Executable Formal Specifications with Clojure

Matti Nieminen

University of Tampere
School of Information Sciences
Computer Science
M.Sc. Thesis
Supervisor: Jyrki Nummenmaa
June 2015

University of Tampere
School of Information Sciences
Computer Science
Matti Nieminen: Executable Formal Specifications with Clojure
M.Sc. Thesis, 68 pages, 4 pages of appendices
June 2015

In software projects, where formal specifications are utilized, programmers usually need to know separate languages and tools for tasks related to programming and formal specifications. To remedy this situation, this thesis proposes a Clojure-based formal specification method consisting of a library and tool for writing and executing formal specifications.

The library and the tool are targeted for Clojure programmers: the library enables programmers to write formal specifications with Clojure, which allows the usage of the same language for formal specifications and the implementation. The tool, that is used together with the library, allows simulating the specifications by executing them. The method presented in this thesis does not aim for formal verification with mathematical proving. Instead, the goal of the method is to offer support for formal specifications without intimidating the developers.

The developed method eases the adoption of formal specifications in projects, where Clojure is used but formal specifications are still considered too costly to adopt; the library and the tool enable Clojure programmers to adopt formal specifications in their software projects without additional costs, as the language for the formal specification and the implementation is the same. The author's method also allows working iteratively from the specification to implementation because the models created with the author's library and tool can be transformed into implementation straightforwardly.

Keywords: Functional Programming, Clojure, Executable Formal Specifications

Table of Contents

1. Introduction

Formal specifications have gained a lot of interest in the academic world. Research has shown that adopting formal specifications is not particularly costly, and it improves the quality of software. Formal specifications are used in domains such as safety, security and transportation. However, even though formal specifications are being adopted steadily, most companies working outside of those domains have not done so at the time of writing this thesis. There are many reasons for that, but most of them are related to the fact that the required skills, languages and tools are different from what is required in traditional software development.

There has been some effort to solve this issue by bringing the process of writing formal specifications closer to programming. A formal specification method called DisCo is a clear example of this approach: the notation of DisCo is based on object-oriented programming, and the validation of DisCo specifications is done by animating the specification instead of performing formal verification. DisCo has utilized the popularity of object-oriented programming to offer a formal specification solution that feels familiar to programmers. In addition, validating DisCo specifications does not require any special skills in mathematics as the validation is not based on performing proofs or formal verification.

However, object-oriented programming is probably not the best programming paradigm to adopt for formal specifications. Imperative programming paradigms, object-oriented programming included, have their weaknesses, such as side-effects, which make it difficult to prove that programs written using these languages are working correctly. For this reason, there has been some interest in using functional programming languages for writing formal specifications. The results of this existing research indicates that functional programming languages are indeed closer to formal specifications due to their purity and mathematical foundation.

Functional programming paradigm has existed for a long time. However, its advantages have not been noticed in the industry until very recently. A new functional programming language called Clojure has gained a lot interest during the rediscovery of the paradigm itself. Clojure does not aim to be a pure functional programming language but instead focuses on being a practical language for general use. In addition, as Clojure is a dialect of Lisp, it is a good language for metaprogramming which allows programmers to apply it according to the needs of the application domain.

The purpose and the contribution of this thesis is to present the concept, implementation and usage of the author's custom-made formal specification method for

Clojure. The method consists of a library providing the necessary functions and macros for writing formal specifications. It also includes a web-based tool for editing and executing the specifications. The library and the tool embody the previously mentioned advantages of using functional programming languages for formal specifications: the specifications created with the library are executable and allow simulating the specified system. They can be also transformed into implementation using an iterative approach as the language used for both the specifications and the implementation is the same. These principles make it easy to adopt the author's method in situations where the developers would find existing formal specification methods too intimidating to adopt.

The library and the tool show that functional programming and formal specifications share many similarities by nature. However, the author will eventually come to the conclusion that although Clojure is an excellent language for implementing formal specification systems, it is not the best language for writing the specifications itself because of its typing mechanisms.

Chapter 2 is used to introduce functional programming and Clojure to the reader. Chapter 3 presents the idea and motivation behind formal specifications, and demonstrates the methodology in practice by showing some examples. Chapter 4 contains an analysis about the relationship between functional programming and formal specifications. Before conclusions, Chapter 5 presents the implementation and the usage of the author's library and tool. Some examples are also given to prove that the author's method is working correctly. Chapter 6 presents the conclusions of this thesis.

2. Functional programming and Clojure

To give a short but complete definition of functional programming is a difficult task. In order to understand functional programming it is necessary to learn at least one functional programming language and understand how the functional paradigm differs from the imperative programming paradigm. The way of solving problems with functional programming language is different from imperative programming languages [Fogus and Houser, 2011].

This chapter describes some common aspects of functional programming languages and the problems of imperative programming that functional programming helps to solve. Later in the thesis, these concepts become important when examining the relationship between different styles of software implementation and formal specifications.

The focus of the chapter then moves on to Clojure. Clojure is a functional programming language created for concurrency in mind and for providing a balance between practicality and pure functional programming [Halloway and Bedra, 2012]. In this thesis, Clojure is the language of choice for the implementation of a formal specification system.

2.1. About functional programming

Functional programming is a declarative programming paradigm in contrast to imperative programming paradigm. Applications built with functional programming languages consist mainly of functions and function calls. There are no assignment statements in pure functional programming [Butler, 1995].

Functions are *first-class* in functional programming languages. First-class entity is an entity that supports all operations generally available to other entities in the programming language. First-class functions therefore share the same properties that for example variables, being first-class, have in most programming languages. First-class functions can be created at runtime, stored in data structures and can return functions and take them as arguments [Fogus and Houser, 2011]. A function that takes another function as an argument or returns a function is called a *higher-order function*.

Assignment statements are used in imperative programming languages to create and update the value of variables. Traditionally, storing values to variables forms the *program state* of the application. The program state is a snapshot taken from the current values of every variable in the application at any specific time [Misra, 2001]. In pure functional programming languages, the program state is avoided almost completely [Hinsen, 2012].

The lack of assignment statements in functional programming languages is explained by the stateless nature of the paradigm: without state to store into the variables, there is no need for variables at all [Hinsen, 2009]. The advantages of stateless programming languages are discussed later in this chapter.

The stateless nature of functional programming leads to the usage of data that cannot change after its initialization. This kind of data is called *immutable data* [Halloway and Bedra, 2012]. As there are no objects, variables or references in pure functional programming, the immutable data concerns mostly data structures like lists, maps and sets. Since immutable data cannot be modified, the functions that alter or transform such data must initialize and return a new value to represent the changes [Hinsen, 2012]. For example, a function that takes a list of numbers as an argument and returns it without negative numbers must create a new list containing only the positive numbers from the input list.

Immutable data represents values in the real world [Halloway and Bedra, 2012]. For example, days of week like Monday and Tuesday are values. Current day of the week is a reference that can refer to a value like Monday. The reference to the value can change: eventually, current day of the week changes from Monday to Tuesday. Current day of the week represents state and acts like mutable data. The value inside the state is immutable and cannot change: Monday is not and never can be Tuesday.

Functional programming languages enforce the usage of immutable data because it is a natural combination with stateless programming model. Without objects that encapsulate references and variables, the functionality to manipulate state is not needed [Hinsen, 2009]. Despite this, immutable data is not characteristic only to functional programming: for example, it is possible to create immutable objects in object-oriented languages such as Java with classes that contain only private final instance variables, constructors, and getter methods. As the instance variables are final and the setter methods cannot exist, the objects initialized from such classes are immutable. An example of such a class is *DateTime* from Joda-Time [Joda.org, 2014]. Joda-Time is a Java library used to represent and handle dates, time and duration. According to the Joda-Time manual [2014], immutable objects like instances of *DateTime* are naturally thread-safe.

A function is called a *pure function* when it does not interact outside its scope and returns the same value every time it is called with the same arguments [Fogus and Houser, 2011]. Pure functions cannot perform operations such as read or write variables outside the function scope, print, access file system, draw to the screen or read user input. When a function does one of the mentioned operations or accesses the scope outside of it in other ways, the function is said to have *side-effects* and is therefore impure. Pure

functions are easier to write, maintain and test compared to impure functions as the programmers who work with pure functions do not have to consider any code outside the function itself [Halloway and Bedra, 2012].

Applications written in functional programming languages consist mostly of pure functions and immutable data [Emerick *et al.*, 2012]. The source code of most applications written in imperative programming languages shows that assignment statements generate side-effects which explains the impure nature of the imperative programming. Like immutable data, pure functions are not characteristic only to functional programming: nothing prevents writing pure functions with imperative programming languages, even though it is not natural to the paradigm.

Although pure functions have advantages over impure functions, they are needed in all applications. Without side-effects, applications can't do anything but silently return a value which is not what the applications are usually designed to do [Hinsen, 2009]. Due to the advantages of pure functions, Halloway and Bedra [2012] recommend that the amount of impure code should be kept to a minimum when programming with functional programming languages. To avoid tainting the trace of pure function calls with impurity, Halloway and Bedra [2012] also separate the impure code to its own layer in larger examples found in their book.

Pure functions are *referentially transparent* which means that a function call of certain arguments can be replaced with the return value of the function call without changing the behavior of the application [Fogus and Houser, 2011]. Expressions consisting only of pure function calls are referentially transparent as well. For example, the expression 3*2+2 uses two pure functions, sum and multiplication. The expression can be replaced with the result value of 8 without affecting the application containing the expression. In addition to sharing benefits with pure functions, referential transparency makes it possible to utilize *memoization* which means caching the results of the functions or expressions. Referential transparency also allows *parallelization*; as functions return the same value no matter what the application state is, it is possible to move the evaluation of the function call to another processor, machine or other environment [Emerick *et al.*, 2012]. It is possible but rare for an impure function to be referentially transparent [Halloway and Bedra, 2012].

The return value of a pure function or any referentially transparent expression does not change no matter what the application state is. Therefore, referentially transparent expressions, calls to pure functions included, do not depend on the time and lifecycle of the running application which makes it possible to *realize* the result of the expression only when it is needed [Läufer, 2009]. For example, the implementation of an application that sorts a list of numbers and prints *nth* element from the list can first sort the list in one

function and use the returned list in another (impure) function to print the result. As the list is immutable and the sort function is pure, the sorting can be realized just before printing the result. This is called *laziness* and is one of the features of functional programming. Lazy evaluation helps avoiding unnecessary evaluation of expressions and enables the use of infinite data structures as their creation does not result in their complete evaluation [Hinsen, 2009]. Instead, only the required data will be realized on demand. Lazy evaluation can utilize more resources, because it is mandatory to keep track of the expressions ready to be realized. This is shown as a higher memory consumption in the applications written in lazy functional programming languages [Hinsen, 2012]. On the other hand, laziness helps to save resources as unnecessary computations are often avoided.

Functional programming uses *recursion* over traditional looping. Looping requires either a loop variable or some other kind of state variable to exit the loop and accumulate the results [Fogus and Houser, 2011]. The lack of variables therefore makes loops not only useless, but impossible to implement in pure functional programming languages. Instead of utilizing loops, functions recursively call themselves with different arguments to accumulate the results [Hinsen, 2009]. For example, a function returning the *nth* Fibonacci number *fib(n)* is often implemented using recursion similar to mathematics. Function call *fib(0)* returns 0, *fib(1)* returns 1 and otherwise *fib(n)* returns *fib(n-1)* + *fib(n-2)* [Lipovača, 2011]. This implementation of *fib* calls itself in case the given argument is bigger than one.

2.2. The problems of imperative, procedural and object-oriented programming paradigms

Applications written in imperative or procedural programming languages consist mostly of assignment statements which is clearly different from the pure nature of functional programming. As previously argued, assignment statements lead to side-effects and impure code which does not allow features of functional programming such as laziness, memoization or parallelization.

Side-effects require programmers to consider the control flow of the whole program instead of focusing on the arguments and the return value of the function. For this reason, developing software with side-effects leads to bugs more easily compared to developing programs with pure functions [Emerick *et al.*, 2012]. Fortunately, these problems can be managed in single-threaded applications. However, combining side-effects with multi-threaded programming, which is usually done with complicated locking, could result in bugs that are difficult to find and reproduce [Fogus and Houser, 2011].

Functional programming languages are less verbose and more expressive than imperative and procedural languages due to the lack of the assignment statements and

higher-order functions [Halloway and Bedra, 2012; Lipovača, 2011]. Therefore, applications written in functional programming languages are usually shorter than their imperative versions. According to Hughes [1989], this does not mean that programmers using functional programming languages are more productive than the programmers using imperative and procedural languages. Productivity is obviously a difficult subject as Halloway and Bedra [2012] argue in contrast to Hughes [1989] that applications with less code are cheaper to develop.

Modularity and reusability are important features in any programming language and paradigm. Object-oriented programming was created to help the programmers to design applications that use classes and objects to represent real-world entities [Lewis and Loftus, 2011]. In addition, object-oriented programming was said to solve the modularity issues by encapsulating the state and operations to modular classes. Similarly, reusability issues were supposed to be solved with class inheritance [Schach, 2010]. As the object-oriented programming is now the most used programming paradigm in the industry, it is logical to presume that object-oriented programming made creation of modular and reusable components easier. However, although scientific literature does not reveal it, object-oriented programming is being criticized by many influential people. Inventor of Erlang programming language Joe Armstrong has said in *Coders at Work* [Seibel, 2009] that "the problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle". Computer scientist Luca Cardelli [1996] wrote an article that criticized object-oriented programming by stating that initially the good design principles of object-oriented programming have evolved into more complex versions compared to what is needed. The creator of Clojure, Rich Hickey, said in his keynote presentation *Are we there yet?* [2009] that although classes represent real-world entities, they do not adequately model the time related to those entities causing difficulties when implementing concurrency. At least according to the study by Potok *et al*. [1999], object-oriented programming does not seem to have an effect on programmer productivity.

The statement of Joe Armstrong presented above is what the author would like to elaborate. Object-oriented programming has tight coupling inside the classes itself: even if the classes would not have any dependencies to other classes or the dependencies would be loosely coupled, the data as instance variables is encapsulated and coupled with the *methods* that manipulate the data [Lewis and Loftus, 2011]. In addition, the side-effects and the complexity that comes with them are present in object-oriented programming just as much as they are in other imperative programming paradigms.

Finally, the author of this thesis would like to point out that the productivity of the different programming paradigms is hard if not an impossible task to compare. Opinions, preference, expertise of the developers, chosen programming languages, tools and the software project itself most likely affect the results of the productivity studies more than the language itself which makes a single study unreliable. Also, the references to the productivity claims used in this thesis are over ten years old. The point of these remarks is just to remind the reader that the community of software professionals is not unanimous about the productiveness of different programming paradigms.

2.3. Clojure and Lisp

The author has chosen to give all of the code presented in this thesis in Clojure. Clojure is a functional programming language and therefore utilizes the previously presented features of functional programming such as first-class functions, higher-order functions, pure functions and laziness. Clojure was released in 2007, and as a modern language it is designed for the current trends of its time; Clojure is a general-purpose programming language but it also includes several mechanisms to deal with concurrency which helps the programmers to utilize modern multi-core processors [Hickey, 2014].

Clojure is a dialect of *Lisp* [Halloway and Bedra, 2012], the first functional programming language implemented in 1958 and presented in 1960 by John McCarthy. Lisp utilizes the polish notation to apply functions to arguments where function calls are written before their arguments inside the same parentheses [McCarthy, 1960]. For example, the algebraic notation *5(2+1)* is written in Lisp as *(* 5 (+ 2 1))*.

Lisp uses parentheses to express lists and lists for calling functions and applying arguments to them [Emerick *et al.*, 2012]. This means that the code itself is written using lists. For example, the expression *(+ 3 1)* is both the application of sum function to two integers and a list of three elements: the sum function, number three and number one. Lisp is therefore *homoiconic* which is to say that the language itself is composed of the same structures it manipulates [Fogus and Houser, 2011]. This is often shortened to phrases such as "code is data" and "syntax is structure".

McCarthy managed to define Lisp using only seven functions and two special forms, making the amount of syntax in Lisp small [McCarthy, 1960]. Still, Lisp is often perceived as an intimidating language by both the beginner and experienced programmers as it looks different from any other programming language due to the use of the parentheses and the prefix notation [Emerick *et al.*, 2012]. Lisp has evolved over the years and now has dialects for different kinds of programming paradigms, including the object-oriented programming [Henderson, 1986].

Clojure tries to improve Lisp in this sense by avoiding nested parentheses in places where they are not necessarily required. In addition, in Clojure lists are often replaced with other data structures in certain forms to enhance the readability. For example, the arguments of a function are presented inside of a vector instead of a list. To visually separate the arguments from the rest of the function, Clojure uses brackets to express vectors instead of parentheses [Halloway and Bedra, 2012]. Code example 1 shows an example function that demonstrates the prefix notation, syntax and the vector arguments of Clojure functions.

```
(defn hello-with-function
  "Returns 'Hello ' with hello-subject appended to it
   and function f applied to the whole string."
  [hello-subject f]
  (f (str "Hello " hello-subject)))

=> (hello-with-function "world" clojure.string/upper-case)
"HELLO WORLD"
```

Code example 1: A higher-order function written in Clojure.

The function in Code example 1 takes two arguments where the latter is a function. Function *hello-with-function* first creates a string with the first argument appended to the string *"Hello "* and then applies the function *f* to that string. As the function *hello-with-function* accepts a function as an argument, it is a higher-order function.

Clojure, like any Lisp, has powerful metaprogramming capabilities through macros [Halloway and Bedra, 2012]. Macros are pieces of code that are not evaluated like normal expressions. With normal functions, Clojure *reader* reads the function calls before they are evaluated whereas macros are called by the *compiler* before the evaluation happens. Macros return data structures including functions and function calls that are ready for evaluation like normal expressions [Emerick *et al.*, 2012]. Macros take advantage of the homoiconicity of Lisp which allows programs to write other programs and programmers to add new features to the language itself. This makes Clojure and other Lisps great programming languages for implementing domain-specific languages [Fogus and Houser, 2011]. This is an important trait later in this thesis when implementing the formal specifications library.

Clojure is a language for *Java Virtual Machine* which ensues the tight interoperability with Java components. Clojure code compiles to the same bytecode that Java components compile to. It is possible to initialize objects from Java classes, access their fields and

invoke their methods directly from the Clojure code [Halloway and Bedra, 2012]. It is also possible to extend an existing Java class or implement an existing interface with Clojure [Hickey, 2014]. However, as a functional language, Clojure does not allow object-oriented design in new applications written in Clojure. The function in Code example 2 demonstrates the Java interoperability by using a Java class *Random* to return random integers.

```
(defn random-number
  []
  (. (new java.util.Random) nextInt))
```

Code example 2: The interoperability with Java in Clojure.

The *new* special form creates a new object much like the *new* keyword in Java does. The . (dot) special form either gets a field or invokes a method of the object similarly to Java. The difference comes from the prefix notation: the same expression is written in Java as *new Random().nextInt();*

Unlike Java, Clojure is a dynamically typed language. Clojure utilizes *duck typing* which means that the type of an object is determined by the properties of that object [Halloway and Bedra, 2012]. For example, the function in Code example 3 does not care about the type of data it receives as long as it has *:name* and *:age*.

```
(defn greeting
  [x]
  "Returns a greeting using the name and age of the argument."
  (str "My name is "(:name x) " and I'm " (:age x) " years old"))

=> (greeting {:name "Joe" :age 3})
"My name is Joe and I'm 3 years old"
```

Code example 3: A function demonstrating the dynamic typing of Clojure.

The return value of the function in Code example 3 is always a string, but it could vary just like the type of the argument can. The argument used in the example upon calling the function is a map of key-value pairs which is expressed using curly brackets in Clojure.

Clojure programmers usually use *REPL* (*read-eval-print loop*) to experiment during the development process. REPL is a tool that provides a prompt for interactive programming. The REPL repeatedly waits for the user to type a Clojure expression for evaluation, after

which it will print the result back to the user [Fogus and Houser, 2011]. REPL gives an immediate feedback to the user which makes it a good tool for experimenting with Clojure. The author of this thesis invites the reader to install a tool called *Leiningen* which can be used to install Clojure and start the REPL [Hagelberg, 2014]. Doing so, the reader is able to try out the code examples presented in this thesis.

2.4. Between practicality and pure functional programming

Clojure is a functional programming language with dynamic typing that encourages the use of immutable data. However, Clojure is not a pure functional language like for example statically typed Haskell is [Halloway and Bedra, 2012]. At the time of writing, according to TIOBE [2014] Haskell is more popular than Clojure but less popular than Lisp. The experiences of the author and the literature review suggest that functional programming, regardless of the specific language, is rarely used in the industry. This chapter aims to present how Clojure sacrifices its purity to bring practicality to functional programming in order to gain the interest of the industry.

Pure functional programming languages like Haskell allow side-effects only inside of certain language constructs [Hinsen, 2009]. In Clojure, every form can contain side-effects. For example, if a function body has multiple forms, the result of the last form will be the one that is returned. However, all of them will be evaluated. In cases where only a single form is allowed, it is possible to use the *do* special form which takes a variable amount of forms as arguments, evaluates them all and returns the result of the last of them [Hickey, 2014]. A function with side-effects and the usage of *do* special form is shown in Code example 4.

```
(defn impure-function
  "First prints Hello World, then information about given arguments.
   Returns the argument. Uses do special form."
  [x y]
  (println "The arguments are" x "and" y)
  (if (= x y) (do (println "The arguments are the same.") true) false))

=> (impure-function 1 1)
The arguments are 1 and 1
The arguments are the same.
true
```

Code example 4: A function demonstrating side-effects with and without *do* special form.

In addition to allowing side-effects, Clojure differs from pure functional programming by offering functionality to manipulate shared state. Clojure has four major mutable references with different use cases and APIs which are presented in Table 1.

Agent	Uncoordinated, asynchronous: used for example for controlling I/O when features of the other reference types are not needed.
Atom	Uncoordinated, synchronous: for situations where a single value is required that can be read and swapped with another value.
Ref	Coordinated, synchronous, retriable: safe access to multiple refs is guaranteed without race conditions. Utilizes software transactional memory.
Var	Provides thread-local state by isolating the state to the current thread.

Table 1: Four major reference types in Clojure [Fogus and Houser, 2011].

Clojure utilizes *software transactional memory* instead of locks to protect the shared state created with refs. Clojure's software transactional memory uses *multiversion concurrency control* which functions by creating an *isolated snapshot* of the required references every time a transaction occurs [Fogus and Houser, 2011]. The transactions access the references only in the isolated scope until the transaction is ready to expose the changes to the rest of the application. Upon commit, the references are checked for changes that may have happened during the transaction. If some other transaction has already changed the values of the references, the transaction updates the snapshot with the updated values and restarts. Otherwise the transaction commits [Bernstein and Goodman, 1983]. Later in this thesis STM proves to be an important feature as it can guarantee that the execution of a formal specification cannot lead to unwanted state.

Different reference types have different APIs for changing the contents of the reference. For example, the coordinated nature of refs requires transactions whereas atoms can be changed with a simple function call due to their uncoordinated nature [Halloway and Bedra, 2012]. Atoms and refs also share some similarities: function *deref* is used to *dereference* them which means reading the contents of the reference. For convenience, there is also a reader macro @ for *deref*. Code example 5 presents the API and basic usage of refs.

```
=> (def current-day-of-the-week (ref "monday"))
#'user/current-day-of-the-week
=> current-day-of-the-week
#<Ref@5054d877: "monday">
```

```
=> @current-day-of-the-week
"monday"
=> (ref-set current-day-of-the-week "tuesday")
IllegalStateException No transaction running  clojure.lang.LockingTransaction.getEx
(LockingTransaction.java:208)
=> (dosync (ref-set current-day-of-the-week "tuesday"))
"tuesday"
=> @current-day-of-the-week
"tuesday"
```

Code example 5: Basics of using refs in Clojure.

At the first line of Code example 5, a ref is created with a function called *ref*. The ref is accessed using a var which is created using a *def* special form. Evaluating the var without dereferencing it returns the ref itself, and using the @ reader macro returns the contents of the ref. In this example, *ref-set* function is used to change the contents of the ref which throws an exception if it is not called inside of a transaction. The transaction is started with a *dosync* macro which runs the expressions given as arguments inside the transaction [Hickey, 2015a]. In addition to *ref-set*, there are other functions such as the higher-order function *alter* that can be used to change the contents of the ref more conveniently in some situations.

Besides freeing programmers from the use of locks, Clojure's design for concurrency has other advantages as well. Updates to the shared state satisfy ACI of the *ACID* as updates are *atomic*, *consistent* and *isolated* much like in relational databases [Emerick *et al.*, 2012]. In addition, nondeterministic deadlocks are not possible unless programmers utilize Clojure's interoperability with Java to create threads and locks. Instead of deadlocks, Clojure applications can get to a *livelock:* a transaction may never finish as it continuously restarts due to other transactions blocking the commit by modifying the references that the transaction requires [Fogus and Houser, 2011].

In Clojure, the evaluation of the functions and expressions are generally not lazy operations. However, most of the benefits of laziness are available in Clojure because a big part of programming with Clojure is working with s*equences.* Sequence is an abstraction for data structures in Clojure. All Clojure and Java collections are *seq-able* which means that they are lazy and can be processed with the *sequence library*. Sequence library is a library for Clojure that provides a common API for different types of sequences. Sequence library consists of functions that for example create, filter or transform sequences. It also

offers solutions to many problems that are usually solved with loops in imperative programming languages [Halloway and Bedra, 2012].

Unlike in Haskell or in other functional programming languages, direct recursion should be avoided in Clojure [Halloway and Bedra, 2012]. Because Java Virtual Machine does not support *tail call optimization*, each recursive function call consumes a *stack frame* in the virtual machine. Recursive functions work fine when the amount of recursive calls is small. However, larger amounts cause the function to eventually throw an exception. Clojure has special forms *loop* and *recur* which are used to create constructs similar to loops without consuming the stack [Hickey, 2014]. Code example 6 presents three functions each with a different solution to returning a sum of *n* random numbers using the function *random-number* presented in Code example 2.

```
; This should not be done.
(defn faulty-sum-of-n-random-numbers
  "Returns a sum of n random numbers using direct recursion."
  [n]
  (if (zero? n)
    0
    (+ (random-number) (faulty-sum-of-n-random-numbers (dec n)))))

=> (faulty-sum-of-n-random-numbers 100000)
StackOverflowError   java.util.Random.<init> (:-1)

; This works but is unnecessarily complicated.
(defn working-sum-of-n-random-numbers
  "Returns a sum of n random numbers using loop and recur."
  [n]
  (loop [times n, sum 0]
    (if (zero? times)
      sum
      (recur (dec times) (+ sum (random-number))))))

=> (working-sum-of-n-random-numbers 100000)
-214366413332

; The best way is to use the functions from the sequence library.
```

```
; It is possible that even a simpler solution exists.
(defn correct-sum-of-n-random-numbers
  "Returns a sum of n random numbers using the sequence library."
  [n]
  (reduce + (take n (repeatedly random-number))))

=> (correct-sum-of-n-random-numbers 100000)
-173966581446
```

Code example 6: Three functions returning sum of n random numbers.

The function *faulty-sum-of-random-numbers* works fine with small numbers but throws an exception when the amount of calls in the stack grows too big for the virtual machine to handle. The second function *working-sum-of-n-random-numbers* works correctly and does not throw any exceptions as it is implemented using the *loop* and *recur* special forms. The final function *correct-sum-of-n-random-numbers* shows that the simplest solution is often found directly from the sequence library.

3. Formal specifications

Formal methods in software engineering are methods that follow mathematical principles and presentation techniques. These methods can be used for example for analyzing specifications and verifying applications [Sommerville, 2009]. Being part of the formal methodology, the focus of this thesis and especially this chapter is in the *formal specifications*.

This chapter presents the rationale and core principles of formal specifications. Some probable causes for why the adoption of formal specifications has been slow in certain domains is presented as well. The chapter then moves on to explaining the advantages of validating the specification by executing it instead of performing proofs. Finally, this chapter presents examples of simple formal specifications given in different formal specification languages.

It is important to note that there are many specification languages, tools and techniques in addition to those discussed in this chapter. The author merely aims to present the general ideas behind formal specifications and give out some specific examples which help the reader to understand the design of the author's formal specification method presented later in this thesis.

3.1. About formal specifications

Formal specification is a specification of a software system written in a language that has formally defined vocabulary, syntax and semantics. Formal languages are therefore rigorous: they are precise, unambiguous and do not leave room for interpretation like natural languages do [Sommerville, 2009]. In that sense, formal specification languages are similar to programming languages which are formal as well. The formal nature of the language is based on discrete mathematics and uses concepts from algebra, logic and set theory [Lightfoot, 2001]. Like other types of software specifications, formal specifications describe the properties of the system and operations related to it [Lamsweerde, 2000].

In addition to having a precise syntax, formal languages also have precise semantics and proof theory. These features make it possible for computers to automatically analyze, execute and otherwise manipulate formal specifications. Lamsweerde [2000] summarizes different uses for automated manipulation of formal specifications such as generating test cases or confirming that the specification satisfies the expectations for the system. In the scope of this thesis, the most interesting capability of formal specifications is the ability to animate, execute or simulate the specification.

Like all software specifications, formal specifications can be used for example to design, document and communicate software system requirements. In addition, the formality helps in writing of higher quality specifications: the precise rules of the language do not allow the ambiguity which is often the problem in specifications written in natural languages [Lamsweerde, 2000].

As formal specifications are difficult to understand for everyone else but the experts of the field, formal specifications are often used together with informal specifications written in natural languages [Palshikar, 2001]. This approach is similar to programming where natural languages are used to document complex parts of the formal notation.

Usually, partly because of the required expertise, formal specification of a system contains only the areas that need clarification [Sommerville, 2009]. In addition to containing only a part of the system or its properties, the formal specification usually has some level of abstraction [Lamsweerde, 2000]. For example, the specification may include the system with its environment and users, just the software system or just the user interface.

Formal specifications can come in to play in almost any phase of the software development project. The usual case is to write the formal specification after the functional model has been designed. However, formal specifications can be used earlier for example when elaborating the goals or eliciting the requirements of the software system. Formal specifications can also be utilized later in the requirement management phase to validate the changes of the requirements before implementing them. This is helpful as implementing changes becomes even more costly if the changes were not legitimate [Lamsweerde, 2000].

Formal specification languages can be categorized in different ways. Again, similar to programming languages, there are multiple formal specification languages with different styles and paradigms. There are languages like Z, VDM and B based on the state that the application may have. There are also languages based on state transitions which focus on the control flow and concurrency issues of the system. An example of such a language is CSP and its derivative FSP which will be quickly demonstrated with an example later in this chapter. There are of course many other paradigms and languages which are not mentioned in this thesis but can be found in the paper written by Lamsweerde [2000].

Sommerville [2009] uses another kind of categorization to divide the formal specification languages into two groups called *algebraic approach* and *model-based approach*. The first one is based on defining abstract data types with the operations that are related to the types and their relationships. The algebraic approach includes languages such as Larch, OBJ and Lotos. The model-based approach is based on expressing the specification

in a system state model using constructs such as sets and sequences. In addition, operations that alter the system state are defined. Model-based approach includes probably the most known formal specification language Z. It also includes languages like VDM, B, Petri nets and CSP. Sommerville [2009] argues that algebraic approach is more difficult to understand which may be one of the reasons why model-based approach has been adopted more widely in the software industry.

3.2. Motivation behind formal methods

The previous chapter explained the basic rationale of using formal specifications: developing software is a complex process and it is difficult to clarify what is actually required from a system to be developed. Even after the requirements are elicited, developing software that is guaranteed to satisfy the requirements is a difficult task [Sanders and Johnson, 1990b]. Formal specifications help to avoid the ambiguity by providing precision. This chapter aims to expand that idea and introduce more benefits of formal specifications.

The advantages discussed in this chapter have been proven in multiple research papers. For example, in a study presented in the paper by Pfleeger and Hatton [1997], software quality was studied using a set of components that had been developed with different levels of utilization of formal specifications. Their study was inconclusive but pointed out that the components whose developers had used formal specifications together with other quality assurance related methods, such as unit testing, produced more reliable code than the other teams. This particular study suggests that testing is more efficient in projects that have utilized formal specifications because possibilities to make an error are more apparent compared to the projects that have utilized just informal specifications.

One of the main advantages of formal specifications is that they force the development team to deeply analyze the requirements and the informal specifications. This process often leads to finding ambiguity, inconsistencies and errors [Sommerville, 2009]. As it becomes more expensive to fix the requirement and design related errors and inadequacies during the implementation or testing, it is essential to detect these problems as early as possible [Lightfoot, 2001]. For the same reason, writing formal specifications about the components that would be costly to refactor is often a good idea [Palshikar, 2001].

As the cost of detecting an error later in the project is high, formal specifications are best utilized during the early phases of the project. Formal specifications reduce the costs of the software project as ambiguities are found during the requirements specification phase [Sanders and Johnson, 1990b]. This effect is indirect as using formal specifications

does not fix the errors that have been made but makes them visible for the developers [Hall, 1998]. Even though it is challenging, the customer and the end users should be involved in the process to maximize the advantages of the formal specifications.

Formal specifications can be used as the means of communication between stakeholders. Users can get to know what they are going to get after the delivery, and the developers will know what to design and implement. Formal specifications can also be used for testing the implemented system as they serve as a reference to what the system should do [Hall, 1998]. Tools, that can either execute the specification or provide visualization based on it, will help in involving the non-technical stakeholders during the review of the specification as they partially eliminate the need to know the mathematical notation.

It is possible to use formal specifications to analyze the set of all possible states in a software system. This analysis is usually performed to find out if a formal specification violates properties such as *fairness*, *progress, liveness* or *safety*. The fairness of a certain state means that it is not possible to systematically omit that state every time it becomes available [Kurki-Suonio, 2005]. This is closely related to the progress property which means that a certain wanted state will be eventually reached. Progress property is violated if the set of possible system states contains a subset of states from where it is impossible to find a transition to the wanted state [Magee and Kramer, 2006]. Progress is associated with liveness which is a property that states that something good will eventually happen [Kurki-Suonio, 2005]. The last mentioned property, safety, is satisfied when the set of possible states does not contain any states that should not ever happen [Kurki-Suonio, 2005]. The safety property can be communicated informally by stating that nothing bad will happen in the execution of the system [Magee and Kramer, 2006]. Some formal specification notations, such as FSP, allow the user to define the set of states that violate these properties [Magee and Kramer, 2006]. When this is done, it is possible to check whether or not the formal specification violates these properties.

Concurrency is considered one of the most difficult things in the implementation of software systems. Doing multiple things at the same time while sharing resources and state can lead to *deadlocks*, *livelocks* and *race conditions*. Formal specifications can bring clarity to the concurrency aspects as well [Hall, 1998]. The previously mentioned formal specification languages CSP and FSP are designed especially for modeling concurrency. In fact, FSP notation is used in the book *Concurrency: State Models & Java Programs* by Magee and Kramer [2006] for demonstrating the pitfalls of multi-threaded programming and their solutions in Java.

Formal languages may have additional benefits in terms of structuring the specification. Like programming languages, formal specification languages may have characteristics, features or syntax that allows better structuring compared to the natural languages [Lamsweerde, 2000]. This is particularly helpful when the amount of properties or requirements to communicate with the specification is large.

Different formal specification languages and tools have each their own advantages. For example, if it is required to execute the specification, an *executable formal specification* system is needed. Executable specifications may lack in expressiveness which is an important trait in some other use cases of formal specifications. Therefore, different formal languages should be used in different cases depending on the suitability of the notation and the tools [Hall, 1998]. Executable formal specifications are discussed later in this chapter.

The rigorous nature of formal specifications allows analysis using tools. There are different kinds of tools available such as theorem provers, syntax checking editors, or simulation or execution tools. There is a lot of discussion in the literature about tools that can or could generate the implementation or test cases from the formal specifications [Palshikar, 2001]. However, creation of the formal specification is a valuable task by itself: although the written specification and the tools are important, the process of writing the specification with a formal language brings clarity to the developers even without conducting an analysis [Hall, 1998].

3.3. Current trends and usage in practice

Many researchers predicted in the 1980s that after the turn of the millennium formal specifications along with the other formal methods would be largely adopted by the software industry. Formal methods were thought to play the key role in improving the software quality [Sommerville, 2009]. Reading research papers with encouraging results and success stories from different companies using formal methods is confusing: if formal specifications are truly such a great way to improve software quality, why their adoption has been slow in other domains except where safety and security are critical aspects?

Formal specifications are used in safety- and security-critical applications because there is a lot of pressure to invest in the software quality in those specific domains [Sanders and Johnson, 1990b]. Transportation is a good example of such domain: formal specifications have been used in railway, aerospace and aviation systems [Palshikar, 2001]. In other domains, formal specifications have not been adopted widely in practice. This is partly because the quality of software systems has improved in other ways that were not thought of in the 1980s. In addition, focus of the development has shifted from quality to

time: today, rapid software development and fast delivery is considered more important than trouble-free software [Sommerville, 2009].

As it is expensive to fix the requirement or design related problems during the later phases of the development or after the delivery, formal specifications have been traditionally used together with waterfall-like development processes. Waterfall has a planning phase before the implementation begins where the main advantages of the formal specifications are leveraged naturally. Using the formal specifications in this manner front-loads the costs and effort of the software project [Sommerville, 2009].

The author would like to point out that the agile software development was not invented in the 1980s, the golden age of formal specifications, which may explain why the formal specifications are often discussed together with the waterfall development process. Compatibility between agile software development and formal specifications is an interesting research topic but outside the scope of this thesis.

Traditionally, adopting formal specifications was considered expensive and difficult in the industry. Although this is no longer the case, adopting formal specifications leads to some training costs as the mathematical notation must be taught to the developers. However, the bigger reasons for slow adoption of formal specifications is the lack of tools [Palshikar, 2001]. The adoption of formal specifications is not the main issue of this thesis, although the creation of a formal specification tool for programmers may partially help to solve problems related to industry adoption.

As formal specifications are not part of any popular development process, the chosen process must be modified to include the usage of formal specifications. It is not usually mandatory to create formal specifications for the whole system and it may not be necessary to utilize the advanced features such as theorem proving at all. Instead, it is important to identify the interested stakeholders, purpose and scope for the formal specifications. Decisions about choosing the correct language and tools are affected by questions such as who will write the formal specification, and for whom and for what purpose it is for. Using multiple notations in the same project is not out of question [Zave and Jackson, 1996]. For example, as functional requirements describe the application state and its manipulation, developers may use a notation such as Z to describe those aspects as it supports them well. Languages such as CSP and FSP could be then used for specifications that are related to concurrency, control flow or transitions of the application.

One way to utilize formal specifications is to involve the customer directly in the early phase of creating the formal specification. To make it less difficult for the customer to understand it, the first version should be simple and focused on the end user requirements. Sommeville [2009] argues that the final version of the formal specification

should be mainly created for the needs of the development team without specifying details about the implementation.

Formal specifications should be readable, structured well, valid and consistent with the requirements. They answer to the question what the application does, not how it does it or how it will be implemented [Palshikar, 2001]. The specification should be written according to the natural structure of the requirements and domain, not according to the architecture or the design of the application. The author would like to point out that even though unnecessary information about the software design and implementation is often presented as a problem in formal specifications, it is just as big of a problem in informal specifications.

3.4. Executable formal specifications

Formal specifications can be used for verifying the correctness and legitimacy of the system that is going be implemented. One way to do this is by performing proofs. By proving, it is possible to verify some properties or consequences of the system and to make sure that nothing undesirable can happen during the execution of the system. In addition, proofs can provide valuable information for validating that the finished application satisfies the specification [Gaudel, 1994]. Depending on the situation, proofs can be conducted by hand or by using formal theorem provers.

However, performing proofs is a difficult task as it requires a lot of skills in mathematics [Sanders and Johnson, 1990b]. An alternative approach is to use a formal language and tools that allow executing the specification. Execution can be used to validate the correct behavior and adequacy of the specification, and to illustrate the desired or undesired features [Lamsweerde, 2000]. Executable formal specification works as a prototype which together with human reasoning is enough to strengthen the confidence of the stakeholders to the legitimacy of the specification [Palshikar, 2001]. The ability to execute the formal specification can also be used to communicate the formal specification to non-technical stakeholders, such as the customer or end users.

In order to execute the formal specification, the language must have a tool that is able to read and evaluate the specification. Like there are theorem provers that can be used for formal verification, there are animators that can be used to execute the specification [Gaudel, 1994].

To avoid misunderstanding, the author wishes to emphasize that validating a formal specification by execution is not a formal verification technique [Sanghavi, 2010]. Executable formal specifications merely make it possible to animate the specifications and simulate the applications that will be implemented. Finding issues by executing the specification is similar to investigative testing used in the traditional software

development: not every possible state will be systematically checked, but instead it is possible to manually check situations that are known to be problematic in the specification. Animating or simulating the specification and formal verification techniques are not mutually exclusive methods as they complement each other in validating the formal specifications. However, they certainly have different use cases and targeted audiences.

3.5. Examples

As mentioned earlier, this sub-chapter demonstrates formal specification languages in practice by giving the same specification in three different notations: FSP, Z and DisCo. These examples provide the reader an understanding about how formal specification languages work which is important in order to understand the author's Clojure-based solution presented later in this thesis. In order to save the time of the reader, the chosen example is simple. The imaginary specification consists of a simple bank account that has a balance and operations for withdrawal and deposition.

3.5.1. FSP

FSP is an acronym for *Finite State Process*. It is a notation based on Tony Hoare's *CSP* (*Communicating sequential processes*). FSP specifications are used to produce finite *Labelled Transition Systems* (*LTS*) which basically means systems that have a finite number of possible states [Magee, 1997]. FSP notation is used together with LTSA, the *Labelled Transition System Analyzer*. LTSA can execute the specification and analyze its safety, liveness, progress and errors [Magee and Kramer, 2006]. The formal specification of the bank account example mentioned earlier in this chapter is presented using the FSP notation in Formal specification example 1.

```
const MAX = 10
range RANGE = 0..MAX

ACCOUNT = ACCOUNT[0],

ACCOUNT[balance:RANGE] = (account_has_balance[balance] ->
    (withdraw -> WITHDRAW[balance]
    |deposit -> DEPOSIT[balance])),

WITHDRAW[balance:RANGE] = (amount[amount:RANGE] ->
    (when(balance>=amount) success -> ACCOUNT[balance-amount]
    |when(balance<amount) not_enough_money -> ACCOUNT[balance])),
```

```
DEPOSIT[balance:RANGE] = (amount[amount:RANGE] ->
    (when(balance+amount<=MAX) success -> ACCOUNT[balance+amount]
    |when(balance+amount>amount) range_exceeded -> ACCOUNT[balance])).
```

Formal specification example 1: The bank account example in FSP notation.

FSP specifications are built of processes and actions that may be parameterized. The first two lines in Formal specification example 1 are used to declare a range of integers between zero to ten. Then, a process called *ACCOUNT* is declared. *ACCOUNT* does nothing but direct the execution to the parameterized version of the *ACCOUNT* process. The parameterized version has an action called *account_has_balance* which is parameterized to include the balance of the account. From there, the user can choose to continue either to the *WITHDRAW* or the *DEPOSIT* process.

WITHDRAW and *DEPOSIT* are similar processes. They both start with a parameterized action called *amount*. This action is used to choose the amount that is either withdrawn from the account or deposited into it. After that action, both processes state some conditions that must be satisfied for the next actions to occur. For *WITHDRAW*, *success* action is reached only if the account has enough money. *DEPOSIT* process leads to *success* only if the balance does not exceed the range set in the second line of the example. After *success*, both processes return to the parameterized version of the *ACCOUNT* process which restarts the process of choosing between withdrawal and deposition.

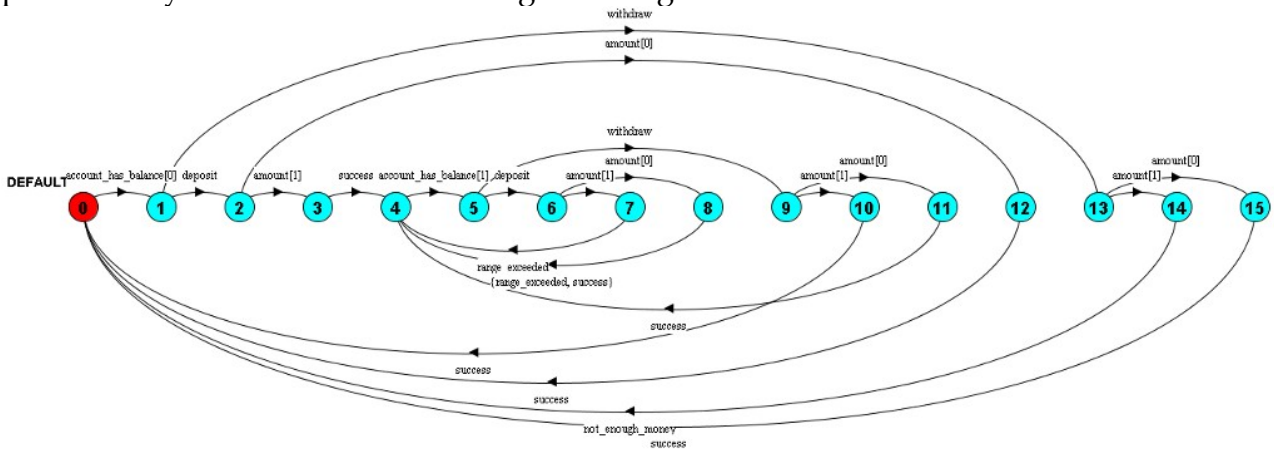
Analysis using the LTSA shows that this specification cannot get into a deadlock. It also tells that no progress violations are detected which means that every action of every process will be eventually available for execution. The author thinks that this sort of automated analysis is one of the strongest features of the LTSA tool.

However, these features come with a cost. As the name of notation suggests, FSP can be used only when the amount of states in the system is finite. This approach limits the type of data that can be modeled in the FSP specifications but makes it possible to systematically analyze every state that the system may have. This limitation is apparent in Formal specification example 1: the account can only have a balance that is an integer between zero to ten. In fact, normally there would no need to guard the deposit action with a condition like it is done in the FSP specification. Without it, the balance of the account could get values outside the set range which would lead to an infinite amount of possible states.

FSP supports *interleaving* which means that it is possible to write separate processes that execute concurrently but synchronize from time to time. This is done with shared actions; if two processes are executed concurrently and they have an action in common, both processes need to execute that action at the same time [Magee and Kramer, 2006].

Unfortunately, the example specification presented here has no use for interleaving which is why this feature is not visible in Formal specification example 1.

LTSA tool can visualize the control flow of the system by drawing a graph. The problem is that even this simple example has 286 different states so the tool refuses to generate the picture. Picture 1 presents the visualization of Formal specification example 1 produced by the LTSA when the range is changed from zero to ten to zero to one.



Picture 1: Visualization generated by the LTSA from the simplified bank account example.

It can be argued that the FSP specifications are good for presenting the control flow of the system using independent processes that can communicate and synchronize from time to time. However, it is not especially good for modeling data and operations that manipulate it. Other formal specification languages such as Z and DisCo are better suited for that purpose.

3.5.2. Z

Z is a formal specification language created by Jean-Raymond Abrial in France and further developed by a team in Oxford University. The team was led by the same Tony Hoare who, as previously mentioned, developed the CSP notation. At the time of writing, Z is one the most known and used notations in the formal specifications field [Lightfoot, 2001].

Z specifications are based on declaring schemas. With the exception of validations, the schema is a black box as it includes just the type declaration for the input, output and their relationship [Diller, 1994]. These schemas model either the state of the system or the operations that change that state [Lightfoot, 2001]. Each schema is graphically separated using *boxes* or *frames* from the other schemas. Schemas may, in addition to the formal notation, also include informal specifications written in natural languages [Sommerville,

2009]. The bank account example is presented in Formal specification example 2 as a Z specification.

```
Account
  balance : ℕ
---
  balance ≥ 0

Withdraw
  △Account
  amount? : ℕ
---
  balance ≥ amount > 0
  balance' = balance - amount?

Deposit
  △Account
  amount? : ℕ
---
  balance' = balance + amount?

Init
  Account'
---
  balance' = 0
```

Formal specification example 2: The bank account example in Z notation.

Formal specification example 2 consists of four schemas. Each schema is divided into two parts. The upper part contains variable declarations and the lower part contains constraining predicates and the body of the schema itself. In the case of the first schema *Account*, the upper part is used to declare a variable *balance* which contains a value from the set of natural numbers.

The *Account* schema also has a predicate which states that the variable *balance* must be greater than or equal to zero. In this case, the predicate is actually not needed because *balance* is a natural number which by definition already has this constraint. In Z, natural numbers also include the number zero [Lightfoot, 2001]. The predicate is written here for the sake of familiarizing the notation to the reader of this thesis.

The schemas *Withdraw* and *Deposit* define operations for the *Account*. They describe how an account changes when the money is being deposited or withdrawn. The usage of the delta sign is a convention that is used to signal that the schema will result in the

change of some state. In this case, it is used before the word Account ($\triangle Account$) in both schemas. In addition, both schemas get an input variable *account?* which is also from the set of natural numbers. The usage of the question mark is also a convention in Z used to mark a variable as an input variable.

The *Withdraw* schema contains a predicate that states that it is not possible to withdraw a larger amount than what the account holds, and that the amount for the withdrawal must be greater than zero. The *Deposit* schema does not have any constraining predicates. Both schemas then introduce a variable *balance'* which gets a value from an assignment statement. In Z, variables ending with a prime are used signify the value of the state schema after the operation has ended [Lightfoot, 2001].

Now, these three schemas are enough to communicate the informal requirements given in the assignment. However, even if by some tool it would be possible to execute Z specifications, this specification would deadlock immediately without a way to initialize an account. For this reason, the third schema called *Init* was added to the example. *Init* is a simple schema, used for creating accounts with a balance of zero.

Z specifications are used to model the data that the system may have in its different states. It is clearly a very different language from FSP which is logical considering that they have different use cases. Although FSP may be a better language to model control flow and concurrency, Z has some features targeted for that area as well. In addition to the predicates which are used to enable or disable operations, Z has a feature called schema conjunction which can be used to join two or more schemas [Lightfoot, 2001]. The author of this thesis assumes that this feature could be used to model similar things that shared actions model in FSP specifications. Of course, this method is less expressive compared to the notation provided by the FSP.

3.5.3. DisCo

The term DisCo comes from the words Distributed Co-operation. It is a formal specification method developed at the Tampere University of Technology and is therefore used often for research purposes. In fact, the origins of the book cited in this thesis, *A Practical Theory of Reactive Systems*, written by Kurki-Suonio [2005] come from the DisCo language. Similar to FSP, DisCo is more than just a formal language as it includes a collection of tools called The DisCo Toolset. These tools can compile, execute and visualize the specification. The DisCo Toolset also has a support for theorem provers although it does not include one [The Disco Project, 2002].

DisCo specifications are built using layers. Layers consist of classes, assertions and actions. Classes encapsulate state in the form of variables. Assertions are conditions that must hold true at all times in the system similarly to predicates in Z. Actions are

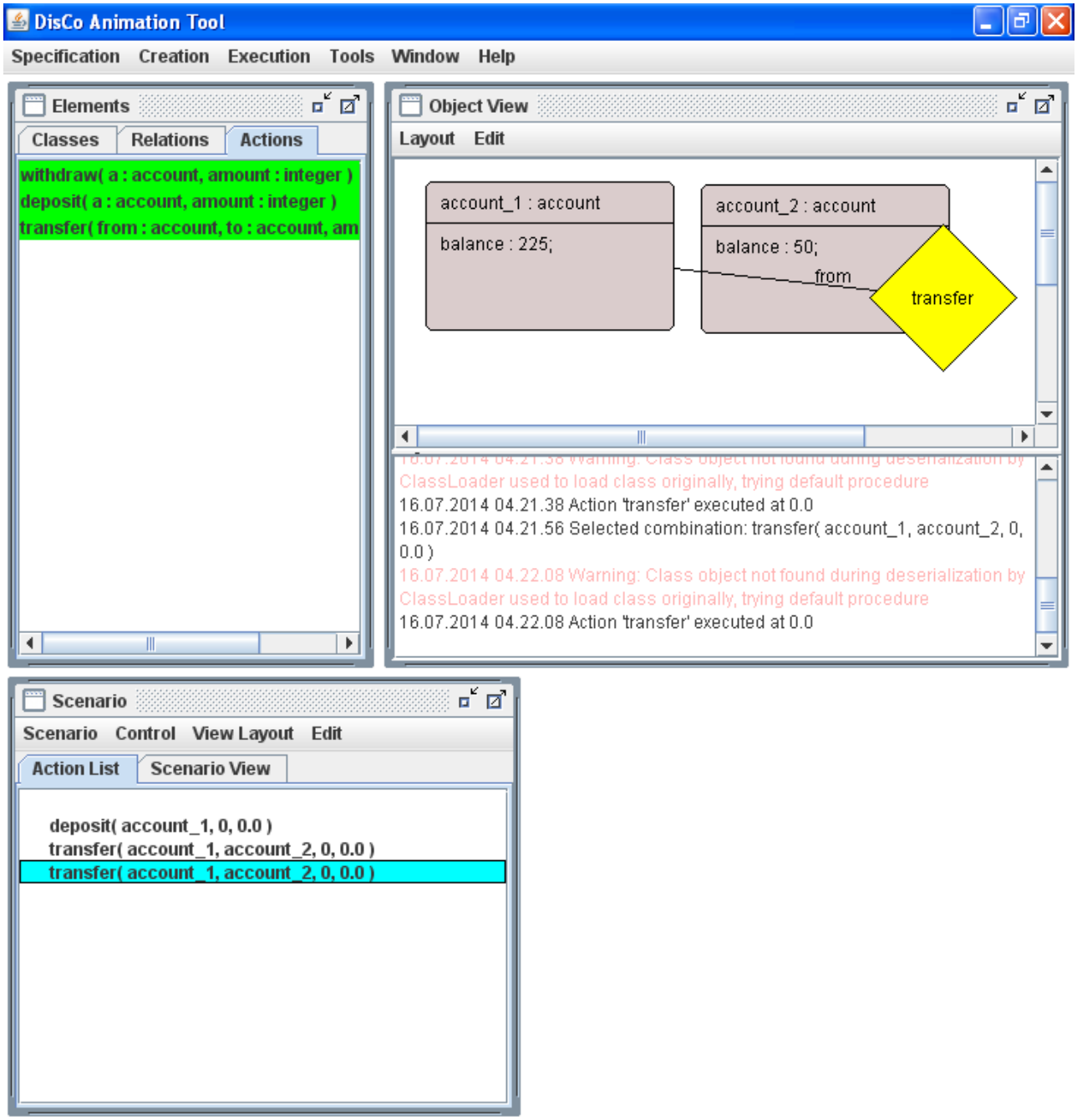
operations that manipulate objects which are instances of classes [The Disco Project, 2002]. Formal specification example 3 presents the familiar bank account example as a DisCo specification.

```
layer bank is
    class account is
        balance : integer;
    end;

    assert accountAssertion is
        forall a : account :: a.balance >= 0;

    action withdraw(a : account; amount : integer) is
        when (a.balance > 0 and a.balance >= amount) do
            a.balance := a.balance-amount;
    end;

    action deposit(a : account; amount : integer) is
        when true do
            a.balance := a.balance+amount;
    end;
end;
```

Formal specification example 3: The bank account example as a DisCo specification.

The specification starts with the declaration of the layer. The layer *bank* includes a class named *account* which has the balance of the account as an integer variable. Then, *assert* is used to declare a safety property which guarantees that all the objects initialized from *account* have a balance that is larger than or equal to zero at all times. Then, the actions called *withdraw* and *deposit* are declared which both take two arguments: an object *a* instantiated from the class *account* and the *amount* that is being either withdrawn from the account or deposited into it. Both actions change the balance of the object *a* after recalculating the new value by using the original *balance* of *a* and the *amount*. Both actions also include a predicate which either makes the action available or unavailable. The *deposit* action is always available while the availability of the *withdraw* action depends on the balance of the account and the amount that is being withdrawn.

DisCo specifications do not require actions for initializing objects. The DisCo Toolset includes a user interface for creating the objects before the actual execution of the actions begins. Picture 2 contains a screenshot from the animator of the DisCo Toolset.



Picture 2: The DisCo Animation Tool.

The DisCo Toolset does not include an editor for editing the specifications. Any normal text editor can be used which is undoubtedly flexible. On the other hand, standard text editors do not include features such as syntax highlighting for the DisCo notation. The author would like to point out that one of the biggest problems of DisCo is its age: the last version of the tool was released in 2002 which makes it impossible or at least very difficult

to run the animator with a modern 64-bit operating system. The author of this thesis had to run the animator using a virtual machine with Windows XP installed.

DisCo has been developed especially for reactive systems in mind. In addition to that, DisCo has a goal of providing a notation that feels natural to people with background in traditional software development [The Disco Project, 2002]. The author of this thesis thinks that this is true and a very good thing: DisCo looks and feels like a familiar object-oriented programming language. This observation is discussed in detail in the next chapter of this thesis.

4. Functional programming paradigm in formal specifications

In the previous chapters, the author gave a brief introduction to functional programming and formal specifications. This chapter summarizes how those two topics are related. The chapter begins with concrete examples about how programming and formal specifications have influenced each other. In addition, the Z and DisCo examples from Chapter 3.5 are analyzed for junctions between the two activities. Some of the existing research about combining functional programming with formal specifications is discussed as well.

All of this aims to show that the previously mentioned flaws of state- and object-oriented programming are partially present in some of the formal specification languages. This leads to an important research question: can functional programming paradigm improve formal specification methodologies just like it can improve programming in general?

4.1. The relationship between formal specifications and programming

Formal specifications and programming have influenced each other in multiple ways. This becomes evident by examining the examples from Chapter 3.5. As mentioned before, one of the goals set for the development of DisCo was to invent a formal specification language that feels familiar to people in traditional software development [The Disco Project, 2002]. By looking at Formal specification example 3, it is clear that this design goal has been reached by utilizing concepts from strongly typed object-oriented programming: DisCo has classes that encapsulate state variables with type definitions, and the execution is based on initializing objects from those classes. Some of the data types in DisCo such as integer, Boolean and set are also familiar concepts from almost every programming language.

There is one key difference between object-oriented programming and DisCo that is worth mentioning here: as can be seen from Formal specification example 3, the actions that manipulate object state, which are comparable to methods in object-oriented programming, are not encapsulated inside the classes. The reason for this is that DisCo supports multi-object actions. In contrast to methods in object-oriented programming, actions in DisCo are asymmetric in a sense that there is no division between the caller and the arguments [Kurki-Suonio, 2005]. In DisCo, there is no need to encapsulate actions inside classes where they don't quite belong, and the formal specification stays clean from the unnecessary classes containing these actions. The author would like to point out that this problem is often solved in object-oriented programming by writing handler or manager classes to manage the state of multiple objects. As this solution is related to the

software design and implementation, its place is not in the formal specifications. Therefore, the approach that DisCo utilizes is well-grounded. Nevertheless, actions in DisCo contain side-effects as they don't return any values but instead change the state of one or more objects. As stated before, side-effects make the implementation harder to reason, and the author suspects that the same applies to formal specifications as well.

Unlike DisCo, Z notation does not seem to be influenced by object-oriented programming at least according to Formal specification example 2 presented in Chapter 3.5. However, there are extensions to Z called Z++ [Lano, 1991] and Object-Z [Smith, 2000] which provide concepts of object-oriented programming such as classes and inheritance to Z notation. The author's review of the literature does not reveal an existing extension to Z based on functional programming. Instead, the interest to the relationship between functional programming and Z notation seems to be focused on producing and verifying the functional implementation based on Z specifications. This methodology is discussed by Sanders and Johnson [1990a] who emphasize that this approach allows the advantages of prototyping.

One of the most known methods for verifying that the implementation satisfies the formal specification is called Floyd-Hoare logic. This method consists of a set of rules for reasoning between formal specifications, implementation and their correctness. Floyd-Hoare logic can be used for example to produce an imperative implementation from Z specifications as demonstrated by Diller [1994]. Although Floyd-Hoare logic is not limited for this purpose, it was designed for imperative languages in mind [Régis-Gianas and Pottier, 2008]. The development of Floyd-Hoare logic is another example of how popularity of imperative languages has affected formal methods. Besides giving criticism about disregarding other programming paradigms, Régis-Gianas and Pottier [2008] have provided a starting point for the research about applying Floyd-Hoare logic to functional programming.

In some papers, even though no specific method or language is mentioned, formal specifications are discussed using terms and concepts from object-oriented programming. For example, Lamsweerde [2000] states that an ideal formal specification language should allow the specification to be organized into units which interact with each other through relationships like specialization, use or instantiation. According to Lamsweerde [2000], each of these units should contain the declaration of variables and their assertions. This kind of approach to formal specifications shares many similarities with object-oriented programming.

The influence between formal specifications and programming is mutual. Just as concepts from programming have influenced formal specifications, formal specifications

have influenced programming. For example, there are multiple programming languages and frameworks that are at least partially based on the ideas behind CSP and its derivatives. In the scope of this thesis, a library based on CSP named Core.async is worth mentioning. Core.async is a Clojure library for asynchronous programming and communication that utilizes *go* blocks and channels for executing blocking operations asynchronously [Hickey, 2013]. Another good example is a programming language called *Go* which has its concurrency aspects based on CSP [Google, 2015].

4.2. Applying functional programming to formal specification languages

The relationship between programming and formal specifications has been discussed extensively in the literature. In addition, object-oriented and imperative programming have had its effects on DisCo, Z and the formal methods in general. However, the subject of applying ideas from functional programming to formal specification languages is not researched as nearly as much. This remark is visible in the examples from Chapter 3.5 as none of the examples show concepts or patterns from functional programming.

However, some research has been made about the subject. The subject of this sub-chapter is in discussing a few independent papers where the combination of functional programming languages and formal specifications was tested. All of the discussed papers have been written after conducting an experiment where functional programming was leveraged to improve the formal specification methods.

The first paper to be discussed is titled *Functional Programming, Formal Specification, and Rapid Prototyping*. It is written by Peter Henderson [1986]. Although the research is old, it is an important paper in the scope of this thesis as it discusses the compatibility between formal specifications and functional programming for the first time. Henderson and his team created a formal specification of an example system that is designed to arrange textual notes into groups by their subject. They found that formal specifications share similarities with functional programming: in both activities, data is defined together with operations that form new data based on some input. New operations are then composed by combining the already created operations. Some formal specification languages are even more similar to functional programming languages: for example, a formal specification language VDM utilizes recursion and abstract data types just like many functional programming languages [Henderson, 1986].

Because formal specifications are close to programs written in functional programming languages, it is an easy task to execute the specification by making the formal specification into a program. Henderson [1986] demonstrates this with a special language and a method created for this purpose called *me too*. The methodology itself is borrowed from functional programming: abstract data types are defined together with operations that manipulates

them. In addition, Henderson's method utilizes recursion. Much like the author of this thesis, Henderson [1986] has embedded his solution in Lisp for execution purposes. He also states that REPL can be used for experimenting with the method.

The literature that was cited in the chapter 3.3 stated that formal specifications should not include details about the design or implementation of the system. Henderson [1986] seems to disagree to some extent as he mentions that writing formal specifications with functional programming languages allows the developers to proceed directly to the software design. After all, the schema of the data and the related operations are already defined in the specification. This approach makes the first version of the software design an iterated version based on the specification. It is even possible to use formal program transformation methods to turn this executable specification into a finished program. Communication becomes easier as well as the formal model includes terminology that the development team can continue to use during the later phases of the project [Henderson, 1986].

Because the formal specification may contribute to the software design, it is even more important to validate the correctness of the specification to avoid defects in the final system caused by the mistakes in requirements elicitation. Being able to execute the specification is useful at this point as it allows finding both the incomplete parts and the defects in the specification. Henderson [1986] also argues that execution helps in evaluation of the design related alternatives. An executable formal specification provides the development team a working prototype which provides confidence to the specification that would be not achievable by reasoning only.

In summary, Henderson [1986] argues that writing specifications as formal specifications with functional programming languages achieves two goals: the specification becomes understandable but precise, and it allows validating the specification without mathematical reasoning by treating the executable formal specification as a prototype. In addition, he wishes to show how software design can benefit from formal methods in general.

Jenny Butler [1995] presented a similar method in her paper *Use of a functional programming language for formal specifications*. Butler was part of a research team whose task was to implement an imaging system for diagnostic microscopy using safety critical methodologies and functional programming. The project had an additional goal of experimenting with formal specifications written in functional programming languages. Unlike Henderson, the team working with this project used Gofer which is a partial implementation of Haskell. The premise of the experiment was that referential

transparency and the mathematical basis make functional programming languages suitable for formal specifications [Butler, 1995].

Like Henderson, Butler [1995] reported that proceeding from the formal specification phase to the implementation was easier and faster as both the formal specification and the implementation were written using the same language. In addition, by avoiding the transformation between different notations and paradigms, some of the possible defects in the final system were avoided [Butler, 1995]. However, the initial version of the formal specification was verbose, too complex in terms of algorithms and did not utilize the possibilities of the specification method properly. Butler suspects that the initial failure happened because the problem-domain was presented incorrectly and the team had too little experience with formal methods. The team proceeded to develop a second, more abstract formal specification that corrected the mistakes of the first version [Butler, 1995].

Butler's research is interesting, because it compares the Gofer-based formal specification to another formal specification written in Z for the same system. The team argues that both the Gofer-based and the Z-based specifications were equally powerful and user-friendly methods. However, Gofer excelled in expressing the functional parts whereas Z was better in expressing the parts related to the state of the system. Overall, they found Gofer less complicated than Z. The author would like to point out that as they are the creators of the Gofer-based method, their objectivity is questionable.

The team utilized many features included in Haskell such as list comprehensions, λ-expressions and pattern matching in their specification method. To complete their method, they had to extend the language with a few minor features such as new data types. Haskell's strong, polymorphic type system worked exceptionally well as it enabled function reuse and compiler time error checking [Butler, 1995].

The Gofer-based specification was executed with a custom-made animation system. In addition, the team's confidence in the specification was increased with formal verification techniques. This task was simple to perform due to the mathematical nature of Gofer. In addition, referential transparency allowed evaluating and verifying the specification in a modular fashion as the return values of the functions do not depend on the application state. It would have been also possible to verify the specification with tools such as theorem provers if they would have been available. Butler [1995] mentions that the lack of formal semantics and tools is currently a problem in using functional programming languages for formal specifications.

In the end, the experiment showed that functional programming languages can be used for formal specifications. Utilizing the same language for the implementation and the formal specification reduces training costs and allows the development team to iterate

from the specification to the implementation [Butler, 1995]. However, the method is not ready for industry usage before tool support is improved and tasks like project management evolve to support these methods.

The last paper discussed in this chapter is titled *Functional languages for the implementation of formal specifications* by Sanders and Johnson [1990b]. They experienced with a different kind of method compared to Butler's and Henderson's methods. They used Haskell like Butler and her project team did few years after, but chose a hybrid method between writing formal specifications with functional programming language and modeling constructs of the programming language using the specification language. First, a formal specification was produced using the Z notation, and it was verified to make sure that the requirements were satisfied. Then, the formal specification was transliterated into a program written in functional programming languages. This was an iterative process with a goal of improving the performance of the program one step at the time. The last part of the process included a step where the program was implemented using an imperative language [Sanders and Johnson, 1990b]. In simple cases, some of these steps were omitted. For example, if the performance was acceptable after the iterative development with the Haskell-based program, there was no need to rewrite the program using an imperative language.

This method is clearly different from the previously discussed methods. The author of this thesis would intuitively guess that there is a lot of overhead in transliterating the Z specifications into functional programs. However, Sanders and Johnson [1990b] argue that this was not the case. The transformation was carried out by creating data types from Z schemas that represent state, and functions from schemas that manipulate state. The naming conventions of the input and output variables in Z notation helped in this process. In addition, parts of the mathematical toolkit and data types of Z were implemented in Haskell [Sanders and Johnson, 1990b]. Polymorphism was utilized similarly to Butler's approach to provide generic approach to the data.

Like other experimentalists in the area, Sanders and Johnson [1990b] gave an example of their method in practice. They demonstrated their method by creating a formal specification of a procedure that writes data into a file at certain offset in the Unix file system. They had to implement both the problem-domain related data types such as *BYTE* and *ZERO* and the Z-based data types such as *set* and *relation* in Haskell. The first prototype of the procedure was inefficient so the program was manipulated using formal program transformation methods until the performance was satisfactory [Sanders and Johnson, 1990b].

The main difference of this method compared to the traditional formal specification methods lies in the program transformation. As opposed to refining the formal specification and performing proofs, this approach preserved the correctness of the program by using methods that alter the structure and the form of the program but not the behavior. In addition, this approach helped in improving the performance of the program. The method itself was proven to be a viable option as the method was successfully applied in practice [Sanders and Johnson, 1990b].

5. Implementing formal specification library and tools with Clojure

Much like Butler [1995], Henderson [1986] and Sanders and Johnson [1990b], the author of this thesis has experimented with formal specifications and functional programming. This chapter aims to present the goals, the problem-domain and the solutions related to the experiment. The chapter also includes a lot of code examples. In addition, a formal specification example is used to prove that the method works as intended. Before the end of chapter and moving on to the conclusions of this thesis, the author wants to briefly evaluate his own work and the suitability of Clojure for formal specifications.

The author has taken into account the previous research and experiments, and implemented a library and a tool for writing executable formal specifications. However, some of the goals and design decisions are different from the other approaches. For example, continuing with the theme established in Chapter 2, the language chosen for the task is Clojure. It offers a modern perspective and an alternative to Haskell, and its metaprogramming capabilities are useful in implementing domain-specific languages and similar libraries. All of the code and examples related to the project are open-source and available on the author's GitHub page at https://github.com/MattiNieminen?tab=repositories. In addition, the core namespace of the author's library is available in Appendix 1.

5.1. The characteristics of a Clojure-based formal specification method

The previous chapters have mentioned a lot of different approaches to formal specifications. As stated in Chapter 3.2, different notations have different uses and they perform well at different tasks. Just like other notations and methods, the author's formal specification method has its advantages and disadvantages. Some of the design goals for the method are based on improving other notations and examples already discussed in this thesis, while other goals are based on the observations found in the literature about improving formal specifications. Of course, the decision to use Clojure affects greatly both to the goals and characteristics of the developed method.

The main goal of developing a custom formal specification method is to experiment with writing formal specifications using functional programming languages with similar methods that Butler [1995] and Henderson [1986] used. The method of Sanders and Johnson [1990b], where specifications where transliterated from Z to Haskell, is not something that the author aims for, although the methods share some similar characteristics. As it is easier to check the correctness of programs written in functional programming languages than programs written with imperative languages, the approach

to combine formal specifications with functional programming is justified [Régis-Gianas and Pottier, 2008].

The author pursues similar results that Butler [1995] and Henderson [1986] achieved. Out of all the mentioned advantages, the author believes that the most important advantage is the ability to validate and test the formal specification by executing it. As already mentioned in Chapter 3.4, performing proofs is difficult and it requires expertise in the field of mathematics. Therefore, the method of the author has been developed particularly for execution in mind. Of course, as functional programming languages are executable by nature, this goal is achieved without any special effort.

As discussed in Chapter 4.2, it is possible to refine formal specifications into implementation. As Butler [1995] and Henderson [1986] reported, this goal can be achieved when the same language is used for both the implementation and formal specifications. This is important as developers tend to underestimate the complexity of software [Nummenmaa and Nummenmaa, 2011], and think that formal specifications do not suit or contribute to the development process. The author of this thesis believes that developers are more likely to accept formal specifications when it is possible to refine them into the implementation.

The ability to execute the formal specification and refine it into implementation makes the method more user-friendly and approachable. However, even then the creator of the specification must know how to use the notation. Using Clojure helps mitigating this disadvantage as developers who are familiar with Clojure or any Lisp don't have to learn a new notation to get started with formal specifications. If the development team does not yet know Clojure, but is committed to adopting it for implementation, there will be no separate training costs for learning the notation.

Using an existing language has other benefits as well. For example, as creating a new formal language would take several years [Sommerville, 2009], using an existing language that already has an execution environment saves a lot of time. The author does not need to come up with the syntax or prove that the language is truly a formal language. In addition, if the existing language has structuring facilities mentioned in Chapter 3.2, the new method automatically gains those facilities. As a modular language that provides structure to the code in the form of namespaces, Clojure offers these facilities directly [Halloway and Bedra, 2012]. It should be noted that using an existing language leaves the author with less control over these aspects as the new method gains a lot of properties automatically from the existing language.

As discussed before in Chapter 3.1, formal specifications are used together with informal specifications. For example in Z notation, it is possible to write informal parts of

the specification inside the schemas using natural languages [Sommerville, 2009]. Luckily, Clojure has syntax for commenting code like most programming languages do. This allows the usage of natural languages among the formal notation. In Clojure, comments are written by prefixing a line with a semicolon as can be seen in the examples in Chapter 2. Although not meant to replace the informal specifications or other requirement related documents, the ability to write comments helps in understanding the complex parts of the formal notation. Therefore, the author aims to leverage the existing notation for writing comments in Clojure.

It would be a simple task to implement a domain-specific language based on CSP or FSP with Clojure. However, as functional programming languages are good at defining data and operations that manipulate it, an action-based language similar to DisCo makes better use of the paradigm. Action-based specifications are built by declaring atomic actions that are enabled or disabled depending on the application state [Kurki-Suonio, 2005]. Enabling and disabling actions using predicates allows modeling of some control-flow for the system even though these methods are not designed to model it like CSP and FSP are. In accordance with functional programming paradigm, the availability of actions should not depend on the global program state but on the arguments that the action receives; the actions defined with the author's library, which will be close to normal functions, should not contain side-effects unless they are absolutely required.

Clojure utilizes dynamic typing as discussed in Chapter 2.3. This causes a clear distinction between the Haskell-based [Butler, 1995], [Sanders and Johnson, 1990b] and Clojure-based formal specification methods. Haskell has a versatile static type system: it uses *type inference* to avoid labeling code with types where they are not required. In addition, Haskell has powerful facilities for creating custom types and *type classes* [Lipovača, 2011]. Type classes are like interfaces that define behaviors and allow deriving types from them. Clojure also has some mechanism such as *deftype*, *defrecord* and *reify* that provide possibilities for defining custom data types [Hickey, 2014]. However, without real static typing, these mechanism cannot provide compile-time checking which means that the code does not fail until it is run. Although writing unit tests helps in catching errors related to the misuse of types, this is a clear disadvantage compared to Haskell and a challenge for writing formal specifications with Clojure: after all, the idea of formal specifications is to indicate errors as early as possible.

As it would be difficult to pick those features of Clojure that the creators of the formal specifications want to use, the author has decided to expose all of Clojure to his method. In other words, the method does not include just a subset of functions and macros in Clojure but the whole core namespace and everything else that the user wants to depend

on. Ultimately, this may not be a good thing, but for now the author considers it to be the best solution for experimenting what is possible to achieve with the library.

Although the focus is in experimenting with the combination of functional programming and formal specifications, the author wants to set some goals for the method that are not directly related to the research questions. These goals are designed to avoid some of the issues present in DisCo and Z. For example, the developers are not likely to adopt DisCo because of its difficult installation and its age [Nummenmaa and Nummenmaa, 2011]. In addition, as mentioned in Chapter 3.5.3, the author found that the DisCo Toolkit cannot be run using a computer with a modern processor and operating system which is forgivable only for legacy systems.

Luckily, Clojure programs can be packaged into executable JAR files similar to Java programs. These JAR files can be executed with any computer and operating system that has Java SE installed [Hagelberg, 2014]. In addition, the user-interface of the tool will be accessed with a web-browser which makes it possible to use the tool over the network. The tool is therefore well suited for schools, universities, companies and other parties who can host the it for its users. This approach effectively eliminates the need to install anything to the users' workstations.

Another deficiency that the author aims to avoid is the usage of non-ASCII characters which are found in the Z specifications. Z schemas include characters like the Greek letter delta (Δ) and the frames that surround the schema and its parts. This kind of markup is difficult to produce using a keyboard. According to a study by Larsen *et al*. [1996], developers rather work with ASCII-based formal specification languages than with mathematical notations. This is not a problem in DisCo which resembles a programming language by design: the author's method just takes this approach even further as the language is a real programming language.

In summary, the goal of the author is to create an executable action-based formal specification method that demonstrates how Clojure can be used to create executable formal specifications. The author aims to utilize the benefits of functional programming paradigm as much as possible. The method should be easy to introduce to companies and teams already committed to Clojure, and its usage should not generate training or installation costs. In addition, the usage of the method should not compromise project work by generating work that does not contribute to project deliverables. In the best case, the method should show potential for surpassing DisCo and Z.

5.2. Implementation and the usage of the library

For a working action-based formal specification method, the author must start by finding a way to define the actions. Actions should have a body that gets evaluated and its result

returned when the action is executed, and a predicate to either allow or disallow the execution of the body. After actions can be defined, the author must implement a way to execute those actions. In addition, some helper functions are required for checking the predicate and storing the results of executing an action for later use. This sub-chapter presents the author's implementation that aims to satisfy these requirements.

In their simplest form, actions could be just normal functions as Clojure functions support runtime pre- and postconditions as metadata. These conditions can be used to validate the arguments or the return value of the function upon execution [Hickey, 2014]. Code example 7 shows the usage of pre- and postconditions in Clojure functions.

```
; Sums two numbers if they are not the same and their
; result is an odd number
(defn weird-sum
  [x y]
  {:pre [(not= x y)]
   :post [(odd? %)]}
  (+ x y))

=> (weird-sum 0 1)
1
=> (weird-sum 1 1)
AssertionError Assert failed: (not= x y)
=> (weird-sum 2 4)
AssertionError Assert failed: (odd? %)
```

Code example 7: A Clojure function with pre- and postconditions.

The pre- and postconditions are added to a function by inserting a map of conditions before the function body. In Code example 7, the function *weird-sum* successfully returns a value only in cases where the conditions are true. The example also shows that the precondition can refer to the arguments of the function by name, and the postcondition to the return value of the function with a percent sign. The example also shows that if the conditions are not satisfied when calling the function, an exception is thrown.

However, these conditions are impossible to test reliably without calling the function. The author aims to avoid bad design by creating a solution where it is possible to test the availability of an action without executing it. This is especially important when actions contain side-effects that are meant to be run only once when the action is executed. In

addition, at the time of writing this thesis, the REPL shipped with a Clojure editor called Light Table does not support pre- and postconditions [Fogus, 2012]. As the author uses Light Table for development, using pre- and postconditions is not sensible for now.

The author has decided to represent actions with normal Clojure maps by storing the predicate and the body under separate keys called *:available* and *:body*. This approach separates the action body from the availability which makes it possible to manage them separately. However, the map solution does not automatically solve any problems as evaluating a map results in the evaluation of all of its key-value pairs: it is still impossible to evaluate the availability without evaluating the body.

To solve this problem, the author has decided to wrap both the availability and the action body into functions. This way the map evaluates into key-value pairs where the values are just normal functions. These functions must be explicitly called which prevents accidentally evaluating the action bodies when the availability is checked and vice versa. Code example 8 presents an initial version of a function that returns a map representing an action.

```
(defn square
  [x]
  {:available (fn [] (pos? x))
   :body (fn [] (* x x))})

=> (def square-of-5 (square 5))
#'user/square-of-5
=> (:available square-of-5)
#<user$square$fn__2162 user$square$fn__2162@67d6027c>
=> ((:available square-of-5))
true
=> (:body square-of-5)
#<user$square$fn__2164 user$square$fn__2164@ca79270>
=> ((:body square-of-5))
25
```

Code example 8: A prototype of a function used for defining actions.

In order to use the action from Code example 8, the function *square* is first called normally with an argument. The returned map, called an *action map* from now on, contains functions for the action body and the availability which were created with a macro called

fn. As *(:body square-of-5)* returns the function under the *:body* key, two pairs of parentheses are required for calling the function under the key. It is important to notice that despite the need for the double parentheses, calling these functions is easy as they don't take any arguments of their own. Instead, they refer to the argument of *square* which makes them *closures*. There are many ways to define a closure, but in simple terms, it is a function that knows the value of a binding belonging to its parent scope [Sussman and Steele, 1975]. Code example 8 proves that the closures work: the first set of parentheses is used to get the functions under the keys *:body* and *:available*, and the second set of parentheses is used to call those functions without arguments.

Writing actions as functions that return maps of functions is a tedious process. In addition, these functions should be labeled as special functions that return action maps in order to allow tools to identify them. For these reasons, the author has decided to write a custom macro to define these action-returning functions. As mentioned in Chapter 2.3, macros can be used to add new features to Clojure itself.

Normally, Clojure programs are compiled on the fly. Compilation starts when the reader reads the textual source code from a file or some other input source. The reader then creates the data structures which the compiler then compiles to Java Virtual Machine bytecode. Whereas function calls and other forms are evaluated at runtime, macro calls are evaluated at compile time [Hickey, 2014]. For example, unlike functions, macros get their arguments as unevaluated forms and symbols. The difference may seem small but it enables a lot: the compiler does not have to know how to compile the arguments of a macro call as long as the macro returns code that the compiler can compile.

The macro that the author created is called *defaction*. It is used to produce similar functions that were shown in Code example 8. The macro works by iterating the given action map and wrapping each of its values into a function. The *defaction* macro is presented in Code example 9.

```
(defmacro defaction
  "Like defn in style, but is used to define functions that return executable
  formal specification actions. action-map must be a map which will be the
  return value of by the defined function with all the values wrapper into
  closures."
  ([name args action-map]
   {:pre [(map? action-map)]}
   `(defn ~(with-meta name {:action true}) ~args
      ~(reduce-kv #(assoc %1 %2 `(fn [] ~%3)) {} action-map)))
```

```
([name doc-string args action-map]
 {:pre [(map? action-map) (string? doc-string)]}
 `(defn ~(with-meta name {:action true}) ~doc-string ~args
    ~(reduce-kv #(assoc %1 %2 `(fn [] ~%3)) {} action-map))))
```

Code example 9: A macro for creating functions that return action maps.

The *defaction* macro presented in Code example 9 looks intimidating at first so the author will explain it thoroughly. First of all, macros are defined using a *defmacro* macro which is similar to *defn* macro used to define functions [Hickey, 2014]. Both macros take the same arguments in the same order: the name of the function or macro, the optional documentation string, a vector of arguments and the bodies of the function or macro which will be evaluated and returned [Hickey, 2014].

The *defaction* macro is similar to *defn* and *defmacro* macros by design. For example, the arguments that *defaction* takes are similar to its role models *defn* and *defmacro*. The optionality of the documentation string is achieved with *arity overloading*: *defaction* has two separate argument lists and bodies starting at lines six and ten in Code example 9. The first five lines are used to define the macro by name and add documentation to it, but after that the macro gets two different argument lists and bodies separated by parentheses. The only difference between these two versions is that the latter is used to attach optional documentation to the resulting function. The difference between *defaction* and its role models *defn* and *defmacro* is clearly visible after the argument lists: preconditions are used to check that the *action-map*, which is used as a body of the resulting function, is truly a map: unlike *defn* and *defmacro*, *defaction* does not accept or return any other kind of bodies.

Macros usually utilize *grave accent* (`) and *tilde* (~) characters which are used to control the evaluation of symbols and forms inside the macro: prefixing a symbol or a form with a tilde evaluates it, and the grave accent prevents that [Fogus and Houser, 2011]. As *defaction* is designed to return code that creates functions, it is natural that both bodies of *defaction* start with calls to *defn* with a grave accent before them. This way, the code that the macro returns leads to the creation of a new function. This happens, because everything that is not evaluated by the macro itself will be evaluated eventually in any case.

At the same lines where the bodies of the macro are started with *defn*, tilde is used to evaluate the symbol *args*, which leads to the symbol being replaced with the argument given to the macro when it was called. Tilde is also used before the calls of the function *with-meta* which is used to add metadata to the new resulting function. This metadata is used by the tool that the author created to identify the functions that generate actions.

The lines nine and thirteen in Code example 9 transform the values of the *action-map* into closures. This is done with a function *reduce-kv* which is a variant of a normal reduce function designed especially for reducing maps. *Reduce* or *fold* functions are used often in functional programming. They iterate over collections by calling some given function first for the first and the second element of the collection, then to the result of that and the third element of the collection and so on until one value remains [Hickey, 2015a]. The standard *reduce* function can be seen in action in Code example 6 in Chapter 2.4. In *defaction*, *reduce-kv* is used with an anonymous function that wraps each of the value of the original *action-map* into a new anonymous function. This process transforms the normal forms of the *action-maps* into closures.

The author has written unit tests to make sure that the *defaction* macro works as intended. The reader of this thesis is invited to explore these tests at the GitHub repository of the project. In addition, the functionality of the macro is shown in Code example 10.

```
; Quoting is used to avoid evaluation of the form before it is passed to
; macroexpand. Normally, there is no reason to define a var that contains
; the unevaluated form. However, it is often used together with macroexpand
; in order to manually test the expansion and evaluation of macros.
(def unevaluated-square
  '(defaction square
  [x]
  {:available (pos? x)
   :body (* x x)}))

=> (macroexpand-1 unevaluated-square)
(clojure.core/defn square [x] {:body (clojure.core/fn [] (* x x)),
                               :available (clojure.core/fn [] (pos? x))})
```

Code example 10: Testing *defaction* macro with *macroexpand-1*.

The functionality of *defaction* macro is demonstrated in Code example 10 by using a function called *macroexpand-1* which takes a form as an input and returns it after the macro form has been expanded [Hickey, 2015a]. The example shows that *macroexpand-1* returns a form that is equivalent with the form presented in Code example 8. The important thing is that the values under the keys *:body* and *:available* are indeed closures as they should.

The way of executing these actions using the double parentheses was presented in Code example 8. However, executing an action like that does not check the availability of

the action which is why the author has created a helper function for executing actions. This function is shown in Code example 11.

```
(defn execute
  "If action is available and well-formed, executes its body and returns the
  result. If ref is given, the return value will be also stored into the ref.
  See execute-init for creating the ref."
  ([action]
    (if (test-action action) ((:body action))))
  ([action ref]
    {:pre [(instance? clojure.lang.Ref ref)]}
    (dosync (ref-set ref (execute action)))))

=> (execute (square 5))
25
=> (execute (square -5))
Exception action is not available for execution
```

Code example 11: A function for executing actions created with *defaction*.

As can be seen from Code example 11, the function *execute* has two argument lists and bodies similar to *defaction* macro. The first version with a single argument is used to normally execute the action. It uses the *if* special form [Hickey, 2014] for checking the availability of the action and proceeds to execute its body if it is available. The function *test-action is* available for examination in the GitHub repository and in Appendix 1.

It is often required to pass data successively from an action to action. The latter version of *execute* was implemented for this purpose as it helps avoiding nested calls to *execute*. The second version of the function can be seen in Code example 11: it takes a ref as an extra argument and saves the result of the execution into that ref. Refs were discussed earlier in this thesis in Chapter 2.4. Before the execution, precondition is used to check that the *ref* is truly a ref. After that, the first version of the *execute* function will be called, and the returned value will be saved into the *ref* using a function called *ref-set* [Hickey, 2015a]. As discussed in Chapter 2.4, a transaction must be opened with *dosync* when modifying values inside refs.

Refs are usually created with a function called *ref* [Hickey, 2015a]. However, the author created a helper macro that simultaneously executes an action, saves the result into a new

ref and adds metadata to it for tool support. This macro, called *execute-init*, is presented in Code example 12.

```
(defmacro execute-init
  "Calls execute normally for the given action, but stores the returned
  value into a ref. A var is created with the name var-name which refers to
  the ref. An optional validator function can be given to the ref.
  See set-validator! function and Clojure documentation about refs for more
  details."
  ([var-name action-expr]
  `(def ~(with-meta var-name {:spec-ref true}) (ref (execute ~action-expr))))
  ([var-name action-expr validator]
   `(def ~(with-meta var-name {:spec-ref true})
      (ref (execute ~action-expr) :validator ~validator))))

=> (execute-init square-result (square 5) #(>= % 0))
#'clj-formal-specifications.testing/square-result
=> @square-result
25
```

Code example 12: A function for executing actions and saving their results to refs.

The macro *execute-init* shown in Code example 12 has a lot of similarities with the *defaction* macro discussed earlier in this chapter: *execute-init* is a macro with multiple argument lists and bodies, and it utilizes grave accent and tilde for controlling the evaluation. First, the macro uses the normal *execute* function to execute the given action, and stores the returned value inside of a new ref. Then, a technique from Code example 5 from Chapter 2.4 is used: *def* is used to create a var which points to the created ref. The previously mentioned *with-meta* function is used to add metadata to the var which is useful when the developed tool has to distinguishing refs created with *execute-init* from other refs. The tool is discussed in detail later in this thesis.

The function *ref* supports many optional options [Hickey, 2015a]. In the scope of this experiment, the most interesting option is the *validator*. Validators are functions that can be attached to a ref when it is created. The validator must be a function that takes a single argument. Then, when the value of the ref is about to change, the validator gets automatically applied to the new value. If the validator returns a value that evaluates to *true*, the change is committed successfully. Otherwise, an exception is thrown and the

change will not be committed. In Clojure, all values except *nil* and *false* are evaluated into true [Hickey, 2015a].

The macro *execute-init* utilizes arity overloading in order to provide support for attaching validator functions to the created refs. In Code example 12, an anonymous function that returns true only if its argument is greater than or equal to zero is used as a validator for *square-result*. If some action that returns negative numbers would be executed with *square-result* as a target ref, an exception would be thrown. The validator is useful for formal specification purposes as it can be used to check the safety property of the formal specification discussed in Chapter 3.2: if an exception is thrown during the execution because of the validator, the specification is certainly defective as the previous combination of executed actions has led to a state that is not allowed in the system.

5.3. Example specifications created with the author's solution

It is a straightforward task to test that the author's library works as planned. However, it is very difficult to prove that it is intuitive to use and as suitable as other action-based notations for its intended use. In order to convince the reader of this thesis that the library is indeed a credible alternative for DisCo and Z, the author has decided to give examples that demonstrate the kind of formal specifications that can be built using the author's method.

It is natural to proceed with the familiar bank account example from Chapter 3.5. This way, some comparison can be made between DisCo, Z notation and the author's library. As FSP is used for different purposes, the comparison between the author's library and FSP is not meaningful. Formal specification example 4 is used to demonstrate how the bank account and its operations can be modeled using Clojure and the author's library.

```
(ns clj-formal-specifications.examples.account
  (:require [clj-formal-specifications.core :refer :all]))

; Actions
(defaction account
  []
  {:body {:balance 0}})

(defaction deposit
  [account amount]
  {:body (update-in account [:balance] + amount)})
```

```
(defaction withdraw
  "Decreaces the balance of an account unless the balance would become
  negative."
  [account amount]
  {:available (>= (:balance account) amount)
   :body (update-in account [:balance] - amount)})

; Validator for refs
(defn valid-account?
  [acc]
  (not (neg? (:balance acc))))
```

Formal specification example 4: The bank account example using the author's library.

There are three actions in Formal specification example 4. The first one, called *account*, is used to initialize a new account with a balance of zero. This action has a similar role to the Z schema *Init* discussed in Chapter 3.5.2. The *account* action is followed by the *deposit* action which utilizes a higher-order function *update-in* to update the balance of an account by using the + (sum) function. The *deposit* action does not have its availability defined as the library is designed to assume that actions without *:available* key are always available. The last action in the example is the *withdraw* action which has a similar predicate to its DisCo and Z counterparts. The *:body* of *withdraw* is almost identical to the *:body* of *deposit* action but instead of increasing the balance it is decreased.

The example also has a standard Clojure function called *valid-account?* which is not used in the example itself. Instead, it is meant to be used as a validator function for the refs containing an account. Code example 13 shows how to use *execute*, *execute-init* and *valid-account?* to execute the banking specification presented in Formal specification example 4.

```
=> (execute-init some-account (account) valid-account?)
#'clj-formal-specifications.testing/some-account
=> @some-account
{:balance 0}
=> (execute (deposit @some-account 750) some-account)
{:balance 750}
=> @some-account
{:balance 750}
=> (execute (withdraw @ some-account 500) some-account)
```

```
{:balance 250}
=> @ some-account
{:balance 250}
```

Code example 13: Example expressions for executing the bank account specification.

The first line in Code example 13 is used to call *execute-init* with the *account* action. In addition, the previously mentioned validator function is attached to the ref. In this example, as the specification satisfies the safety property, the validator does not throw any exceptions. The example also shows how the *execute* function is used to modify the ref. As the *execute* function is responsible for updating the ref, the actions itself stay clean of side-effects. The value inside the ref is read using the reader macro @ which was discussed in Chapter 2.4.

Formal specification example 4 is not perfect as the specification is kept short for this thesis in mind. In practice, it would be wise to define a standard pure function for creating the account, and then call it from the *account* action. In addition, it would be a good idea to avoid repeating the same pattern in the bodies of actions *deposit* and *withdraw*. Instead, a higher-order function should be defined that takes an account, the amount and a function as arguments and uses them to return a new account with a new balance. These changes make the example more verbose and harder to compare with Z and DisCo specifications but provides a better reusability as the standard pure functions can be used in the implementation phase without any modifications.

As in any programming language, functions can only return one value in the end. In all the formal specifications examples, the author has presented actions and processes that manipulate a single account at a time. But what about situations that require atomic changes between multiple entities? The author mentioned in Chapter 4.1 that DisCo supports multi-object actions that can take multiple objects as arguments and make changes to all of them. This is possible because actions in DisCo don't return anything but instead use side-effects to modify the state of the objects. Even though actions in DisCo contain side-effects which are harder to reason than pure actions, the upside of this approach is that it is easy to write atomic multi-object actions.

It is possible to use the author's library to write multi-object actions. Instead of writing actions that take values as arguments, it is possible to write actions that manipulate the refs directly. In this case, the creator of the action must start a transaction in the *:body* of the action using the *dosync* form. As mentioned in Chapter 2.4, Clojure is an impure practical language that provides mutable references for situations where their usage is

justified. As the author's library is designed to embrace the practical side of Clojure, this approach should be considered acceptable.

The other way to solve the same problem is to wrap multiple objects in the same data structure. Actions that manipulate and return these kinds of data structures are not real multi-object actions as they manipulate just a single object that wraps other objects. However, they do achieve the same goal that real multi-object actions achieve. Both of the mentioned solutions are demonstrated in Formal specification example 5. It shows two ways to write an advanced version of the *withdraw* action which transfers money from an account to a person's wallet.

```
(defaction withdraw-with-refs
  "Transfers money from an account to a person's wallet in a transaction."
  [account-ref person-ref amount]
  {:available (>= (:balance @account-ref) amount)
   :body (dosync
           (alter account-ref update-in [:balance] - amount)
           (alter person-ref update-in [:wallet] + amount))})

=> (def person-ref (ref {:wallet 300}))
#'clj-formal-specifications.testing/person-ref
=> (def account-ref (ref {:balance 100}))
#'clj-formal-specifications.testing/account-ref
=> (execute (withdraw-with-refs account-ref person-ref 100))
{:wallet 400}
=> @person-ref
{:wallet 400}
=> @account-ref
{:balance 0}

(defaction wrapped-withdraw
  "Transfers money from an account to a person's wallet using a single map."
  [data amount]
  {:available (>= (:balance (:account data)) amount)
   :body (update-in (update-in data [:account :balance] - amount)
                    [:person :wallet] + amount)})
```

```
=> (def data {:person {:wallet 300}
              :account {:balance 100}})
#'clj-formal-specifications.core/data
=> (execute (wrapped-withdraw data 100))
{:person {:wallet 400}, :account {:balance 0}}
```

Formal specification example 5: Writing multi-object actions using the author's library.

The first action in Formal specification example 5 is called *withdraw-with-refs*. That action contains a body with a transaction that modifies both refs given as arguments to the action. Software transactional memory works exceptionally well here: either all the refs will be updated, or none of them. This brings confidence to the specification as there is no need to worry that execution-related errors lead to violations of the safety property. Below the action itself, some test data is created and the action is executed for testing purposes. After that, the other action called *wrapped-withdraw* is presented. That action takes a single map as an argument, and returns a new map where the *:balance* of the account and the *:wallet* of the person has been recalculated using the *amount* and the original values. Both actions are tested in the example by using test data.

By now, the reader of this thesis may have noticed that actions created with the author's library consist of similar components than actions in DisCo and schemas in Z. Actions generally consist of a name, arguments, a predicate and a body. There may be other ways to define actions in action-based formal specification methods. However, as the author has studied DisCo and Z during the writing of this thesis, it is natural that the author's method shares some similarities with both of them. The goal of this experiment is not to reinvent action-based formal specifications, but to provide more value to the methodology with functional programming.

The author has created a full example of the bank account specification which contains a possibility to create multiple accounts and persons. In that example, an account is owned by one or more persons. Everyone can deposit money to an account but only the owners can withdraw it. As this example is long, the author has decided to not to explain it in this thesis; the exploration of the example is left to reader. The shared bank account example is available in GitHub and in Appendix 2.

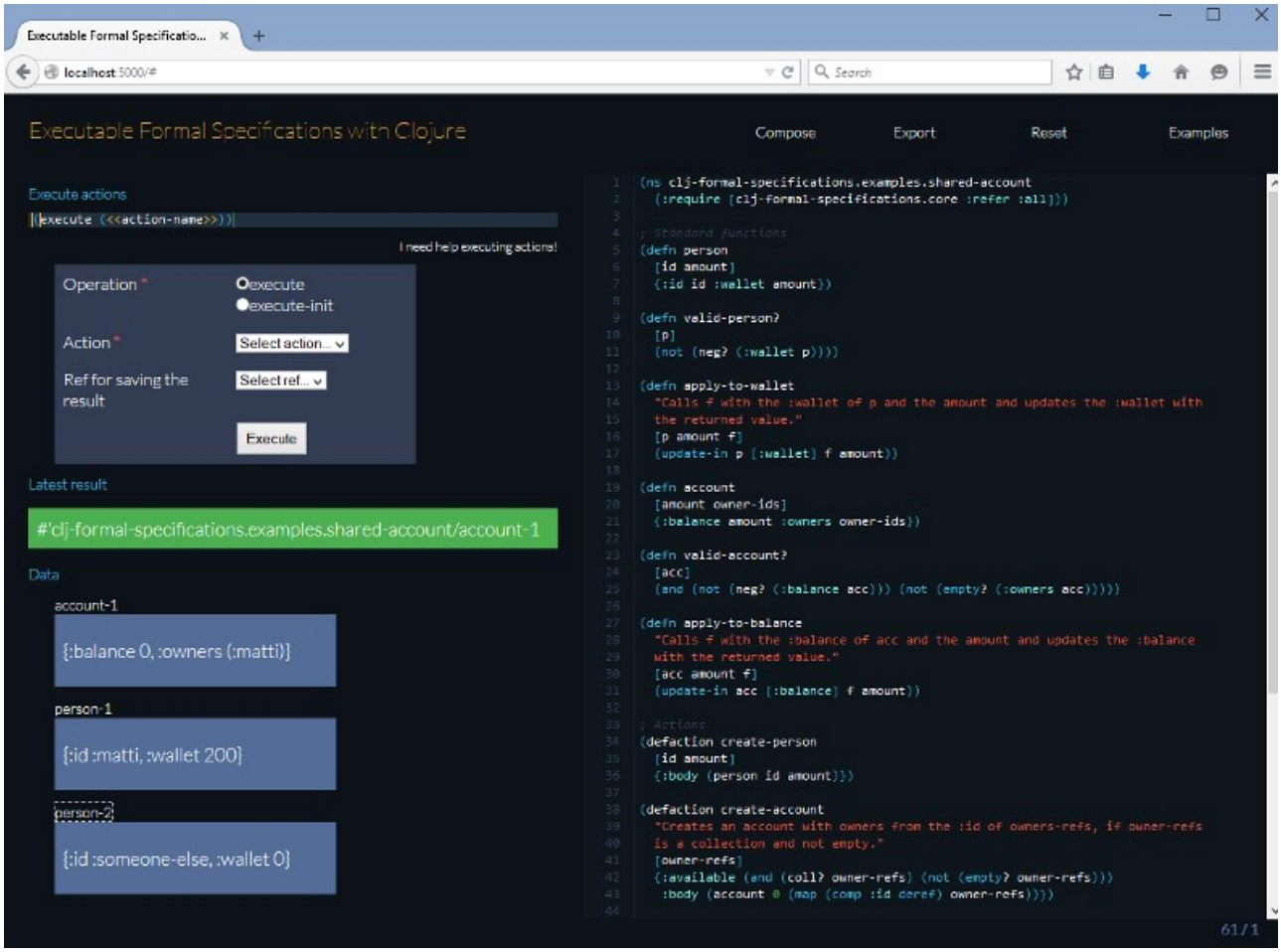
The author has also created unit tests for the examples which are available in the GitHub repository. Originally, the examples and their unit tests were used to verify that the library is working correctly. However, the author has noticed that the unit tests related to the formal specifications itself are actually useful in catching some errors. As Clojure is a dynamically typed language, the unit tests can be used to catch errors that compilers

catch in statically typed languages. For example, if by mistake the formal specification in Appendix 2 would include an action, that would try to access the *:wallet* of an account instead of the *:balance*, the mistake would only become apparent when the action is executed. Writing unit tests that execute actions with some generated test data and validate the output of the actions will help in catching these kinds of errors.

5.4. Development of the browser-based tool

The REPL is a suitable environment for experimenting with the author's library, but it cannot compete with other methods that have support for tools with graphical user interfaces. As mentioned in Chapter 3.4, an executable formal specification can be used to validate the correctness and legitimacy of the specification with non-technical stakeholders. The author feels that non-technical stakeholders are likely to reject formal specification methods that don't have a tool that they can use or at least understand.

As mentioned before in this thesis, the author has created a tool with a graphical user interface for executing the formal specifications created with the author's library. The tool can be used to execute actions and visualize the returned values and the contents of the created refs. Unlike the DisCo Toolkit, the author's tool includes an editor for editing the specifications. A screenshot from the tool is shown in Picture 3.



Picture 3: The user interface of the author's tool.

The left side of the user interface is used for executing the specification and for giving instructions to the user. The largest area of the user interface on the right side contains the editor for editing the specifications. As the learning curve of Clojure and the library is steep for a programmer without suitable background, the tool can load the examples packaged with the library in the editor. It is also possible to export the created specifications as files for later use. When the specification is ready for evaluation, the user should click the *Compose* button which will evaluate the specification and starts the execution process. It is possible to write *execute* or *execute-init* commands by hand, or use a simple form to select the action, arguments, refs and the validator function. After each execution command, the returned value and the list of available refs is shown in the user interface. At any time during the execution, the user may reset the state of the tool or edit the specification and compose it again.

As discussed in Chapter 5.1, the tool is packaged into a JAR file and accessed with web browser. The back end has been written in Clojure using web-related libraries such as

Ring, *Compojure* and *HTTP Kit*. However, as a modern web application, most of the code is in the front end which is written in JavaScript. The user interface has been written using Facebook's *React* which is a simple library for creating reusable user interface components for the web [Facebook, 2015]. In addition, the tool uses *jQuery* for sending HTTP requests from the front end to the back end [The jQuery Foundation, 2015]. As implementing user interfaces with JavaScript or web services with Clojure is not in the scope of this thesis, the author will only discuss the parts of the implementation that are directly related to the domain of formal specifications.

When the user wants to compose a formal specification, the contents of the editor are sent to the back end as a string. The back end then calls the Clojure reader and compiler manually at runtime to evaluate the specification. The function that carries out the described process is presented in Code example 14.

```
(defn compose
  "Expects spec to be a formal specification with ns form in the beginning.
  If the namespace exists, erases it. Then evaluates the spec and returns
  the namespace used in the specification."
  [spec]
  (let [ns (spec/get-ns-name spec)]
    (try
      {:body (do (remove-ns (symbol ns)) (load-string spec) ns)}
      (catch Exception e (bad-request e)))))
```

Code example 14: The function that evaluates the formal specification at runtime.

The *compose* function, visible in Code example 14, gets a single argument *spec* which contains the formal specification as a string. When the function is called, the name of the namespace that the specification will use is parsed and saved to a binding named *ns*. The parsing happens in a function called *get-ns-name* which is available in the project's GitHub repository. The function then cleans the namespace *ns* with a function called *remove-ns* and evaluates the *spec* using a function called *load-string* which takes a single argument of Clojure code as a string and evaluates it [Hickey, 2015a]. The functions *remove-ns* and *load-string*, which are both available in the core namespace of Clojure, contain side-effects which makes the compose function impure. Eventually, the function returns the name of the used namespace inside of a map which will be transformed into a HTTP response and sent to the user interface.

After this process, a namespace exists that contains the actions defined in the specification. In addition, if some refs were created with *execute-init*, the namespace contains them as well. The tool may query the namespace for its contents at any time during the execution with a function called *ns-publics* which returns the *public interns* of the given namespace [Hickey, 2015a]. This is where the metadata attached to the actions and refs becomes useful: the tool is able to filter the unwanted functions and refs when it produces a list of available actions and refs related to the execution. The functions that can be used to filter lists of unwanted vars are presented in Code example 15.

```
(defn action-entry?
  [map-entry]
  (contains? (meta (val map-entry)) :action))

(defn spec-ref-entry?
  [map-entry]
  (contains? (meta (val map-entry)) :spec-ref))
```

Code example 15: Functions that identify actions and specification related refs from the output of *ns-publics*.

As *ns-publics* returns a map, the functions in Code example 15 get map entries as arguments. Both of the functions apply the *meta* function to the value of the map entry which returns the metadata of the intern. The *contains?* function is used to check if the metadata contains either the *:action* or the *:spec-ref* key which can be found only in the interns that were created with macros *defaction* or *execute-init* respectively.

The author has presented how the tool is able to evaluate formal specifications in their own namespaces and fetch the actions and refs created in those namespaces for the user to observe. The last functionality that the author wants to discuss is the execution of the actions in the back end of the tool. As the user interface is used to form the commands containing calls to either *execute* or *execute-init*, the back end has no other role but to evaluate the received form in the correct namespace. The function that does this is presented in Code example 16.

```
(defn execute-with-ns
  "Evaluates command in namespace ns. Command should be a valid Clojure form
  with execute or execute-init. Returns the result of the evaluation as
  string."
```

```
[ns command]
(try
  {:body (str (binding [*ns* (find-ns (symbol ns))] (load-string command)))}
  (catch Exception e (bad-request e))))
```

Code example 16: Function that evaluates a Clojure form in a certain namespace.

The function *execute-with-ns* shown in Code example 16 returns a map with a key *:body* similarly to the function *compose* in Code example 14. The formation of the value for the *:body* may seem complex even for an experienced Clojure programmer and therefore requires some explanation. Normally, Clojure forms are evaluated in the current namespace which is referred to with a symbol **ns** [Hickey, 2014]. In order to evaluate the *command* in another namespace, a macro called *binding* is used to temporarily change the **ns** object to refer to another namespace [Hickey, 2015a]. The target namespace is resolved by transforming the *ns* string into a symbol which is then passed to a function called *find-ns*. Like in Code example 14, *load-string* is used to evaluate the command. Because of the temporary **ns** binding, the evaluation happens in the same namespace where the original specification was evaluated which is why the actions and refs are available during the evaluation of the *command*. The return value of the evaluated form is returned to the front end as a string.

The current implementation of the tool is missing a lot of important features that the animator of the DisCo Toolkit has. For example, the animator is able to repeatedly execute a set random actions. This is a nice feature that can be used to forcefully find deadlocks or violations of the safety property. In addition to this feature, the tool should include some sort of visualization for the history of executed actions. The tool can currently show the history of executed commands, but a more powerful version should be implemented that could visualize how the data has changed when it moved from an action to action. Overall, the author thinks that the tool shows a lot of potential as the current implementation of the tool already includes the most important features. In addition, because of the modern web-based open-source technologies, new features are easy and cheap to develop. The tool is also executable with any modern computer and operating system that has Java installed.

5.5. Evaluation of the outcome

The result of the author's experiment is a custom formal specification method that consists of a library and a tool. The created library cannot be defined as a domain-specific language as the language that it used with the library is just the normal Clojure. It is also not a framework as the control flow of the programs is not determined by it [Riehle, 2000]. The

created library is just a set of functions and macros that add more features to Clojure. The author wants to emphasize that the library does not modify the existing functionality of Clojure.

The goals set by the author in the beginning of this chapter were achieved for the most part. First of all, the formal specifications created with the author's library are indeed executable. The author targeted the library and the tool for Clojure developers. This goal is achieved as the syntax of the specifications is easy to grasp for Clojure programmers, and the tool lowers the threshold of adopting formal specifications in real-life software projects.

The author also set a goal for creating a formal specification method that allows transforming the specifications into implementation. This goal was achieved as well. However, the difficulty of the transformation depends on the user's ability to write modular specifications: if the actions are separated from the pure functions that manipulate data, the transformation is a simple task of copying the pure functions from the specification into the implementation. It is also possible to transform the actions themselves into normal functions that can be used in the implementation. The developers have to just write a function for each action, and transform the availability of each of those actions into the pre- and postconditions of the new functions. Otherwise, the functions and the actions are similar: the name, the arguments and the body do not need any changes in the transformation. In fact, the author suspects that it is possible to automate the transformation process by implementing a macro that can transform the actions into normal functions. The developers would then have to implement only those parts of the system that were not included in the formal specification. These parts include components such as the user interface and integrations to external interfaces.

In addition, the experiment embodies the advantages and disadvantages of using Clojure to write formal specifications. The examples presented in this thesis prove that the method is a credible alternative to Z and DisCo at least in small projects. Although not tested, the experiment has not revealed any reasons for why the author's method would not be suitable for large real life projects.

Some of the goals set for the experiment were not achieved. For now, one of the biggest drawbacks of the tool is its unsafe nature: as the tool evaluates the specification without any safety checks in the host machine, users with malicious intentions can do a lot of damage to host machines that run the author's tool in public networks. This means that the goal of creating an application that could be run as a service in the network was not achieved.

One way to solve this problem is to run the tool inside of a safe container. Different kinds of automated virtualization tools such as Vagrant [Hashicorp, 2015] or Docker [Docker, 2015] could be used for providing a safe container for the application to run. If a malicious specification would destroy or take over the container, the admin of the host machine could simply recreate it in a matter of seconds. Another way to solve this problem is to utilize the sandboxing features of Java Virtual Machine. In fact, there is Clojure library created for this specific purpose called *Clojail*. It can be used to create a blacklist of functions and other operations that would be prohibited to execute inside the sandbox where the application would be running [Grimes, 2013].

In the future, one way to provide security for the tool is to use ClojureScript which is a compiler that compiles Clojure into JavaScript [Hickey, 2015b]. By using ClojureScript, it would be possible to compile the specifications into JavaScript and execute them in the front end side. This approach would make it unnecessary to evaluate the specification in the back end which would eliminate the security issue of running malicious code in the host machine. At the time of writing, the problem is that ClojureScript does not support refs or software transactional memory which are the cornerstones of the author's solution [Hickey, 2015b].

Another drawback of the author's current method is the lack of automated formal verification and analysis mechanisms. By using the validator functions, the user of the library is able to find the violations of the safety property. However, this is manual labor based on simulating the system defined in the specification. The current version of the library itself has no features that help conducting a true formal verification for the specifications.

In addition, testing other properties besides safety such as liveness, fairness or progress is currently not supported even by animating the specification. An easy, but rudimentary way to provide some analysis for the specifications would be to implement a function that randomly executes actions for a specific amount of time as mentioned in Chapter 5.4. Of course, with time, it would be possible to develop a theorem prover or at least a support for it for more elegant approach. Another way to provide support for formal verification would be to implement a feature or a tool that could export a formal model from the specification for some existing formal verification tool such as the LTSA.

Because refs are used to store data, it may seem that the author's method does not make use of the pure nature of functional programming. This is true only in cases where actions take refs as arguments and manage their contents and lifecycle directly. This approach is needed only in situations where multi-object actions are required, and using a single map to store the whole state of the system does not seem reasonable. In other cases,

the library conceals the handling of refs which allows the user of the library to concentrate on writing pure functions and actions. In complex cases, where multi-object actions are required, the author recommends writing the specifications using a multi-layer architecture where the pure functions that manipulate the data are separated from the actions that handle the transactions. This approach can be seen in the formal specification example in Appendix 2.

Because the library exposes all of Clojure, it is clear that the author's method does not suffer from lack of features. The user of the library is able to utilize every function, macro and library available for Clojure, and even use the large pool of Java libraries due to the interoperability. The author suspects that the formal specifications written with the library don't benefit much from this possibility. However, it makes it possible to extend the library or the tool with all kinds of features. For example, Nummenmaa and Nummenmaa [2011] have presented an idea of using databases for formal specifications to fix the disadvantages of DisCo, and to allow several simultaneous processes to execute the same specification. Because of Clojure and its ecosystem, it would be a straightforward task to implement a database support to the author's library.

Sanders and Johnson [1990b] refined their executable formal specifications and prototypes partly because of the poor performance. Even though measuring the performance of the formal specifications created with the author's library is not in the scope of this thesis, the author would like to point out that the performance of the specifications depends directly on the ability of the user to write efficient Clojure code. The author would like to point out that even though Clojure programs run on top of the Java Virtual Machine, the performance may vary from Java programs because the Clojure compiler does not generate identical byte code with the Java compiler.

Clojure has proven to be an excellent language for implementing formal specification systems as already predicted in Chapter 2.3. The core namespace of Clojure includes a lot of useful functions such as *load-string*, *binding* and *meta* which can be used together with macros to extend the language at runtime. This claim is not based on the author's personal experience alone: without examples, tests and comments, the author's library has only 46 lines of code which is an impressive result that proves the capabilities of the language. Even with the comments and tests, the library still has less than 200 lines of code. The sparseness of the code does not apply to the tool as most of it has been written in JavaScript.

Even though Clojure is a good language for developing formal specification systems, it is not the best language for writing the specifications itself because of the duck typing. The author has noticed a common pattern where a formal specification written using the

author's library evaluates without errors but won't execute without exceptions. Usually, these exceptions are related to duck typing. As mentioned earlier in this chapter, unit tests can be used to provide checks for these kinds of mistakes. The author's opinion is that writing unit tests to validate formal specifications is not a bad idea in general. However, writing unit tests to catch errors that the compiler can catch in statically typed languages is a lot of unnecessary work. As noticing errors early in the development process is one of advantages of using formal specifications, using some other language with a compiler that takes care of the large part of the validation seems more reasonable.

The idea of using programming languages to write formal specifications is a controversial topic. The author has cited sources such as Palshikar [2001] in Chapter 3.3 for arguing that formal specifications should not include implementation level details. It is reasonable to presume that writing formal specifications with an intention of transforming them into implementation bounds to break this exhortation. The same argument is emphasized by Diller [1994] who argues that the same language should not be used for formal specifications and the implementation. Diller [1994] justifies his argument by stating that formal specifications and programming are fundamentally different activities, and that it is inappropriate to add programming language constructs to formal specification languages and vice versa.

The author himself is currently working with a real-life Clojure project, and is using the library and the tool presented in this thesis. As Clojure is going to play even bigger role in the author's future, the tool will be exposed to even more developers. In addition, as the library and the tool is free to use and open-source, its use may spread to other companies as well. However, due to mentioned controversy and the author's probable lack of objectivity for his own work, the task of evaluating the author's experiment and the premise itself is ultimately left to the reader of this thesis.

6. Conclusions

Programming and formal specifications share many similarities. Both of them are based on the usage of formal languages and have different styles and paradigms with different advantages and disadvantages. In addition, the evolution of one activity shapes the other. This is clearly visible in the relationship between DisCo and object-oriented programming. However, there is still a clear gap between implementing software and formal specifications. Some effort has been put into closing this gap, but unfortunately the popularity of imperative and object-oriented programming paradigms have received most of the attention on the subject even though programs written in functional programming languages are naturally closer to formal specifications.

The possibilities of combining the functional programming paradigm with formal specifications has not been researched enough. The research about the subject is focused on using existing functional programming languages for writing formal specifications. Although the amount of research on the subject is not that large, the results are consistent: functional programming languages are suitable for writing formal specifications. Referential transparency makes it easier to reason with the language, and the specifications are executable by nature which allows animating the specification and simulating the specified system. In addition, writing the formal specification with the same language that will be used in the implementation makes it possible to transform the specification into implementation with a few simple steps.

The author has created a formal specification method based on using Clojure for writing and executing the formal specifications. The author's method is targeted for Clojure programmers, which reduces the threshold of adopting formal specifications in Clojure projects. In addition, the developers don't need to have skills in mathematical proving, as the specifications created with the author's method are meant to be validated by executing them instead of performing proofs. It also provides a modern view and implementation for formal specification systems, which will be required in the coming years to replace the legacy systems such as the DisCo Toolkit.

The author of this thesis has found Clojure to be an excellent language for developing libraries and tools for formal specifications. Clojure has extensive metaprogramming capabilities due to homoiconity and macros which allow extending the language with new notations and features. Clojure's support for the software transactional memory is also helpful as it can be used to check the violations of the safety properties, and guarantee the atomicity of multi-object actions. In addition, the core namespace of Clojure contains a lot

of useful functions for turning textual formal specifications into executable bytecode at runtime.

The author had hoped that the industry's increasing interest in Clojure and the promise of cost-free adoption would raise the interest of the industry to adopt formal specifications. However, after writing this thesis, the author does not feel that Clojure is the best language for writing formal specifications. Because of the duck typing, a lot of errors in the specifications go unnoticed when the code is compiled which is not a good trait for a method whose primary purpose is to catch errors early.

There is a lot of room for further research. The first step would be to have someone else than the author objectively analyze the author's method. This analysis would reveal whether or not the results of the experiment can be generalized, and if it is reasonable to continue developing the author's library and tool. If the feedback for the author's work would be positive, it would be possible to finalize the tool to provide the security that is required for running the tool in public networks. Some support for formal verification methods could also be implemented. The tool could also be developed further to support other notations such as the DisCo language. In fact, it would be possible to develop a platform for action-based formal specifications that would support multiple notations.

If the author's solution would present more problems than it would solve, it would be an interesting experiment to implement a similar or completely different kind of library with Haskell in order to compare it with the author's Clojure-based solution. The future research could also take a different approach all together. For example, it could focus on the transformation of formal specifications into implementation. As long as the future research focuses on making formal specifications more interesting for all programmers, the author is satisfied.

References

[Bernstein and Goodman, 1983] Philip A. Bernstein and Nathan Goodman, Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems* **8**, 4 (Dec. 1983), 465-483.

[Butler, 1995] Jenny Butler, Use of a functional programming language for formal specification. In: *Practical Application of Formal Methods, IEE Colloquium on*, IET, 2/1 – 2/3.

[Cardelli, 1996] Luca Cardelli, Bad engineering properties of object-oriented languages. *ACM Computing Surveys (CSUR) - Special issue: position statements on strategic directions in computing research* **28**, 4 (Dec. 1996).

[The Disco Project, 2002] The Disco Project, The Disco Home Page, 2002. Retrieved Mar. 29, 2015 from: http://disco.cs.tut.fi/General.html

[Diller, 1994] Antoni Diller, *Z: An Introduction to Formal Methods Second Edition*. Wiley, 1994.

[Docker, 2015] Docker, Docker - Build, Ship, and Run Any App, Anywhere, 2015. Retrieved Apr. 29, 2015 from: http://www.docker.com/

[Emerick *et al.*, 2012] Chas Emerick, Brian Carper and Christophe Grand, *Clojure programming*. O'Reilly Media, 2012.

[Facebook, 2015] Facebook, A JavaScript library for building user interfaces | React, 2015. Retrieved Apr. 26, 2015 from: https://facebook.github.io/react/index.html

[Fogus, 2012] Michael Fogus, Pre and post conditions are not respected in the instarepl, Issue #163, LightTable/LightTable, 2012. Retrieved May 1, 2015 from: https://github.com/LightTable/LightTable/issues/163

[Fogus and Houser, 2011] Michael Fogus and Chris Houser, *The Joy of Clojure*. Manning, 2011.

[Gaudel, 1994] Marie-Claude Gaudel, Formal specification techniques. In: *Software Engineering, 1994. Proceedings. ICSE-16., 16th International Conference on*, IEEE, 223-227.

[Google, 2015] Google, The Go Programming Language, 2015. Retrieved Apr. 1, 2015 from: https://golang.org/

[Grimes, 2013] Anthony Grimes, Raynes/clojail, 2013. Retrieved Apr. 29, 2015 from: https://github.com/Raynes/clojail

[Hall, 1998] Anthony Hall, What does industry need from formal specification techniques? In: *Industrial Strength Formal Specification Techniques, 1998. Proceedings. 2nd IEEE Workshop on*, IEE, 2-7.

[Halloway and Bedra, 2012] Stuart Halloway and Aaron Bedra, *Programming Clojure, Second Edition*. Pragmatic Bookshelf, 2012.

[Hashicorp, 2015] Hashicorp, Vagrant, 2015. Retrieved Apr. 29, 2015 from: https://www.vagrantup.com/

[Henderson, 1986] Peter Henderson, Functional programming, formal specifications and rapid prototyping. *IEEE Transactions on Software Engineering* **12**, 2 (Feb. 1986), 241-250.

[Hickey, 2009] Rich Hickey, Are we there yet, Nov. 12, 2009. Retrieved Jun. 2, 2014 from: http://www.infoq.com/presentations/Are-We-There-Yet-Rich-Hickey.

[Hickey, 2013] Rich Hickey, Clojure core.async Channels, Jun. 28, 013. Retrieved Apr. 1, 2015 from: http://clojure.com/blog/2013/06/28/clojure-core-async-channels.html

[Hickey, 2014] Rich Hickey, Clojure, 2014. Retrieved Jun. 14, 2014 from: http://clojure.org/.

[Hickey, 2015a] Rich Hickey, Clojure/core.clj at master, 2015. Retrieved May 3, 2015 from: https://github.com/clojure/clojure/blob/master/src/clj/clojure/core.clj

[Hickey, 2015b] Rich Hickey, Differences from Clojure, clojure/clojurescript wiki, 2015. Retrieved Apr. 29, 2015 from: https://github.com/clojure/clojurescript/wiki/Differences-from-Clojure

[Hinsen, 2009] Konrad Hinsen, The promises of functional programming. *Computing in Science & Engineering* **11**, 4 (July-Aug. 2009), 86-90.

[Hinsen, 2012] Konrad Hinsen, Managing state. *Computing in Science & Engineering* **14**, 1 (Jan.-Feb. 2012), 80-86.

[Hughes, 1989] John Hughes, Why functional programming matters. *The Computer Journal – Special issue on Lazy functional programming* **32**, 2 (April 2009), 98-107.

[Joda.org, 2014] Joda.org, Joda-Time – Java date and time API – User Guide, 2014. Retrieved May 16, 2014 from: http://www.joda.org/joda-time/userguide.html.

[The jQuery Foundation, 2015] The jQuery Foundation, jQuery.ajax() | jQuery API Documentation, 2015. Retrieved Apr. 26, 2015 from: http://api.jquery.com/jquery.ajax

[Kurki-Suonio, 2005] Reino Kurki-Suonio, *A Practical Theory of Reactive Systems, Incremental Modeling of Dynamic Behaviors*. Springer, 2005.

[Lamsweerde, 2000] Axel van Lamsweerde, Formal specification: a roadmap. In: *ICSE '00 Proceedings of the Conference on The Future of Software Engineering*, ACM, 147-159.

[Lano, 1991] Kevin Lano, Z++, an object-oriented extension to Z. In: John Nicholls, *Z User Workshop, Oxford 1990*. Springer London, 1991, 151-172.

[Larsen *et al.*, 1996] Peter Larsen, John Fitzgerald and Tom Brookes, Applying formal specifications in industry. *IEEE Software* **13**, 3 (May 1996), 48-56.

[Läufer, 2009] Konstantin Läufer, The promises of typed, pure and lazy functional programming: part II. *Computing in Science & Engineering* **11**, 5 (Sept.-Oct. 2009), 68-75.

[Hagelberg, 2014] Phil Hagelberg, Leiningen Tutorial, 2014. Retrieved Apr. 17, 2015 from: https://github.com/technomancy/leiningen/blob/stable/doc/TUTORIAL.md

[Lewis and Loftus, 2011] John Lewis and William Loftus, *Java Software Solutions: Foundations of Program Design (7th Edition)*. Addison-Wesley, 2011.

[Lightfoot, 2001] David Lightfoot, *Formal Specification Using Z*. Palgrave, 2001.

[Lipovača, 2011] Miran Lipovača, *Learn You a Haskell for Great Good!*. No Starch Press, 2011.

[Magee, 1997] Jeff Magee, FSP-notation, 1997. Retrieved Mar. 26, 2015 from: http://www.doc.ic.ac.uk/~jnm/LTSdocumention/FSP-notation.html

[Magee and Kramer, 2006] Jeff Magee and Jeff Kramer, *Concurrency: State Models & Java Programs Second Edition*. Wiley, 2006.

[McCarthy, 1960] John McCarthy, Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM* **3**, 4 (April 1960), 184-195.

[Misra, 2001] Jayadev Misra, *A Discipline of Multiprogramming: Programming Theory for Distributed Applications*. Springer Science & Business Media, 2001.

[Nummenmaa and Nummenmaa, 2011] Jyrki Nummenmaa and Timo Nummenmaa, Database-driven tool support for DisCo executable specifications. In: *Proceedings of 12th Symposium on Programming Languages and Software Tools (SPLST'11)*, TUT Press, 44-54.

[Palshikar, 2001] Girish Keshav Palshikar, Applying formal specifications to real-world software development. *IEEE Software* **18**, 6 (Nov. 2001), 89-97.

[Pfleeger and Hatton, 1997] Shari Lawrence Pfleeger and Les Hatton, Investigating the influence of formal methods. *Computer* **30**, 2 (Feb. 1997), 33 – 43.

[Potok *et al.*, 1999] Thomas Potok, Mladen Vouk and Andy Rindos, Productivity analysis of object-oriented software developed in a commercial environment. *Software—Practice & Experience* **29**, 10 (Aug. 1999), 833-847.

[Régis-Gianas and Pottier, 2008] Yann Régis-Gianas and François Pottier, A Hoare logic for call-by-value functional programs. In: Philippe Audebaud and Christine Paulin-Mohring, *Mathematics of Program Construction, 9th International Conference, MPC 2008, Marseille, France, July 15-18, 2008. Proceedings*. Springer Berlin Heidelberg, 2008, 305-335.

[Riehle, 2000] Dirk Riehle, *Framework Design: A Role Modeling Approach*. Ph.D. Thesis No. 13509. Zürich, Switzerland, ETH Zürich, 2000.

[Sanders and Johnson, 1990a] Paul Sanders and Michael Johnson, From Z-specifications to functional implementations. In: John Nicholls, *Z User Workshop, Oxford 1989*. Springer London, 1990, 86-112.

[Sanders and Johnson, 1990b] Paul Sanders and Michael Johnson, Functional languages for the implementation of formal specifications. In: *UK IT 1990 Conference* (1990), IET, 213-220.

[Sanghavi, 2010] Alok Sanghavi, What is formal verification?, 2010. Retrieved May 8, 2015 from: http://www.eetasia.com/STATIC/PDF/201005/EEOL_2010MAY21_EDA_TA_01.pdf?SOURCES=DOWNLOAD

[Schach, 2010] Stephen Schach, *Object-Oriented and Classical Software Engineering*. McGraw-Hill Science/Engineering/Math, 2010.

[Seibel, 2009] Peter Seibel, *Coders at Work: Reflections on the Craft of Programming*. Apress, 2009.

[Smith, 2000] Graeme Smith, *The Object-Z Specification Language*. Springer, 2000.

[Sommerville, 2009] Ian Sommerville, Sofware Engineering 9, 2009. Retrieved Jun. 29, 2014 from: http://ifs.host.cs.st-andrews.ac.uk/Books/SE9/WebChapters/PDF/Ch_27_Formal_spec.pdf.

[Sussman and Steele, 1975] Gerald Jay Sussman and Guy L. Steele Jr., *Scheme: an interpreter for extended lambda calculus. Artificial Intelligence Memo 349*. Massachusetts Institute of Technology, 1975.

[TIOBE Software, 2014] TIOBE Software, TIOBE Index for June 2014, Jun. 2014. Retrieved Jun. 25, 2014 from: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

[Zave and Jackson, 1996] Pamela Zave and Michael Jackson, Where do operations come from? A multiparadigm specification technique. *Software Engineering, IEEE Transactions on* **22**, 7 (Jul. 1996), 508-528.

The core namespace of the author's library

```
(ns clj-formal-specifications.core)

(defmacro defaction
  "Like defn in style, but is used to define functions that return executable
  formal specification actions. action-map must be a map which will be the
  return value of by the defined function with all the values wrapper into
  closures."
  ([name args action-map]
   {:pre [(map? action-map)]}
   `(defn ~(with-meta name {:action true}) ~args
      ~(reduce-kv #(assoc %1 %2 `(fn [] ~%3)) {} action-map)))
  ([name doc-string args action-map]
   {:pre [(map? action-map) (string? doc-string)]}
   `(defn ~(with-meta name {:action true}) ~doc-string ~args
      ~(reduce-kv #(assoc %1 %2 `(fn [] ~%3)) {} action-map))))

(defn action?
  "Returns true if action is a valid action with body."
  [action]
  (and (contains? action :body) (fn? (:body action))))

(defn available?
  "Returns true if action is available for execution."
  [action]
  (if (contains? action :available)
    (if (fn? (:available action)) (boolean ((:available action))) false)
    true))

(defn- test-action
  "Throws exceptions if action cannot be executed. Used to give reasonable
  error messages for why executing an action has failed."
  [action]
```

```
  (cond
     (not (action? action))
       (throw (Exception. "given parameter is not a valid action."))

     (not (available? action))
       (throw (Exception. "action is not available for execution"))

     :else true))

(defn execute
  "If action is available and well-formed, executes its body and returns the
  result. If ref is given, the return value will be also stored into the ref.
  See execute-init for creating the ref."
  ([action]
  (if (test-action action) ((:body action))))
  ([action ref]
   {:pre [(instance? clojure.lang.Ref ref)]}
   (dosync (ref-set ref (execute action)))))

(defmacro execute-init
  "Calls execute normally for the given action, but stores the returned
  value into a ref. A var is created with the name var-name which refers to
  the ref. An optional validator function can be given to the ref.
  See set-validator! function and Clojure documentation about refs for more
  details."
  ([var-name action-expr]
  `(def ~(with-meta var-name {:spec-ref true}) (ref (execute ~action-expr))))
  ([var-name action-expr validator]
   `(def ~(with-meta var-name {:spec-ref true})
      (ref (execute ~action-expr) :validator ~validator))))
```

An example formal specification of a shared account

```
(ns clj-formal-specifications.examples.shared-account
  (:require [clj-formal-specifications.core :refer :all]))

; Standard functions
(defn person
  [id amount]
  {:id id :wallet amount})

(defn valid-person?
  [p]
  (not (neg? (:wallet p))))

(defn apply-to-wallet
  "Calls f with the :wallet of p and the amount and updates the :wallet with
  the returned value."
  [p amount f]
  (update-in p [:wallet] f amount))

(defn account
  [amount owner-ids]
  {:balance amount :owners owner-ids})

(defn valid-account?
  [acc]
  (and (not (neg? (:balance acc))) (not (empty? (:owners acc)))))

(defn apply-to-balance
  "Calls f with the :balance of acc and the amount and updates the :balance
  with the returned value."
  [acc amount f]
  (update-in acc [:balance] f amount))
```

```
; Actions
(defaction create-person
  [id amount]
  {:body (person id amount)})

(defaction create-account
  "Creates an account with owners from the :id of owners-refs, if owner-refs
  is a collection and not empty."
  [owner-refs]
  {:available (and (coll? owner-refs) (not (empty? owner-refs)))
   :body (account 0 (map (comp :id deref) owner-refs))})

(defaction withdraw
  "Transfers money from an account to a person, if the person owns the
  account and the account has enough balance."
  [account-ref person-ref amount]
  {:available (and (>= (:balance @account-ref) amount)
                   (some #{(:id @person-ref)} (:owners @account-ref)))
   :body (dosync (alter account-ref apply-to-balance amount -)
                 (alter person-ref apply-to-wallet amount +))})

(defaction deposit
  "Transfers money from a person to an account, if the person has enough
  money."
  [account-ref person-ref amount]
  {:available (>= (:wallet @person-ref) amount)
   :body (dosync (alter account-ref apply-to-balance amount +)
                 (alter person-ref apply-to-wallet amount -))})
```