked` and `dosync` equivalent from a developer's perspective.

```
1 (defmacro dosync-checked-ref
2   [refs & body]
3   `@(dosync
4       (ler ~refs ~@body)))
```

Listing B.15: The `dosync-checked-ref` macro

The two macros only differ in how `Ref`s are handled, `dosync-checked` uses the lazy evaluation macro `le` and `dosync-checked-ref` uses `ler`. This means that `dosync-checked` will parse the expressions passed as its `body` arguments for any `Ref`s and take a write lock on them. It is possible to define `Ref`s that cannot be extracted automatically, by placing them inside a called function as described in detail in Section B.2. The function `dosync-checked-ref` is a solution to this problem and takes a `map` as an extra argument compared to `dosync-checked`. The `map` must contain any `Ref`s used in the transaction as keys, and either `commute` or `alter` as values indicating what locking semantics should be used. Furthermore the function `dosync-checked-ref` skips the runtime extraction of the specified `Ref`s.

B.5 Transaction Control

This section will describe the experimental implementation of the transaction control system, described in Section A.4. The final implementation, described in Section 3.2, reuses many parts of the experimental implementation. The reused parts will be briefly described in this section and then further explained in Section 3.2. The implementation consists of two parts. The first part is a set of Clojure functions in the Clojure part of the runtime, described in Section B.5.1. The second part consists of methods that are added to the `LockingTransaction` class and the addition of the `STMBlockingBehavior` class described in Section B.5.3.

B.5.1 Clojure Implementation

The implementation of the Clojure interface for transaction control can be seen in Listing B.16. The interface provides accesses to transaction control functionality implemented as public methods on `LockingTransaction`, allowing developers to use the functionality in Clojure without knowledge about `LockingTransaction` and the public methods.

```
1  (defn retry
2    ([] (.doBlocking (LockingTransaction/getEx) nil false))
3    ([refs] (.doBlocking (LockingTransaction/getEx) refs false)))
4
5  (defn retry-all
6    ([] (.doBlocking (LockingTransaction/getEx) nil true))
7    ([refs] (.doBlocking (LockingTransaction/getEx) refs true)))
8
9  (defn or-else
10   [& body]
11   (.doOrElse (LockingTransaction/getEx)
12              (java.util.ArrayList. ^java.util.ArrayList body)))
13
14 (defn terminate
15   [] (.abort (LockingTransaction/getEx)))
```

Listing B.16: Clojure functions for the `retry`, `retry-all`, `or-else` and `terminate` constructs

The functions `retry` and `retry-all` can be seen in Listing B.16 on Line 1 and Line 5 respectively. They provides the same interface but have different semantics. Both block until a set of specified `Ref`'s are written to by another transaction. The function `retry` block until any of the `Ref`'s are written to. The function `retry-all` block until all the `Ref`s in the set are updated.

The `or-else` function is implemented in Clojure in Listing B.16 on Line 9. The function takes a list of expressions where it will try to execute them in the given order until one of them executes without forcing the transaction to retry.

The function `terminate` is implemented on Line 14 in Listing B.16. This function makes it possible to terminate a transaction by aborting it without any re-executing.

B.5.2 STMBlockingBehavior Implementation

The concept of blocking threads based on a set of `Ref`s is introduced in the abstract class called `STMBlockingBehavior`. A blocking behavior is used to block a thread based on a set of `Ref`s. Each behaviour has different semantics but they all extend the `STMBlockingBehavior`. Two concrete

implementations were created, `STMBlockingBehaviorAny` blocks the thread until any `Ref`s defined have been written to by other transactions and is used by `retry`, while `STMBlockingBehaviorAll` blocks the thread until all `Ref`s defined have been written to by other transactions and is used by `retry-all`.

```
1  abstract class STMBlockingBehavior {
2    protected Set<Ref> refSet;
3    protected CountDownLatch cdl;
4
5    STMBlockingBehavior(Set<Ref> refSet) {
6      this.refSet = refSet;
7      this.cdl = new CountDownLatch(1);
8    }
9
10   void await() throws InterruptedException {
11       this.cdl.await();
12   }
13
14   abstract void handleChanged(Set<Ref> refSet);
15 }
```

Listing B.17: The abstract super class for the specific blocking behaviors

All blocking behaviours are child classes of the abstract super class `STMBlockingBehavior` which defines the interface of the child classes. The constructor of the class is found in Listing B.17 on Line 5. The constructor takes one argument `refSet` which is a `Set` of `Ref` instances the `STMBlockingBehavior` blocks on. Then the constructor initialises an instance of the class `CountDownLatch` on Line 7, this is used to block the thread.

`STMBlockingBehavior` contain two methods which are the `await` method that simply blocks on the `CountDownLatch` created by the constructor and the abstract method `handleChangedSet` that is overridden by subclasses to define when the `STMBlockingBehavior` should unblock the thread. The `handleChanged` method updates its internal `refSet` based on the values of `refSet` given as an argument, and unblocks if the requirements for the `STMBlockingBehavior` is fulfilled. This implementation is therefore optimised for blocking on a large set of `Ref`'s where each transaction only writes to a small set of `Ref`'s.

B.5.3 LockingTransaction Implementation

The two subclasses of `STMBlockingBehavior` are used by methods defined in `LockingTransaction` which will be described now.

```
1  public void doBlocking(HashSet<Ref> refs, IFn fn, ISeq args,
2    boolean blockOnAll) throws InterruptedException, RetryEx {
3    if ( ! info.running()) {
4      throw retryex;
5    }
6    if (refs == null) {
7      refs = this.gets;
8    }
9    if (blockOnAll) {
10       this.blockingBehavior = new STMBlockingBehaviorAll(refs);
11   } else {
12       this.blockingBehavior = new STMBlockingBehaviorAny(refs);
13   }
14   LockingTransaction.blockingBehaviors.add(this.blockingBehavior);
15   throw retryex;
16 }
```

Listing B.18: The `doBlocking` Java method in the transaction `LockingTransaction` class

The method `doBlocking`, seen in Listing B.18, adds the correct blocking behaviour to the transaction which makes the transaction block when it retries. If the `refs` argument is `null` it means that it should block on all `Ref` instances read in the transaction, therefore `refs` is assigned to `gets`, the set of dereferenced `Ref` instances from the current transaction. The argument `blockOnAll`, in Line 9, decides if the `doBlocking` method must wait for changes in any or all `Ref` instances given in `refs`.

```
1 if (this.blockingBehavior != null) {
2   this.blockingBehavior.await();
3   LockingTransaction.blockingBehaviors.remove(this.blockingBehavior);
4   this.blockingBehavior = null;
5 }
6 gets.clear();
```

Listing B.19: If the transaction have a blocking behaviour then it must wait for it to resolve.

In Listing B.19 code from `LockingTransaction` is shown. The code is the first part of the transaction's `run` method and checks if the transaction has a blocking behaviour set on Line 1. If so, the transaction is on Line 2 forced to block until the `CountDownLatch` in the blocking behaviour is counted down to zero where it will unblock.

```
1 for (STMBlockingBehavior blockingBehavior :
2       LockingTransaction.blockingBehaviors) {
3   blockingBehavior.handleChanged(this.vals.keySet());
4 }
```

Listing B.20: Notify all blocking behaviours in other transactions with which `Ref`s have been written to by the committed transaction

When the transaction has committed it notifies all blocking behaviours in the global set of blocking behaviours with the set of `Ref`s that was written to by the committed transaction by calling `handleChanged` on each `STMBlockingBehavior` on Line 3 in Listing B.20.

```
1  void stop(int status){
2    if(info != null) {
3      synchronized(info) {
4          info.status.set(status);
5          info.latch.countDown();
6      }
7      info = null;
8      sets.clear();
9      commutes.clear();
10     if (status != COMMITTED) {
11         vals.clear();
12     }
13   }
14 }
```

Listing B.21: Clear only the set of `Ref`s that have been written to on commit in the `stop` method

As seen in Listing B.20 the set of `Ref`s that has been written to in the transaction is needed when notifying blocking behaviours. This set is cleared by the method `stop` when a transaction either commits or aborts, as each `STMBlockingBehavior` requires the set of `Ref`s changed by the transaction committing, must an exception be added to `stop`. The exception can be seen in Listing B.21 on Line 10, where the set is not cleared if the status of the transaction is not `COMMITTED`. Instead, the set is first cleared when the blocking behaviours of other transactions has been notified.

```
1  public Object doOrElse(ArrayList<IFn> fns) {
2    this.orElseRunning = true;
3    for (IFn fn : fns) {
4      try {
5        return fn.invoke();
6      } catch (RetryEx ex) {
7        // We ignore the exception to allow the next function to execute
8      }
9    }
10   this.orElseRunning = false;
11   throw retryex;
12 }
```

Listing B.22: The `doOrElse` Java method in the transaction `LockingTransaction` class

The `doOrElse` method seen in Listing B.22 takes a single argument `fns` that is an `ArrayList` of `IFn` instances. The method iterates through the list of `IFn` instances, and executes each function until one of them returns without trying to abort the transaction.

```
1  public void abort() throws AbortException{
2    stop(KILLED);
3    throw new AbortException();
4  }
```

Listing B.23: The `abort` Java method in the transaction `LockingTransaction` class

The `abort` method seen in Listing B.23 takes no arguments. The method stops the transaction on Line 2 and moves the continued execution right after the transaction by throwing an `AbortException` that is caught inside `LockingTransaction` and allows the `run` method to stop and the program to continue execution after the `dosync` block.

APPENDIX C

Dosync-ac Design Options

In the following section we will present and discuss different implementation options we had when implementing the `dosync-ac` macro. Modification of `Ref`s are not allowed in a After-commit block since it is not executed inside the transaction. It is possible to give the values of the `Ref`s to the After-commit block and thereby use the values of the `Ref`s in relation to side-effects. First the shared part is presented followed by the different options for the developers interface and their implementation.

```
1  (defn acarg
2    ([arg]
3     (if (= clojure.lang.Ref (type arg))
4       (.add &acargs (deref arg))
5       (.add &acargs arg)))
6    ([arg new-val]
7     (if (= clojure.lang.Ref (type arg))
8       (do
9         (.add &acargs (deref arg))
10        (ref-set arg new-val))
11      (throw (IllegalArgumentException. "cannot ref-set a non ref")))))
```

Listing C.1: The common `acarg` function

The shared part is a function called `acarg` that is used by option 1, 2 and 3 and can be seen in Listing C.1. The function is used to add specified values from the transaction to the After-commit block through an `ArrayList` called `&acargs`. The symbol `&acargs` is prefixed with a `&` to prevent it being hidden by the developers symbols by accident. This method is also used in the implementation of Clojure's own `defmacro` that is used to define macros. If the argument to the function is a `Ref`, the `Ref` is dereferenced and the dereferenced value is then used in the `after-commit` block. If a second argument is given to the function and the first argument was a `Ref`, then the value of the `Ref` is set to `new-val` afterwards. Below is the different implementation options of the `dosync-ac` macro represented with a brief description of the macro and helper functions, followed by a code example

with a subjective discussion.

Option 1

```
1 (defmacro dosync-ac-one [after-commit & dosync-args]
2   `(with-bindings {#'&acargs (java.util.ArrayList.)}
3      (dosync
4        (.clear &acargs)
5        ~@dosync-args)
6      (apply ~after-commit &acargs)))
```

Listing C.2: Option 1, with After-commit as a function

The implementation of Option 1 is shown in Listing C.2 which consists of the macro `dosync-ac-one` and the common helper function `acarg`. Dynamically scoped variables set by `with-bindings` are used in all the presented options because as we only experiment with different interfaces for a `after-commit` function and therefore not concerned with ensuring that the implementation is not leaking data. All the options presented here have a dynamically scoped `ArrayList` called `&acargs` which is used to store values. The `dosync-ac-one` requires that the code to be run after the transaction has committed must be in the form of a function and given as the first argument. The following arguments given to the `dosync-ac-one` macro will be executed as the body of the transaction, from the body of the transaction it is possible to store values in `&acargs` through `acarg`. The function given as the first argument that will be executed after the transaction has committed will have access to the stored values in `&acargs`.

```
1 (dosync-ac-one
2   (fn [& args] (println "after-commit:" args))
3   (acarg test-ref 1))
```

Listing C.3: An example of option 1

An example of Option 1 in use, can be seen in Listing C.2. This option makes it possible to specify the code that should be executed after the transaction has committed in a function as seen in Line 3. The following arguments defines the body of the transaction as normally when using `dosync`, the body of the transaction is defined in Line 3. The downside of this option is that the code specified is not in chronological order which might cause confusion for the developer.

Option 2

```
1 (defmacro dosync-ac-two [dosync-args after-commit]
2   `(with-bindings {#'&acargs (java.util.ArrayList.)}
3      (dosync
4        (.clear &acargs)
5        (~dosync-args))
6      (apply ~after-commit &acargs)))
```

Listing C.4: Option 2, with both `dosync` and After-commit as a function

The implementation of Option 2 is shown in Listing C.4 which consists of the macro `dosync-ac-two` and the common helper function `acarg`. The `dosync-ac-two` requires that the body of the transaction must be in the form of a function and must be given as the first argument. In this function it is possible to store values in `&acargs` through `acarg`. The second argument given to `dosync-ac-two` macro must also be in the from of a function and will be executed after the transaction have committed while having access to the stored values in `&acargs`.

```
1 (dosync-ac-two
2   #(acarg test-ref 2)
3   (fn [& args] (println "after-commit:" args)))
```

Listing C.5: An example of option 2

An example of Option 2 in use can be seen in Listing C.5. Option 2 makes it makes possible to specify a transaction as a function that can store values in `&acargs` as seen in Line 2. The second argument to `dosync-ac-two` are the function to execute after the transaction has committed and can be seen in Line 3, this function has access to `&acargs`. This gives a clear indication of which part of code is executed as a transaction and what is executed after the transaction while having the execution in chronological order which could make the code easier to read.

Option 3

The implementation of Option 3 is shown in Listing C.6 and consists of the macro `dosync-ac-three` and the common helper function `acarg`. The `dosync-ac-three` requires that the transaction must be in the form of a function and must be given as the first argument. In this function it is possible to store values in `&acargs` through `acarg`. The following arguments given the `dosync-ac-three` macro will be executed after the transaction have committed while having access to the stored values in `&acargs`.

```clojure
(defmacro dosync-ac-three [dosync-args & after-commit]
  `(with-bindings {#'&acargs (java.util.ArrayList.)}
     (dosync
       (.clear &acargs)
       (~dosync-args))
     ~@after-commit))
```

Listing C.6: Option 3, with `dosync` as a function

```clojure
(dosync-ac-three
  #(acarg test-ref 3)
  (println "after-commit:" &acargs))
```

Listing C.7: An example of option 3

An example of Option 3 in use, can be seen in Listing C.7. Option 3 it makes possible to specify a transaction as a function, which can be seen in Line 2. The expressions in the following arguments are executed after the transaction has committed while having access to `&acargs` as seen in Line 3. This makes it possible to write the code in chronological order which could help making the code easier to understand while not being forced to have the code that is executed after the transaction in the form of a function, like in Option 2.

Option 4

```clojure
(defn acfunc
  ([func & args]
   (if (fn? func)
     (.add &acargs [func args])
     (throw (IllegalArgumentException. "argument must be fn")))))

(defmacro dosync-ac-four [& dosync-args]
  `(with-bindings {#'&acargs (java.util.ArrayList.)}
     (dosync
       (.clear &acargs)
       ~@dosync-args)
     (doseq [acarg# &acargs]
       (apply (first acarg#) (second acarg#)))))
```

Listing C.8: Option 4, with After-commit as a function

The implementation of Option 4 is shown in Listing C.8 and consists of the macro `dosync-ac-four` and its helper function `acfunc`. The helper function `acfunc` makes it possible to add function calls to `&acargs` that will

be executed by `dosync-ac-four` after the transaction has committed. The functions are executed in the order they were added to `&acargs`.

```
1 (dosync-ac-four
2   (acfunc (fn [& args] (println "after-commit:" args)) (deref test-ref))
3   (alter test-ref + 1))
```

Listing C.9: Option 4, with After-commit as a function

An example of Option 4 is shown in Listing C.9. The helper function `acfunc` makes it possible to specify a function and arguments to be run after the transaction has committed. These arguments makes it possible to pass values from the transaction to the function that will be executed after the transaction has committed. This can be seen in Line 2 where the value of `test-ref` is passed to the After-commit function and will be used by the `println` function after the transaction has committed. The macro `dosync-ac-four` has the same syntax as the normal `dosync` which could increase the chances of a seamless integration.

Option 5

```
1  (defmacro acbody
2    [& body]
3    `(.add &acargs (fn [] ~@body)))
4
5  (defmacro dosync-ac-five [& dosync-args]
6    `(with-bindings {#'&acargs (java.util.ArrayList.)}
7       (dosync
8         (.clear &acargs)
9         ~@dosync-args)
10      (doseq [acarg# &acargs]
11        (acarg#))))
```

Listing C.10: Option 5, with After-commit taking a body

The implementation of Option 5 is shown in Listing C.10 and consists of the macro `dosync-ac-five` that uses the helper function `acbody`. The helper function `acbody` to makes it possible to add a block of expressions to be executed after the transaction has committed. The block of expressions are executed in the order they were added to `&acargs`.

```
1 (dosync-ac-five
2   (acbody (println "after-commit:" (deref test-ref)))
3   (alter test-ref + 1))
```

Listing C.11: Option 5, with After-commit taking a body

This option makes it possible to put `acbody` around a block of expressions with no added boilerplate code, as seen in Listing C.11. The macro `dosync-ac-five` of Option 5 shares the syntax of the normal `dosync` which could increase the chances of a seamless integration.

APPENDIX D

The Santa Claus Problem

The Santa Claus Problem is a known usability problem that have been used to evaluate languages as a means to make them comparable [23, 27]. The problem originates from "A new exercise in concurrency" [23] and is as follows.

> ”Santa Claus sleeps in his shop up at the North Pole, and can only be wakened by either all nine reindeer being back from their year long vacation on the beaches of some tropical island in the South Pacific, or by some elves who are having some difficulties making the toys. One elf’s problem is never serious enough to wake up Santa (otherwise, he may never get any sleep), so, the elves visit Santa in a group of three. When three elves are having their problems solved, any other elves wishing to visit Santa must wait for those elves to return. If Santa wakes up to find three elves waiting at his shop’s door, along with the last reindeer having come back from the tropics, Santa has decided that the elves can wait until after Christmas, because it is more important to get his sleigh ready as soon as possible. (It is assumed that the reindeer don’t want to leave the tropics, and therefore they stay there until the last possible moment. They might not even come back, but since Santa is footing the bill for their year in paradise ... This could also explain the quickness in their delivering of presents, since the reindeer can’t wait to get back to where it is warm.) The penalty for the last reindeer to arrive is that it must get Santa while the others wait in a warming hut before being harnessed to the sleigh.”

APPENDIX E

Project Summary

This project address a problem found in the STM implementation of Clojure, a composable and deadlock free alternative to locks for synchronising concurrent programs. The problem is that Clojure does not providing functionality for handling side-effects and does not allow manual control over transactions for synchronisation of threads. This lack of functionality was determined through an evaluation of both performance and usability of parallel functional programming languages and their concurrency models in our 9th semester project named "Performance and Usability Evaluation of Concurrency in Modern Functional Programming Languages".

In this project we have implemented constructs allowing the use of side-effects and transaction control in STM transactions into the programming language Clojure, the extended language we have dubbed dptClojure. The constructs were developed based on an extensive analysis of the Clojure runtime and STM implementation, and multiple experiments about how methods for handling side-effects in transactions could be performed. Each design for enabling side-effects in transactions were evaluated based on three minimal examples to demonstrate the specific problems that occur by combing STM and side-effects.

The experimental designs were implemented in Clojure's STM implementation, together with constructs for transaction control inspired from Haskell, to understand how such functionality could be enabled in Clojure's STM implementation. Based on the experimental design and their implementation a generic event handling system was developed and implemented in Clojure with minimal changes to the existing STM implementation, enabling the use of side-effects in transactions by ensuring the side-effects are executed only if a transaction is in a specific phase. Constructs for transaction control are added and extends those inspired by Haskell, allowing the developer to not only block a transaction based on transactional data, but also specify a function for determining the state of said data and check if the transaction should unblock.

We evaluated the capabilities of the event handling system using the three minimal examples, and we showed the event handling system could be

used to solve all problems represented by the examples. Transaction control was evaluated by demonstrating the effect of the added functionality. For the constructs that provided alternatives to existing Clojure functionality the constructs were evaluated using a usability evaluation based on the Santa Claus problem, with two metrics and a subjective discussion. This evaluation showed a decrease both in terms of Lines Of Code (LOC) and in terms of development time, and subjectively the added constructs helped to create a simpler implementation. To validate the results a larger usability evaluation and a performance evaluation are suggested as possible future work as well as an exploration of the concept of contexts in the event handling system.