Concurrent Brzozowski DFA construction using Erlang

and how it turned out to be a Google MapReduce algorithm

Hendrik Visagè
SN:91211108

October 24, 2010

Chapter 1

Raison d'etre

1.1 Becoming intrigued

Earlier this year, the author noticed several job advertiments for Erlang programmers on a South African job advertising website. Investigating this further, the author became quite intrigued by the Erlang language claims about concurrency and availability, and even more so because Erlang is a functional programming language. This alone is not a good enough reason to learn a language, especially when there is no task or project per se to code in the language. For that we need to find a reason to do a task in Erlang. But before we get to that question problem, let us examine some of the state of some aspects of Computer Science.

1.1.1 Today's CPUs

Concurrent algorithms are becoming more important lately as the commodity CPUs shipped on laptops and desktops, are nearly without exception multi-cored or multithreaded. **?** already claimed in 1985 that massive parallelism is the future of computing, and it obviously have become a trend with CPUs when Intel's Hyperthreading, Sun's CoolThread CPUs and multi-core AMD Opteron CPUs got introduced. The idea is to rather have more processing units available than to try and raise the core clockspeeds. This apparently helps to keep CPUs cooler, and provide users more processing power. However, this necesitates the need for concurrent algorithms and parallel processing to be able to effectively and efficiently use the processing power available in these processing units, as the core clock is not faster, but the number of processing units increased. The Oracle[1] SPARC T3 is at the extreme end at present with 128 threads across 16cores on a 1.65GHz CPU socket.[2]

[1]previously Sun Microsystems

[2]http://www.oracle.com/us/products/servers-storage/servers/sparc-enterprise/t-series/sparc-t3-171613.html accessed 17 October 2010

1.1.2 Determinate Finite Automata (DFA)

DFAs as a matching algorithm have big importance in the matching of patterns. These patterns could be virus signatures, DNA or even network based intrusion detection and prevention. These DFAs are constructed from regular expressions and as these regular expressions become more complex and extended, it is but natural to ask how the DFA construction could be made faster using the available multi-core and multi-threaded CPUs, and that is the reason for the research into the concurrency of this algorithm.

We will have to point out that in this research, we focussed on the construction of the DFA from an expression, and not the actual DFA application to the data to be matched, nor will this study look into the parsing of the regular expressions into expressions useful for our DFA construction.

1.1.3 Erlang

Erlang in a functional language and for programmers used to procedural languages, there is a couple of interesting features (or some might say annoyances) that would make it at least a learning experience to guage the language. Armstrong boosts about Erlang's built-in concurrency features, and this would be a perfect match to test both the language on multicore CPUs.

1.2 Research focus

The problem this study addresses, is some research into the concurrency possibilities of the sequential Brzozowski algorithm and to implement this in Erlang. This way we will be combining the concurrent features of todays CPUs, the Erlang language that boasts about its concurrency features as well as the quest for a concurrent Brzozowski DFA construction.

The rest of this document will first look at Erlang and its feature set and what makes Erlang (and to some degree functional languages) different from other programming languages especially its support for concurrency. Then we will look at the Brzozowski sequential algorithm and some parallelization proposals. Lastly we will discuss the Erlang implementations for the parallelizations proposals.

Chapter 2

Erlang - the language

2.1 Introduction

In this chapter, we will give some brief overviews to Erlang's history and it's language constructs. We will also explain those constructs and ideas that we have used in this project as well as those constructs that is not obviously the same as in other computer languages like the C/C++ languages. For the purposes of explanation in later parts, we will briefly give an Erlang language introduction to some percularities. However, for proper detailed explanations we will refer the reader to **?**.

2.2 Brief history

Armstrong **?** gives a detailed historical overview of Erlang since it's inception in 1986 till his thesis circa 2001. In summary it started in 1981 from the goal *"to suggest new architectures, concepts and structures for future processing systems development"*. Armstrong started to work on adding concurrent processes to Prolog and the language evolved away from Prolog and evolved with its own abstract machines (virtual machines in the JAVA terminology). Erlang have been used in the AXD301 switch which superceded the failed AXD-N switch and it is claimed to have a **NINE** nines[1] uptime in production, while still having around 2million lines of Erlang code.

The main theme found in Erlang is to have small parts that shares nothing while working together, and if any of these small parts experience problems it can not handle, it rather fail fast and let a supervisor process handle the failure and to restart the failed process.

This fail fast is especially nice as there is not suppose to be any shared memory between processes/parts, which means that a failure in one process

[1]99.9999999% where the usual target for mission critical systems is 5 nines (99.999%) while vendors (from the authors experiences) do not easily target nor claim uptimes higher than 95%

should not impact the other processes by corrupting shared memory. This is quite a different approach from other threading models like the C based Posix threads, where process memory (and thus variables) are shared, and thus have a need for locks and other mutual exclusion methods to prevent the threads from concurrently accessing memory, and thereby corrupting data.

The author would like to point out that this is different from guaranteed and proven correctness used in software development for critical software used in aplications like the space shuttles that can not tolerate any glitches, where as the Erlang model tolerates the glitches by restarting the processes.

2.3 Quick Language Introduction

In this section, we will briefly introduce the reader to the Erlang language. This should be sufficient to be able to grasp the code presented in this research, and it will not be a detailed reference. The reader are referred to Armstrong**?** or O'Reilly**?** for further in depth explanations and references to the Erlang language.

2.3.1 atoms and Variables

Erlang distinguishes between atoms and variables mostly by the first character being uppercase for variables or a lowercase character for atoms.[2]

Erlang's atoms are similar to C/C++ `enums`, just more generic and not typed like the C/C++ enums which are but typed numbers.

2.4 Functional language features

This section we will briefly glance over some of Erlang's peculiar[3] language features to give the reader a grasp of the expressive power that helped to produce the programs in such short time.

2.4.1 Pattern Matching - function overloading

One of the strengths of Erlang (and the author understood other functional languages too, but have not investigated that) is the way pattern matching is used for code flow paths. Program listing 2.4.1 shows this feature with the two functions `area/2`, `area/3` and `area/4`. Remember the atoms start with lowercase letters while the Variables that gets bound to a value, starts with an uppercase letter.

[2]Yes, there are exceptions but that means quoting etc. which have not being used in our code

[3]compared to the C and other procedural type languages

Program 2.4.1 Pattern matching in code flow

```erlang
area(square,Side) -> Side*Side;
area(cube,Side) -> area(square,Side)*6;
area(circle,Radius) -> area(circle,radius,Radius).

area(circle,radius,Radius) -> Radius*Radius*3.14;
area(circle,diameter,Diameter) -> 
    area(circle,radius,Diameter/2);
area(triangle,Base,Height) -> Base*Height/2;
area(rectangle,Height,Width) -> Height*Width.

area(box,Height,Width,Depth) -> ((Height*Width) +
      (Height*Depth) + (Width*Depth))*2.
```

As could be seen in this example, that we used multiple functions (and have them match based on the parameters) rather than having if-then-else or case/switch statements to make code flow decisions. The different distributor states is also handled using these parameter matching.

Guards Another code flow technique is the use of guards (`when` statements) inside functions. These help firstly with pre-conditions (ie. to force only accepting valid values, like positive values for distance), and secondly with another method of conrolling the flow of code, but only after the parameters have been matched (Ie. a parameter could match anything, but we want to handle the circle and square different).

These same pattern matching and guards is extended to the message receiving discussed in section 2.5.1 and shown in program 2.5.1

Something else to note here, is that the underscore denotes a parameter who's value will be unbounded and ignored. Sometimes a variable with a prepended underscore would be a way to name a variable that would not be used, to prevent compiler warnings.

Notation of functions A convention in the Erlang texts, is to refer to **module:function/arity** for a function for example `lists:map/2` which is read as

module lists

function name map

arity taking 2 parameters

2.4.2 Functions as first class members

By definition a function in a functional language is a first class member, where a function can be passed around like a variable. This do allow for interesting concepts where you have a function definition inside a function call, for example to map a list to its squares, we use something like :

```
lists:map(   fun(X) -> X*X end,
             [1,5,3] )
```

Here we provide a list with elements `[1, 5, 3]`, and `map/2` take each element of that list, apply the provided function (in this case `fun(X) -> X*X`) to that element, and returns a list with the new values `[1, 25, 9]`.

2.4.3 Imutable variables

Variables in Erlang is like algebraic variables that have a fixed value during a run of a function block. For example, once you have bound $X = 1$ and then evaluate $Y = X + 2$ we will have $Y == 3$ and we can not have $X = 2$, later on in that run as $X == 1$ from the first assignment. This prevents side effects from C/C++ constructs like `y=x++`.

The other term that is used instead of assignment, is binding, as a variable gets bound to a value, can can't be unbounded to take on a new value during that run.

Having programmed mostly before in procedural C-type languages, this feature of functional languages have initially had an annoying impact on the thought pattern when trying to grasp the workings of the language, but once grasped the author found it to be natural while programming in Erlang.

2.4.4 Tail recursion

Tail recursion is achieved when the compiler can optimize the code to be a `goto/jump`[4] back to the beginning of the function, perhaps with new parameters. This way there is no returning stack needed that would build up.

One of the important reasons for this feature, is that we can write infinitely recursive servers (functions) without having any memory leaks. This will be shown in some of the techniques used to produce our distributor and receivers in 3.2.2 and 3.2.2 without stack space being used.

Program 2.4.2 shows proper tail recursion examples, where the last instruction calls in a flow to `loop/1` is tail-calls.

[4]Yes we all *know* that is a BadThing™ but still CPUs consistent of those instructions and here is a nice GoodThing™ use for them

Program 2.4.2 Right Tail-Recursion

```
loop(0) -> true;
loop(N) when N > 0 ->
 io:format(" iteration: ~p ~n",N),
 loop(N-1).
```

Program 2.4.3 show two cases where it is not possible to use tail recursion by the compiler. The first `loop/1` is called before the output, and this means that it needs to return to that spot to do the rest of the work in that function. The `factorial/1` function also needs to return a value, so yet again this is not proper tail recursion and would need to be rewritten for tail recursion.

Program 2.4.3 No Tail-Recursion

```
loop(N) when N > 0 ->
  loop(N-1),
  io:format(" iteration: ~p ~n",N).

factorial(0) -> 1;
factorial(N) -> N*factorial(N-1).
```

Armstrong (**??** and **??**) as well as Cesarini and Thompson **??**, have in depth discussions and examples related to tail recursion, but for the purposes of this project, the above will suffice.

2.5 Concurency and distributed programming

Armstrong**?** coined the phrase *Concurrency Oriented Programming* to describe how Erlang helps a program to be structured around the concurrency of the application. Armstrong**?** also states that the world is a concurrent place and that in the real world, even though we do things concurrently, we do not share memory as do most threading models in languages like C/C++. As such Erlang is structured so that no process share memory with another process.

What makes this idea of *share nothing* powerful, is that Erlang implements the messaging communication such that both concurrent and distributed processes, communicate in the exact same way. In other words, once you know have the reference PID of the process on the remote node, you can sent a message to it as if it is local, and the response from the remote process can come back to you , without the remote processes knowing whether a local or remote process messaged it.

To create a process[5] in Erlang, we use the `spawn(Fun) -> Pid()`, and to start it on a different (connected) node we use `spawn(Node,Fun)-> Pid()`[6]. As can be seen, both returns a PID to be used for checking and for messages sent to the processes. This makes starting a process locally or distributed just a matter of specifying where, rather than several elaborate methods.[7]

Thus once we have a local concurrent system running, the scaling to a distributed concurrent system would be just adding the node information. Given the ease that we have been able to write a concurrent version we will attempt to do a distributed version too.

2.5.1 Communications

In the real world we use messages to communicate. We also choose to ignore some and to give priority to others. This is the way Erlang processes communicates with each other, by using messages in the same fashion. As we will show later in the code we developed, the processes choose which messages they are interested and even give priority to specific messages.

To communicate with fellow processes, Erlang use asynchronous message passing. This is similar to Ada's rendevouz, but different as the sender do not wait for the receiver to receive, acknowledge nor return a value.

This is so. . . real world. It is very much like a snail mail letter thrown into a post box. . . sent and forget.

The receiver will wait only for messages in specific formats, much like the matching of the function parameters in section 2.4.1 and program 2.4.1, else it will ignore the message. This "wait till right message" is used later in the AsAvailable distributor (section 3.2.2) where the distributor will wait for messages from receivers that is available, before it will accept and handle a processing request message. Program 2.5.1 shows an example where we start in a waiting state with `loop/1`, and after all the available PIDs have been exhausted (lines 9-11) we go back to that state. In this waiting state we do not care about any `process` messages, as we can not process them without available processors in any case, so we only look and wait for `available` messages. For as long as we have available processors (either more than one in `[Head|Tail]=WholeList` or a single one in `Head` see section 2.5.2), we accept both the `available` and `process` messages on a first come first serve basis.

The author's opinion is that this is one of the best methods of inter process communications, as there are just about no real lock contentions, and dead lock situations can be easily elimated (as program 2.5.1 shows)

[5]An Erlang process is more a light weight thread as it runs inside the VM/Abatract machine

[6]I will exclude the more specialized `spawn_link` and `spawn/3, spawn/4` as they work mostly the same way, just having more tunables

[7]granted the code have to be residing on and available on the diffferent nodes

Program 2.5.1 Receiving messages and timeouts in Erlang

```
start_loop() -> loop(waiting).
loop(waiting) ->
  receive
    {available,PID} -> loop(available, [PID])
  after 5000 -> throw({timeout_error})
end.
loop(available, [Head|[]]) ->
        receive
                {available,PID} -> 
    loop(available, [PID|Head]);
                {process,{params}} ->
                                    Head ! 
    {Params},
                                    Loop(Waiting)
        after 5000 -> throw ({timeout_eror})
end;
loop(available, [Head|Tail]=WholeList) ->
  receive
    {available,PID} -> 
    loop(available, [PID|WholeList]);
    {process,{Params}} ->
          Head ! {Params},
          loop(available, Tail)
  after 5000 -> throw({timeout_error})
end.
```

the `after` clause that will handle the case when the process have waited too long and none of the right message(s) have arrived.

Guards in receiving messages

Although none of our code used the guard statements, it have to be noted that it is one of the nice features of Erlang as mentioned in paragraph 2.4.1. A quick example should suffice for our brief introduction for the reader to compare `loop/2` in program 2.5.2 using a guard (the `when` clause) versus the two seperate functions (differentiated using the mathing of `Tail` not an empty string) in program 2.5.1

2.5.2 Parameter List splitting

Program 2.5.1 shows another parameter feature that is quite frequently used in Erlang, that being of the splitting of the head (first element) and tail (all

Program 2.5.2 `loop/2` using guards

```
loop(available, [Head|Tail]=WholeList) ->
  receive
    {available,PID} -> 
     loop(available, [PID|WholeList]);
                {process,{params}} -> when Tail =:= []
                                    Head ! 
     {Params},
                                    Loop(Waiting)
    {process,{Params}} ->
            Head ! {Params},
            loop(available,Tail)
  after 5000 -> throw({timeout_error})
end.
```

BUT the first element) of a list. Also note on the one side the list is split, but the other side we have the while list.

This example could've been rewritten to take the guard

Chapter 3

Brzozowski's DFA construction

In ?, Brzozowski presented the notion of derivates of regular expressions, and showed how that would naturally leads to the construction of a state diagram from such derivation of a regular expression. Watson in ?, shows how the

3.1 Sequential algorithm

Program3.1.1, shows a Guarded Command Language version of Brzozowski's DFA construction algorithm. Although this copied from ?, with comments inserted to ease the discussions.

3.1.1 Reduced derivatives

When looking at this algorithm, the only dependency or shared state between iterations and derivatives, is the adding and removal of the derivatives to T. This is done in two places, the first is when a derivative is removed from the list/set when any q is taken from T and added/moved to D. The next place is when the newly derived $\frac{d}{di}q$ is checked for existance in $(D \cup T)$ and added to T if not. These two actions should either be atomic or inside critical areas if done through concurrent processes.

3.1.2 Path insertation

The path insertion $\delta(q, i) := d$ again is a independent operation that is effectively just a collection of the $RE, i, \frac{d}{di}RE$ tuples, indexed on the RE, i key.

Program 3.1.1 Brzozowski GCL **?**

```
func $Brz(E, \Sigma) \rightarrow$
    $\delta, S, F := \emptyset, \{E\}, \emptyset;$
    $D, T := \emptyset, S;$
    do $(T \neq \emptyset) \rightarrow$
        let $q$ be some state such that $q \in T$
        $D, T := D \cup q, T \backslash \{q\}$
        for $(i : \Sigma) \rightarrow$ #Inner loop
            $d := \frac{d}{di} q$ #Reduced-derivation
            #Already inserted this $\frac{d}{di}$?:
            if $d \notin (D \cup T) \rightarrow T := T \cup \{d\}$
            [] $d \in (D \cup T) \rightarrow$ skip
            fi
            $\delta(q, i) := d;$ #Path insert equivalent to $\delta(q, i) := \frac{d}{di} q$
        rof
        $\#Nullable\ tests:$
        if $\epsilon \in \mathcal{L}(q) \rightarrow F := F \cup \{q\}$
        [] $\epsilon \notin \mathcal{L}(q) \rightarrow$ skip
        fi;
    od;
    return $(D, \Sigma, \delta, S, F);$
```

3.1.3 Nullable tests

The nullable tests ($\epsilon \in \mathcal{L}(q)$) is also independent once we have the list of reduced-derivative REs (In the code it is the qs).

3.1.4 Sequential implementation

The initial code I had to work with, provided a sequential algorithm. This made use of Erlang's `lists:mapfoldl/3`. This is a function is similar to the `map/2` discussed in 2.4.2, but instead of returning a list, the addition that at the same time have an accumulator updated as the items are processed. Struass used the Σ as the list to process, and a function to do the $\frac{d}{di}RE$, and then adding that deritive into the D list being the accumulator. This a very efficient way of coding it in Erlang and a commendable method in the sequential case!

3.2 Concurrent algorithms

3.2.1 First attempt: ParMap

The first obvious parallelization method comes from doing concurrency over the alphabeth on the inner loop. This is also an easy method as the sequential algorithm makes use of `lists:mapfoldl/3`.

The sequential code use the provided `lists:mapfoldl/3` function. This provided a function to be mapped over the Σ alphabeth list. The function is constructed with the RE/q to be derived and it is given an acumulator parameter. In this implementation, the accumulator is the $\delta()$ storage. The output in Struass's implementation is then a list of reduced-derivatives which then is uniquely sorted with `lists:usort/1`[1] and already handled derivatives (those in D set) removed and then uniquely merged with the to-do list T.

The first parallelization attempt was to make use of a parallelized-map function as described in Armstrong**?** and then do the fold operation on the received messages. This will spawn a process[2] for each of the Σ_i and then to collect the various reduced-derivatives.

This method is an easy picking, but the granularity is spread over the alphabeth size. In other words with $l = size(\Sigma)$ there will be l processes processing the same RE, and then we will collected all of them (adding to δ, F, T as the messages arrive) and only after all of the l messages have been received, will another set of l processes be spawned. in short it will have bursts of requests, not a queue, which could cause thrashing.

It should further be obvious that a small l will have little concurrency, while a big l might be too much.

It has to be noted that this was the first consideration during the literature study on the Erlang concurrency model as **?** have a nice example doing a similar example. Looking back, This might be a faste implementatino with lower overhead than the next revisions.

3.2.2 Second revision

After considering the issues mentioned in 3.2.1, a second attempt is shown in figure 3.1. What needs to be pointed out, is that `mapfoldl/3` or rather any `lists:map/2` was not usable in the method chosen, but rather I used `lists:foreach/2` over the Σ alphabeth to generate and send the messages to the distributor.

[1]duplicates removed

[2]Remember erlang processes is not Unix processes, but rather threads inside the virtual machine

Figure 3.1 Flow for distribution of $\frac{d}{di}$

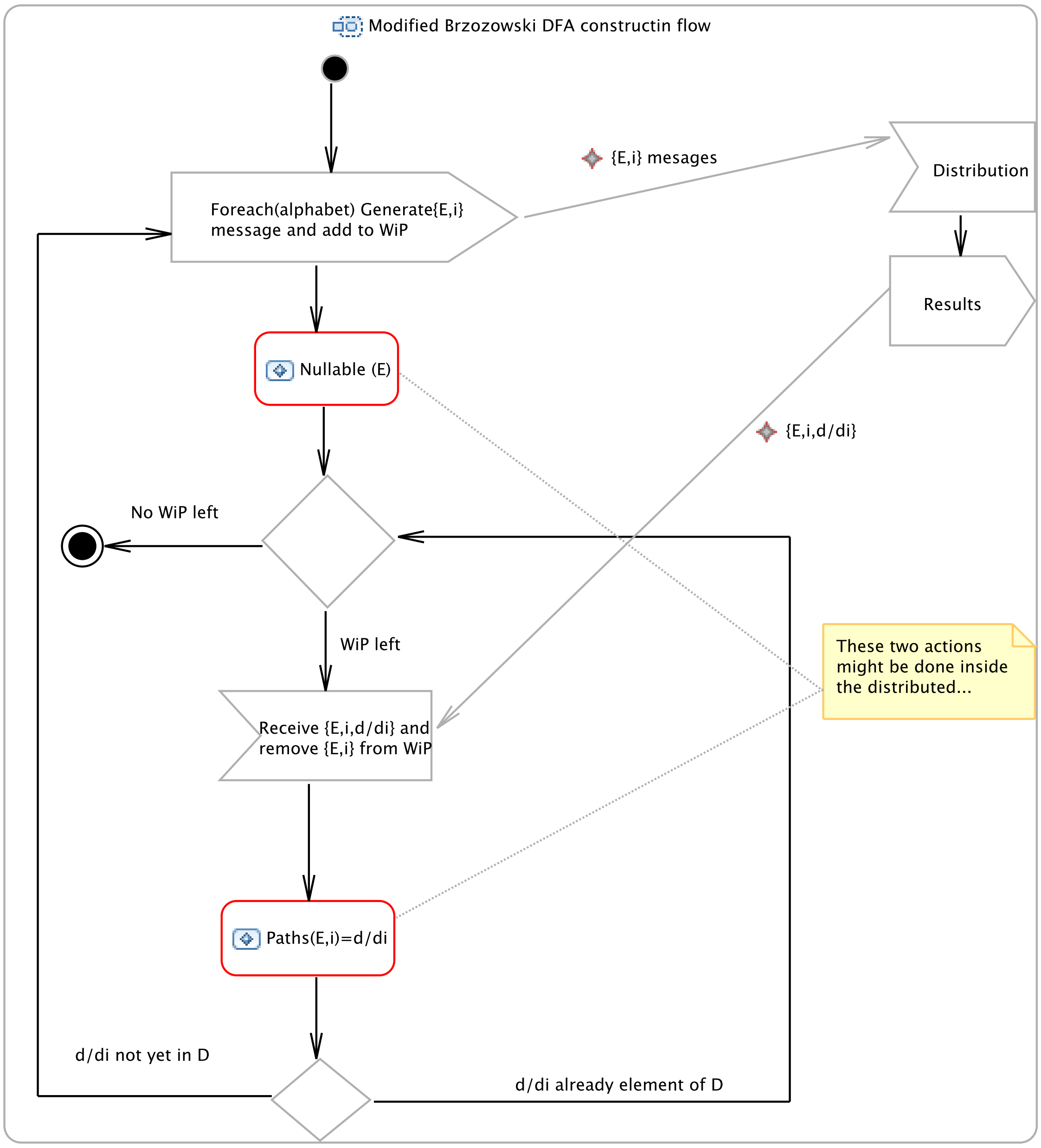

Modified Brzozowski DFA constructin flow
{E,i} mesages
Distribution
Foreach(alphabet) Generate{E,i} message and add to WiP
Results
Nullable (E)
{E,i,d/di}
No WiP left
WiP left
These two actions might be done inside the distributed...
Receive {E,i,d/di} and remove {E,i} from WiP
Paths(E,i)=d/di
d/di not yet in D
d/di already element of D

Diagram notation used

A few notes on the notation used in the diagrams (difficult to find an UML editor that can handle the LaTeX symbols):

RE the original regular expresion

E an expresion, could be the original RE, or part of the RE, in other words a derivative.

$\{E, i\}$ an expresion together with an $i \in \Sigma$

d/di the reduced derivative of E, ie. $\frac{d}{di}E$

WiP Work in Progress - Those messages not yet received.

Paths(E,i)=d/di the $\delta(E, i) = \frac{d}{di}E$

Nullable(d/di) ie the F list containing $\epsilon \in \mathcal{L}(q)$

Flow description

The sequential algorithm put the original RE on the Todo list T. Then it handles the RE as it would handle the derivatives found. In this algorithm we do something similar by not differentiating between the RE and the $\frac{d}{di}E$s in the first step.

"Inner Loop" The inner loop for the sequential algorithm is a creation of messages to be send for processing. These messages E, i consists of the E and the letter (i) of the alphabeth Σ to derive from.

We also put those messages send in a WiP (Work in Progress) list to keep track of those messages send and those received as we do not have any guarantees on the order of messages received given the inherent asynchronous nature of concurrency.

Note: Message parallelization It has to be noted that this algorithm is not concerned with the parallelization of those messages and will not consider it here, as it would be a function and optimization of the distributor. At this point the emphasis will be on the correctness of this algorithm as The distributor will be discussed seperately in 3.2.2.

Nullable($\frac{d}{di}$), Add E to D While writing this and considering the formal aspects to proof the correctness of this algorithm, the Nullable($\frac{d}{di}$) issue needs to be considered in more detail. In the sequential algorithm, this was done at the end of each outer loop. In this algorithm it is also outside and after the "inner loop" but before any of the derivatives are handled. In essence the nullable(E) is handled whenever we try to get more derivatives for an E.

E is also added to D, ie. $D := D \cup E$, to prevent any similar $\frac{d}{di}E$s to be skipped.

WiP test Check for an empty WiP list. If it is empty this process will terminate (perhaps also telling the distributor?). If there is still messages on the WiP list, continue to the receive section.

Receiving $E, i, \frac{d}{di}E$ and $\delta(E, i) := \frac{d}{di}E$ Once a message is received, the corresponding E, i is removed from the WiP list. The Paths is then updated by adding the received $\frac{d}{di}E$ using the $\delta(E, i) = \frac{d}{di}E$ expresion.

Checking $\frac{d}{di}E \in D$ The last step of this algorithm is to check whether the received $\frac{d}{di}E$ have already been looked at be checking the D list. If it has been considered before, the algorithm loop back to the receiving portion, else it loops to the messages generation portion.

no T todo list, but WiP Note that the is no Todo list (the T) as in the sequential case. This is because the algorithm immediately generates messages for those Todo and put them on the D list.

There is however a WiP list that serves the same termination condition as the T todo list in the sequential algorithm.

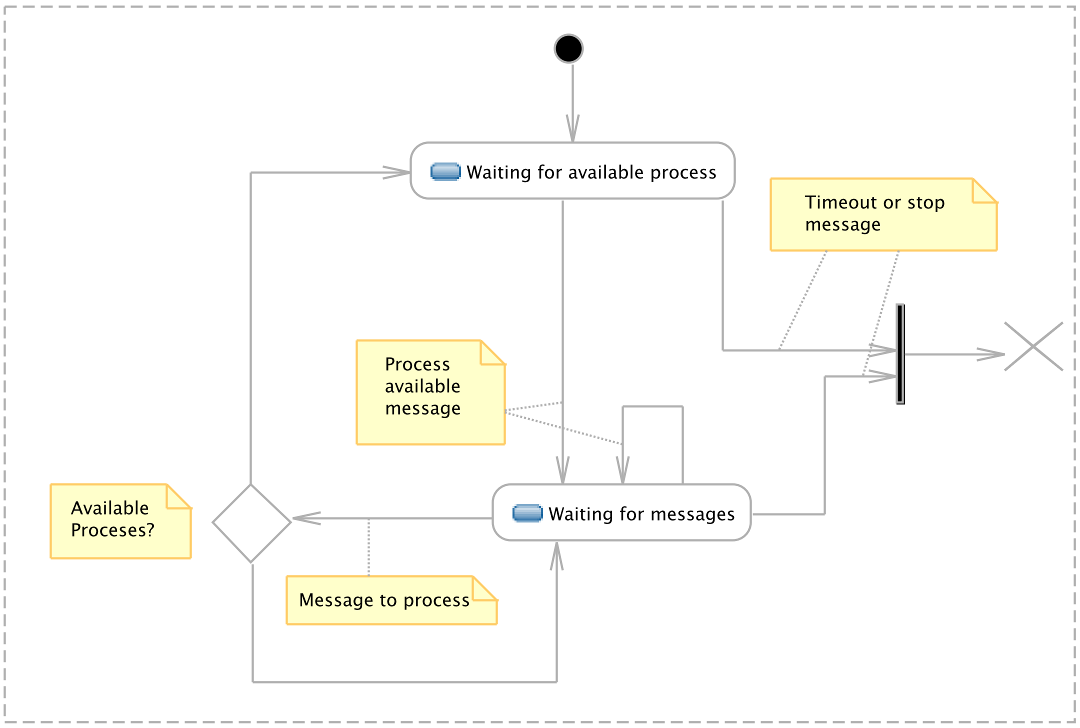
Distributors

Based on the stream of messages that the algorithm generates, there is various methods how these messages could be handled.

Sequential As a first test to confirm the correct algorithm in at least the sequential case (or the messages all getting processed in the same order as the sequential algorithm), each message received will be processed and the result sent back without any concurrency. This could also be implemented as a single process case of the Round Robin (3.2.2) and As-Available(3.2.2) distributors. This will be implemented as a single instance Round Robin case.

Round Robin The distributor will be given a list of processes that have been spawned and will handle requests. The first one on the list (head of list) will be sent the next available message. This process will then be added to the back of this list and the process repeated.

Figure 3.2 State engine for the AsAvailable distributor



As-Available Figure 3.2 shows the states for this distributor.

The distributor will start with no available processes in the WaitingForProcesses state and will wait to be sent the processes available for processing the requests. Once it receives an available process message, it will move to the ProcessesAvailable state.

When the distributor receives messages for processing, the distributor will remove the first process (again the head of the list) from the available list and sent it the message to be processed. The distributor then continue the loop with the tail of the list (minus the process that were sent a message). The distributor stays in this state while it still have processes available, but when it do not have any available processes, it will move to the WaitingForProcesses state.

When a process/thread finished it processing, it will inform the Distributor that it is again available for processing. The distributor will add this to the head of the list of available processes and repeat the ProcessesAvailable state loop.

3.2.3 Nullable also? (third attempt)

Having taken a relook at the algorithm, the `nullable()` part also seemed to be be distributable. After analysis of the nullable implementation from the original code base and having looked at Watson?PhD, it was concluded that there is no need to have it stuck inside the inner loop, as it is also an independent computation.

The new control and data flow is shown in figure 3.3 and we will explain this code module, line by line, in chapter ??. Note that the messages had to be augmented to allow for the differentiation of the `reduce(derive(E,i))` and the `nullable(E)` calculation requests. To be honest, it is not strictly needed in this system, as a simple match for {`E`} versus a match for {`E,I`} would have been sufficient. However, I would rather add this functionality, as it would help make the distributor-receiver pairs to be more easily extended and the same distributor-receiver pair be usable by different mappers.

I made a choice to have the receiver handled both the `reduce(derive())` and `nullable()` computations, as it would simplify the distributor, but having a seperate receiver(s) for each would not be that difficult to add for Erlang.

In this attempt we have moved all the computational intensive parts out of the core loop and delegated it to the receivers. The core loop now only aggregates the results and distribute any results that need to be distributed.

3.3 Map Reduce - the Google connection

After implementation of the second and third attempts, a rereading of ? brough me to Google's MapReduce and a nice figure that explains map reduce. Further researching Google's MapReduce, ? shows how to use MapReduce for counting words in a distributed manner. To do that, the pieces of the document(s) are distributed to mapper processes. The mappers just do the necesary string matching to find a word, and then send a stream of words with the count of "1" to the reducer. The reducer then take the keys (in this case the words) and aggregate the values(counts).

In 3.4 the third implementation is summarized in a MapReduce fashion. In other words, the Receivers is equivalent to the Mappers as they do the `Nullable()` and `Reduce(Derive())` and sent back a stream of answers (keys being the expresion and sigma or just the expresion) back to the Result receiver. The Result receiver (acting as the Reducer) is doing the aggregation either into the `Finish` list or the Delta dictionary.

Thus even though the initial idea was not based on MapReduce, a MapReduce based algorithm followed from a natural progression while dissecting and refining the algorithm presented.

Figure 3.3 Flow for distribution of $\frac{d}{di}$ and *null*

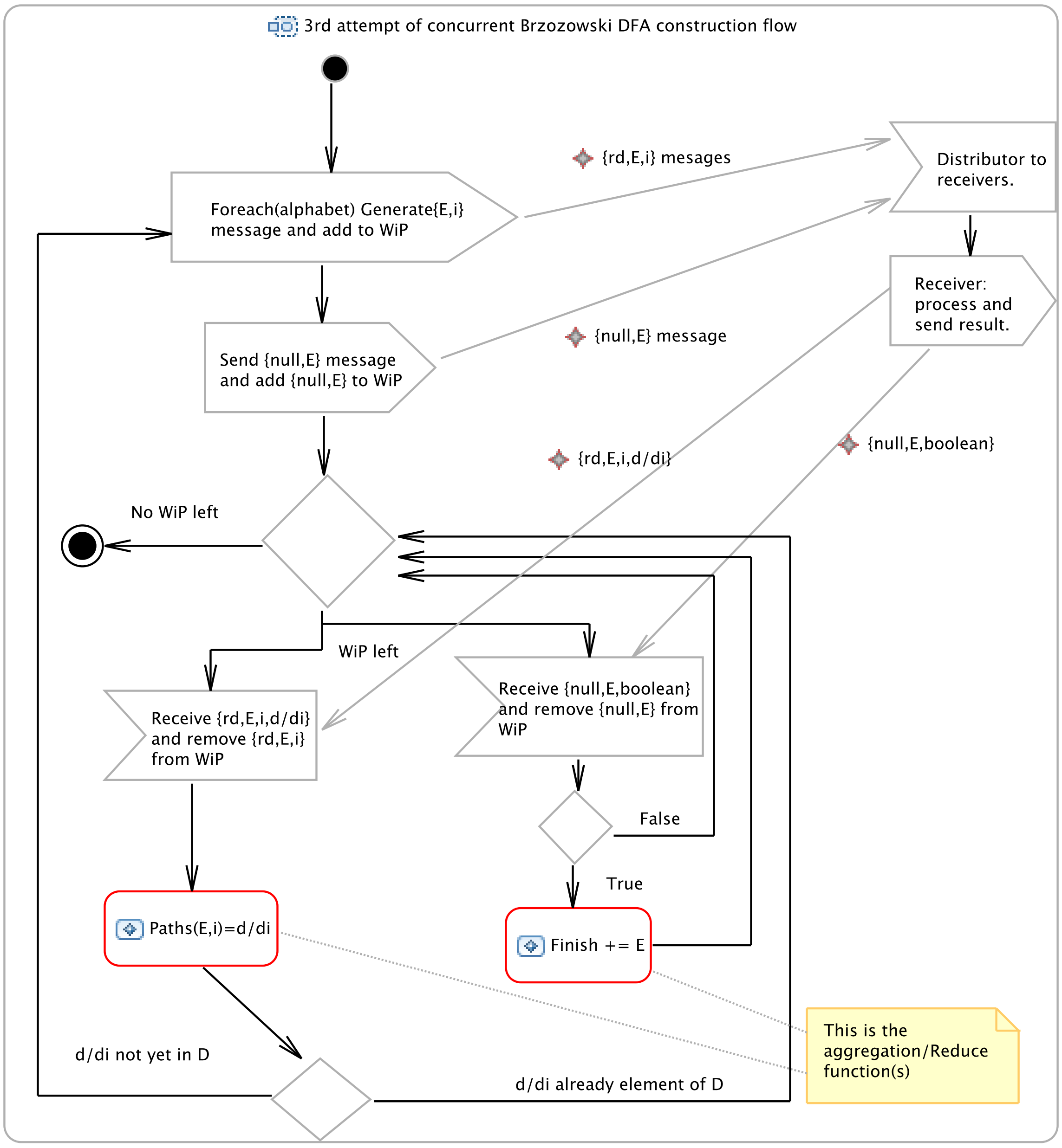
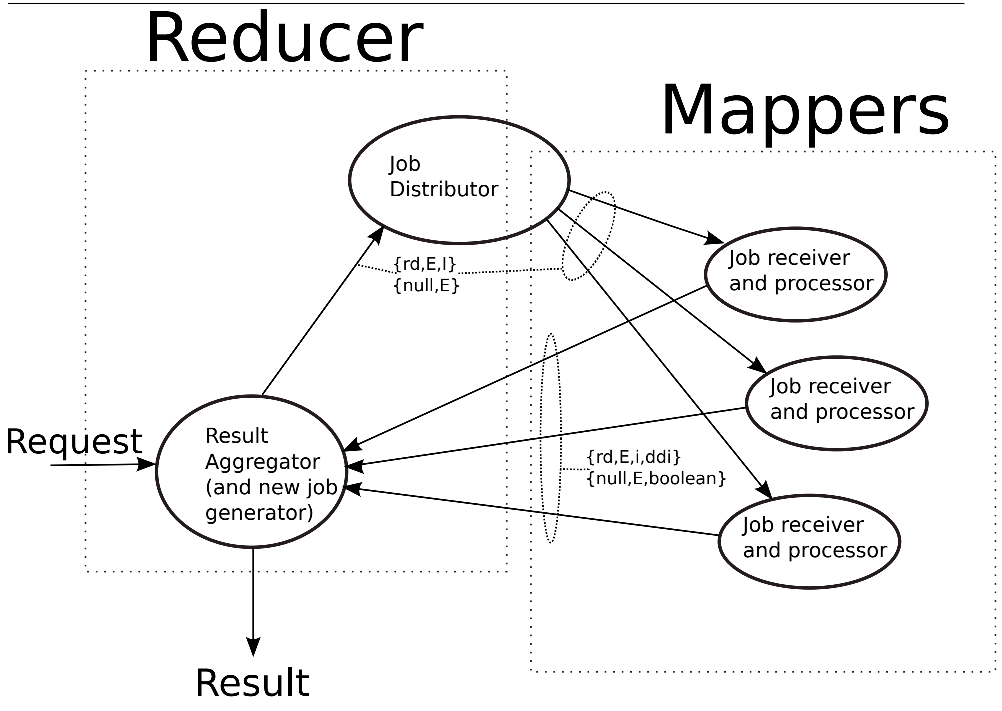

3rd attempt of concurrent Brzozowski DFA construction flow
{rd,E,i} mesages
Distributor to receivers.
Foreach(alphabet) Generate{E,i} message and add to WiP
Receiver: process and send result.
Send {null,E} message and add {null,E} to WiP
{null,E} message
{rd,E,i,d/di}
{null,E,boolean}
No WiP left
WiP left
Receive {null,E,boolean} and remove {null,E} from WiP
Receive {rd,E,i,d/di} and remove {rd,E,i} from WiP
False
True
Paths(E,i)=d/di
Finish += E
This is the aggregation/Reduce function(s)
d/di not yet in D
d/di already element of D

Figure 3.4 MapReduce overview of algorithm



Reducer
Mappers
Job Distributor
{rd,E,I}
{null,E}
Job receiver and processor
Job receiver and processor
Job receiver and processor
Request
Result Aggregator (and new job generator)
{rd,E,i,ddi}
{null,E,boolean}
Result

Chapter 4

Implementation

After spending about a weekend coding the second and third iterations, the author have been impressed by the expressiveness of the Erlang language to do such concurrency in this project. It has to be said that once you understand the pattern matching principles of Erlang, the coding do get easier and much more expressive that similar code the author have wrote in C/C++.

Chapter 5

Code Analysis

5.1 What will be looked at

In this chapter we will analyze the code to show the expressive nature of the Erlang language and the ease it was to be able to create the concurrent threading of the Brozozswki DFA generation and to make changes for different concurrency algorithms.

The reader would be justified to keep figures 3.2 and **??** next to the code to be able to fully understand the code explanations. Even though we will put in an effort follow the flow of the code, the concurrent nature of the code (as the different paths in those figures would attest to) would not make that entirely possible.

The most "important" reason the author would apologize for not following the logical flow of the code to ease the understanding, is the way how the functional language ordering of the code happens. Thus even though the logical ordering/code flow is not that obvious, the authors would argue that the code design and Erlang features would be appreciated by understanding the reasons for the ordering as done in this paper.

Chosen code to explain Given the bit more complexity of the nullable code of **??**, and it being the last written code, that will be the code set used as basis for explanation in this chapter. Certain function names (like `hv_brzp_null`) should be replaced for the previous function names used.

Actual Brzozoswki code As we have used a previously working Brozoswki sequential implementation, and just replaced the core innerloop with our concurrent methods, we will not discuss those code in this document as it was outside this project's scope and function. It will have to be stated that on analysis of those code, the author found it to be closely follow Watson**?** and that would be a good place to research the specific workings of the Brozoswki DFA generator and the code related to that.

Generated DFA correctness proofs The author would like to mention that it was not the scope of this work to investigate the correctnes of the code per se. The author have, however, checked and confirmed that the DFAs generated and used in this work, have been verified against the original implementation, thus errors from DFA generation, would be traced back to the original sequential implementation used.

5.2 Ab Initio

[1]

We will discuss the code by starting at the entry function. As this function starts the distributor that initiates the receiver processes, we will continue to the receiver processes. After we've explained the distributor/receiver workings, we will jump back to the actual core that does the decisions and work creation.

5.2.1 Entry Code

Even though it could have been parameterized more generally, the author chose to have one function for each of the attempts/iterations of parallelism embedded in a seperate file or module for each of the attempts.

However, the RoundRobin and AsAvailable schedulers have been extracted as parameters. First lets look at the entry call for the AsAvailable scheduler:

```
%The entry to hvp2
% third Parameter to chose the roundrobin or available scheduler
hv_brz(RE, Sigma, available, N) ->
    TimeOut = 3000,
    Res = self(),
    WiP = [], %Only the first (and last) one would be "empty"
    Finish = [],
    Dlist = [RE], %Start state.
    Delta = dict:new(),
    Dist = spawn(fun () -> hv_dist_avail_start(TimeOut, Res, N)    end),
    hv_brzp_null(RE, Sigma, Dist, WiP, Finish, Dlist, Delta);
```

[1]Ab Initio: Latin for from the beginning. Introduced to this term from the name of a similarly named product/company used in massive extract-transform-load (ETL) environments like telcos and financials

Lines 4-9 is the parameters we used for timeout values and the reference to ourself as the results receiver. We initiated the work in progress (`WiP`) and expressions already processed (`Finish`) as empty lists. `Dlist` is set to the list containing the single element of the regular expresion to be considered. These values are initialzed the same in all the entry functions.

Line 10 spawns the distributor process[2], and using the fourth parameter (bound to the variable `N`) as the number of worker threads to spawn. Also the timeout and our own process information(`Res`) is passed as parameters.

We then call the actual core of the processing function (`hv_brzp_null` in this case) with the parameters declared and initialized in lines **??**-10. Notice that the `Dist`ributor from line 10 is passed as a parameter to this core processing function, and that this line is also verbatim copy of line 15 of the roundrobin scheduler!

Doing the RoundRobin scheduler, we do the same as the AsAvailable scheduler, except we spawn a roundrobin distributor. However, note that line 12 is the function called with the third parameter matched as the atom `roundrobin` compared to line **??** where the third parameter matched the atom `available`.

```
12 hv_brz(RE, Sigma, roundrobin, N) ->
13 Repeat lines ??-??
14     Dist = spawn(fun () -> hv_dist_rr_start(TimeOut, Res, N) end),
15     hv_brzp_null(RE, Sigma, Dist, WiP, Finish, Dlist, Delta).
```

For special testing, we used a slightly modified distributor starting function, but again we just used a function that used parameter matching to pick the right initialization function.

```
16 %Specialized one with only two named receivers using the RoundRobin method
17 hv_brz(RE,Sigma,rr_spec) ->
18  Repeat lines ??-??
19  Dist=spawn(fun() -> hv_dist_rr_spec_start(TimeOut,Res) end),
20  hv_brzp_null(RE,Sigma,Dist,WiP,Finish,Dlist,Delta).
```

Repeated lines

In retrospect, we could have extracted those repeated lines and used an intermediary function to set and pass those on, but it was not essential to the project goals, and the gains would have been minimal and as such not considered, though for production code this needs to be done.

[2]Remember that Erlang processes are comparable to C/Java threads, but from the program's perspective they are processes

5.3 Processing (Receiver)

The function `hvp2:hv_brzp_null7` is the core receiver we will discuss in this section. The reader should notice the use of the function call pattern matching to follow the major paths in the receiver code, and that the receiver calls itself recursively until there is no more work in progress (WiP), ie. tail recursion at play as explained in ??.

The difference between the empty work in progress (WiP) 5.3.1 and the receive only in 5.3.2, is the empty (`[]`) fourth parameter... and it must be in that order, else the empty WiP state would never be matched.

The derivation of a new expression is done in 5.3.3. Here (again note the order for matching the fucntion parameters) the first parameter is the expression to derive, and did NOT match the atom `receive_only`!

5.3.1 WiP empty (nothing to do anymore

```
%The case when the WiP is empty
hv_brzp_null(receive_only ,Sigma,Dist , [] ,Finish ,Dlist ,Delta) 
    ->
  %io:format("WiP finished"),
  Dist!{stop},
  #dfa{ states=lists:sort(Dlist), symbols=Sigma, 
     start=lists:last(Dlist) , transition=Delta, 
     finals=Finish };
```

As we have already mentioned, the fourth parameter is an empty list, and given the fact that we are only receiving (the `receive_only` atom matched in the first parameter) we conclude that we are finished processing as there is no more outstanding derivations, and we can stop the processes, thus we send the `Distributor` a stop message.

Finally we will return the data in a format depicted by the `#dfa` record format. We will refer the author to the specifics of this record format in ?. Suficient to say that it is nothing different from a norma tuple, just a method to name and order the tuple into a record structure named elsewhere in the pre-compiled code, typically a header file.

5.3.2 Receive only

```
%Receive only, nothing to derive
hv_brzp_null(receive_only, Sigma, Dist, WiP, Finish, 
     Dlist, Delta) ->
  receive
    {rd,E,I,DDI} -> 
                %
     io:format("brzp_null_2:"),io:write({rd,E,I,DDI}),io:format("~n"),
31     NewDelta=dict:store({E,I},DDI,Delta),
32     case lists:member(DDI,Dlist) of
33       true ->hv_brzp_null(receive_only,Sigma,Dist, 
       lists:delete({rd,E,I},WiP),Finish,Dlist,NewDelta);
34       false -> hv_brzp_null(DDI,Sigma,Dist, 
       lists:delete({rd,E,I},WiP),Finish,[DDI|Dlist],NewDelta)
35     end;
36     {null,E,true} -> 
37                 % io:format("brzp_null_2: ~p 
       true~n",[E]), 
38       hv_brzp_null(receive_only,Sigma,Dist,
39                           
       lists:delete({null,E},WiP),[E|Finish],Dlist,Delta);  
       % Add nullable states to F
40     {null,E,false} ->
41       
       hv_brzp_null(receive_only,Sigma,Dist,lists:delete({null,E},WiP),Finish
42     after 5000 ->
43                     io:write(WiP),
44                     output_mailbox(1),
45                     throw(timeoutRec_only)
46   end;
```

This part of the receiver, is executed based on `receive_only` matching the first parameter, and it only waits for messages from the derivation processes. This should either be a result of a derivation tagged with the atom `rd`, or a result from a nullable test, tagged with `null`. It would remove this request from the WiP list, and recurse back to the same function name, but with slightly different parameters, modified based on the result received.

But let us have a look at the options steps a bit more closely:

29 we have received a derivation result based on the `rd` atom that matched.

31 Modify the Delta dictionary by creating a new dictionary with the Expression and alphabet letter (`E,I`) tuple as a key, and the derived expression `DDI` as the value. **Note:** the Delta variable is not changed at all, but a **new** variable with the name NewDelta is created and bounded to the new result.

32 Some expression and alphabet inputs might provide the same derivations, we check if we have seen a similar one before, in which case we do not waste time on deriving for the same path multiple times. This is done by checking for the existence of the DDI just received in the list Dlist, of previous expressions that aready have been issued derivation requests.

?? We have seen this derivation befote, so we just remove the E,I from the WiP list, and repeat the process by recursing back.

34 here we have not seen this derived expression before, thus we will call ourselves again, but this time with the expression that need further derivation as the first parameter. E,I is also remove from the WiP list.

Line 42 we have assumed that after 5seconds of inactivity, there is a problem in the distributor or processor, at which point we exit, dumping the state of the mailbox and throwing an exception. This was a very usefull method to track a typing mistake where we have sent all the data, but with a mistyped atom!

Debugging statements Even though we all want to present ourselves as invincible and perfect coders, we all make mistakes, and thus we need to debug or fault find our code. Line 30 is a typical line used to debug the code. This line outputs a user format string with `io:format/2` similar to the Unix `printf(3)`[3]. This is then followed by the Erlang formating of the data structure passed to `io:write/1`, and finally a linefeed. Line 37 shows an example with some substitution where the `p` is substitute with the value bound to the variable `E` and note that it is a list `[E]` passed on to the function, and the substitution happens based on the list positions.

5.3.3 Expression to derive

This part of the function, have the same code than the receive_only in lines57-70. It is lines 50-54 that is the interesting part in this function part.

```
47 % When we have an RE/E d/di that needs to be derived/etc.
48 hv_brzp_null(E,Sigma,Dist,WiP,Finish,Dlist,Delta) ->
49   %foreach(Sigma) send message to Dist
50   lists:foreach(fun(X) -> Dist!{process,[rd,E,X]} end,Sigma),
51   %F1=nullable(E,Finish),nullable(RE),
52   Dist!{process,[null,E]},
53   %foreach(Sigma) insert {E,I} into WiP, and add the null to the begining ;)
54   NewWiP=[{null,E}|add_wip(WiP,rd,E,Sigma)],
55
```

[3]the 3 inside the brackets refers to the section 3 manual pages, which is the "standard" libc functions on Unix

```
  %WiP would not be empty in this function :)
  receive 
    {rd,E,I,DDI} -> %io:format("brzp_null_why:  
      "),io:write({rd,E,I,DDI}),io:format("~n"),%"~p ~p 
      ~p~n",[E,I,DDI]),
      NewDelta=dict:store({E,I},DDI,Delta),
      case lists:member(DDI,Dlist)  of
        true -> 
      hv_brzp_null(receive_only,Sigma,Dist,lists:delete({rd,E,I},NewWiP),Fin
        false -> 
      hv_brzp_null(DDI,Sigma,Dist,lists:delete({rd,E,I},NewWiP),Finish,[DDI|
      end;
    {null,E,true} -> % io:format("brzp_null: ~p 
      true~n",[E]),
            
      hv_brzp_null(receive_only,Sigma,Dist,lists:delete({null,E},NewWiP),[E|
      % Add nullable states to Finish
    {null,E,false} -> %io:format("brzp_null: ~p 
      false~n",[E]),
      
      hv_brzp_null(receive_only,Sigma,Dist,lists:delete({null,E},NewWiP),Fin
    %Other -> throw (Other)
    after 5000 -> io:write(WiP),throw(timeOut)
end.
```

50 This is an example where we apply a function over the Sigma alphabet. The "function" in this case, is to sent a message to the distributor `Dist`. This message consists of the tuple `process,[rd,E,X]`. For some reason I made it a two pair tuple, with the parameters a list, rather than an extended tuple.

51 The non-distributing versions, used theses lines to check for the end conditions and whether the expression is

5.4 Receivers

5.4.1 Round Robin

```
%The receiver for the round robin/sequential tests
%
hv_rr_rec(Name,Res) ->
  %io:format(" starting: ~p self:~p 
      Res:~p~n",[Name,pid_to_list(self()),pid_to_list(Res)]),
  receive
```

```
    {stop} -> %io:format("stopping ~p ~n", [Name]),
            false;
    {process, [rd,E,I]} ->
        Res!{rd,E,I,mds:reduce(mds:deriv(E,I))},
             hv_rr_rec(Name,Res);
    {process, [null,E]} -> Res!{null,E,mds:null(E)},
                      hv_rr_rec(Name,Res);
    Other -> io:write(Other),throw(Other)
    after 3000 -> io:format("Timeout ~p
        quiting",[Name]),io:nl()
  end.
```

5.4.2 AsAvailable Receiver

```

%==========================================================
%The distributor that  have receivers that tells it
    when
% they are finished and ready for new processing
%==========================================================

%Let the Distributor know when finished with
    processing
% But this state engine not rock solid when more than
    one messages was sent
% etc.
hv_rec_available(Timeout,Name,Res,Dist) ->
  %io:format("~nEntering ~p ~p ~p ~p
    ~n",[Name, integer_to_list(Timeout), pid_to_list(Res), pid_to_list(Dist)])
  %First we handle all stop/process messages on the
    queue
  %If the distributor works correct, this shouldn't
    be necesssary,
  %But we could "modify" the distributor to send
    expected "short"
  % requests in rapid fire??
  receive
    {stop} ->% io:format("stopping ~p~n",[Name]),
            exit(0); %Need to do the exit here else
        continue to next
    {process, [rd,E1,I1]}-> %io:format("~p ~p
        ~p~n",[Name,E1,I1]),
      Res!{rd,E1,I1,mds:reduce(mds:deriv(E1,I1))},
      hv_rec_available(Timeout,Name,Res,Dist);
    {process, [null,E1]} -> %io:format("Null: ~p ~p ~n",[Name,E1]),
      Res!{null,E1,mds:null(E1)},
      hv_rec_available(Timeout,Name,Res,Dist);
    Other1 -> throw(Other1)
    after 0 -> Dist!{available, self()} %Nothing in queue, so we let the Distributor know
  end,
  receive %There were no "normal"/"expected" messages in the queue, so lets wait :)
    {stop} -> %io:format("stopping ~p~n",[Name]),
              true;
    {process, [rd,E,I]} -> %io:format("~p ~p ~p~n",[Name,E,I]),
      Res!{rd,E,I,mds:reduce(mds:deriv(E,I))},
      hv_rec_available(Timeout,Name,Res,Dist);
    {process, [null,E]} -> %io:format("Null: ~p ~p ~n",[Name,E]),
      Res!{null,E,mds:null(E)},
      hv_rec_available(Timeout,Name,Res,Dist);
    Other -> throw(Other)
    after Timeout ->
      io:format("Timeout ~p quiting ~n",[Name])
  end.

%Start N round-robin receivers that will send their results to Res
% returning the list of  PIDs.
list_start_servers(0,_Res) -> [];
list_start_servers(N,Res) ->
  [spawn(fun()->hv_rr_rec("Receiver "++[N+\$0],Res)
            end) | list_start_servers(N-1,Res)].

%Number of servers variable, should make that a number to pass too, but
% for the moment this is adequate to test etc.
hv_dist_rr_start(TimeOut,Res,N) ->
  hv_dist_rr(list_start_servers(N,Res),TimeOut).

%Two specified servers
hv_dist_rr_spec_start(TimeOut,Res)->
  Rec1=spawn(fun() ->hv_rr_rec("Rec1",Res) end),receive after 100 -> true end,
  Rec2=spawn(fun() -> hv_rr_rec("Rec2",Res) end), receive after 100 -> true end,
  hv_dist_rr([Rec1,Rec2],TimeOut).
```

5.4.3 Round Robin distributor

```
%Round Robin distributor.. we know this is not "optimal" :)
hv_dist_rr([H|T]=Receivers,TimeOut) ->
  %io:format("Dist_rr starting: SendTo: ~p Self:~p ~n",[pid_to_list(H),pid_to_list(self())]),
  receive
    {stop} -> lists:foreach(fun(X)->X!{stop} end,Receivers);
    {process,Param} -> H!{process,Param},hv_dist_rr(lists:append(T, [H]),TimeOut);
    Other -> io:write(Other),throw(Other)
    after TimeOut ->
      io:format( "Dist quiting and stopping receivers"),
      lists:foreach(fun(X)->X!{stop} end, Receivers)
  end.
```

5.5 AsAvailable distributor

```
%The Available distributor
%First the "empty" case
hv_dist_available(Timeout,[]) ->
  %io:format("Entering dist_available [] ~n"),
  receive
    {available,PID}->
      receive %We check for the availability of a process message to "fast track", else call the normal wait
        {process,Param}->PID!{process,Param},hv_dist_available(Timeout,[]) %Goody! a message available
        after 0-> hv_dist_available(Timeout,[PID]) %Normal wait since no process message in mailbox
      end;
    {stop} -> io:format("Distributor stopping from empty state~n") %no available receivers to stop :(
    after Timeout ->
```

```
      io:format("timeout distributor from waiting state~n")
  end;
hv_dist_available(Timeout, [H|Tail])-> %At least have a PID to check
  %io:format("Entering dist_available with~n"),
  receive
    {available,PID}->hv_dist_available(Timeout, [H,PID|Tail]); %H or PID, shouldn't matter which is first'
    {process,Param}->H!{process,Param},
              hv_dist_available(Timeout,Tail);
    {stop}-> lists:foreach(fun(X)->X!{stop} end, [H|Tail]); %Stop all the available receivers
    Other -> throw(Other)
    after Timeout ->
      io:format("Timeout distributor fron available state~n")
  end.

%Start the receivers and the distributor
hv_dist_avail_start(Timeout,_Res,0) ->
    io:format("Empty~n"),
    hv_dist_available(Timeout,[]);
hv_dist_avail_start(Timeout,Res,N) when N>0 ->
  %io:format("~n~p: Dist avail ~p ~p ~p~n~n",
  %      [pid_to_list(self()),erlang:integer_to_list(Timeout),
  %       erlang:pid_to_list(Res),erlang:integer_to_list(N)]),
  Dist=self(),
  spawn(fun() -> hv_rec_available(Timeout,"Receiver "++erlang:integer_to_list(N),Res,Dist) end),
  hv_dist_avail_start(Timeout,Res,N-1).
```

5.6 Anxillary functions

5.6.1 Adding to Work in Progress

The Work in Progres (WiP) is a list of the type of processing that have been sent out to the distributors, and that we still have not received any responses back for. It is a (in our implementation) a simple list, but is perhaps a place to optimize if it grows too big. This is especially as we prepend the new work to the beginning of the list, but the expected responses to be received next are those at the back of the list that would need to be removed, so in retrospect a FIFO type queue would be a prefered implementation when optimizing.

```
191 %For all the Sigma add {E, i} to the Work In Progress
192 add_wip(WiP, Type, E, [H]) -> [{Type, E, H} | WiP];
193 add_wip(WiP, Type, E, [H| SigmaT]) -> add_wip([{Type, 
        E, H} | WiP], Type, E, SigmaT).
```

Again we saw the use of specialization by the use of parameter matching. Here we first check on line **??** if the fourth parameter is a list with a single element (and we bound the variable `H` to that single element), else we match on line 193 for a list with more than one element and we bind the head (first element) to `H` while the tail (rest of the list excluding the head) is bound to `SigmaT`.

Mailbox debugging

```
194  output_mailbox(N) ->
195   receive
196     Mess -> io:format("Message ~p~n:", [N]),
197             io:write(Mess), io:nl(),
198             output_mailbox(N+1)
199   after 0 -> exit(123)
200   end.
```

5.7 Optimization choices

In this section we will explain some of the code and give critique how this could be made more resilient and robust.

5.8 Coding enhancements

5.8.1 Distributors

The main decison here was that the distributors will not “really” control the receivers (other than to tell those available when a stop message have been received). There are several ways to remedy this especially using the `spawn_link` that would tell the distributor (the PID that spawned the receivers) which have terminated. This way the distributor could make a decision whether to respawn the process or not.

At this stage we have just proved the distribution in a concurrent fashion as the project’s goal and would leave these enhancements to implementors of production code.

5.8.2 Work in Progress

We do not check at all whether there are work in progress (WiP) that have not returned to us, ie. a node/process failed while working on an expresion. This also need to be considered and rescheduled in production code, especially when using distributed code. Here the `spawn_link` as discussed in 5.8.1 would again be used to inform the work producer (reducer in map-reduce terms) that there were a failure and that it might need to resubmit WiP for recomputation.

Several strategies could be used here, the simplest being that the mapper would only resubmit WiP if notified of a failure and it timed out while waiting for results, meaning that those left in WiP might have been those that have failed. A bit more complex strategy would have the distributor know which job was send to which receiver (mapper) and that it could restart or resubmit that job once it received the failure notice. A control freak case could be that the distributor would also inform the reducer about which mapper received which job, and once the mapper dies, let the reducer know which mapper died so that the reducer can resubmit the job. This last method would also help the reducer to get some performance or processing information from each job.

Chapter 6

Correctness proving????

`Iets wil my sê dat ons dalk net iets hieroor moet noem...`

Chapter 7

Performance

7.1 Speed comparisons

??

The development and tests were all done on Apple Mac laptops, both having dual core Intel processors, and the results of the tests were discouraging, however it were not surprising. Two things in the tests stood out as needing investigation: first the size of the tests never took the CPU utilization above 115%, and the second it that the processing time versus the message sizes, is too little to make a difference. But lets look how bad the results were.

In table 7.1 we see the regular expression (in the syntax used in the code) that we used to test the performance of the algorithms developed in this project.

We conducted 20 test runs of each algorithm using 2 and 10 threads, and then averaged the results. In table 7.2 we tested the "full" ASCI byte range against the regular expression, and in 7.1 we only test against the space and the letters a to z and A to Z.

Table 7.1: Expression used for testing

```
{concat,
{union,
"Is dit nog hierso",
{kclosure,"Here"}},
{kclosure,{union,
"Testing",
"My testing"}}}
```

Table 7.2: $\Sigma \in [1 \ldots 255]$

Sequential	145947	
Threads:	2	10
Round Robin	1168392	1111456
RR nullable	1201972	1147706
AsAvailable	1231590	1253817
AA Nullable	1308366	1300956

Table 7.3: Using space, a-z and A-Z

Sequential	28886	
Threads:	2	10
Round Robin	84879	76499
RR nullable	88057	78519
AsAvailable	89504	85985
AA Nullable	98305	94441

7.1.1 Discussion of the results

As were mentioned early in **??** we noticed the CPU utilization never increased above 115%, which was quite discouraging, but given that the Erlang VMs are optimized on Linux and Solaris we were not that surprised, but as time and available systems were not available to test or confirm this hypothesis, we can not make any further remarks on the MacOSX Erlang VM as such.

However, there is another story to be told given the results in tables 7.2 and **??** and what **?** a;so refer to, and is the issue the overheads versus the work done. If the round robin and as-available algorithms are compared, it is obvious that the as-available algorithm have more overhead per message than the round robin (and given the code size differences it is expected). Even just moving the nullable tests to the threads, showed a decrease in performance.

The other interesting results for the two tables, are the overhead of the unused characters in the alphabet in the regulr expression, made the performance penalty hit go from a factor of approximate 3 in table 7.1 to a factor of over 8 in table 7.2. This tells us that the processing done per work-unit is not enough to warrant the overhead of the fine grained concurrency of our algorithms.

Chapter 8

Conclusion

In this project we investigated the concurrency features of Erlang, and applied that to the Brozoswki DFA construction. Erlang's concurrency features are quite expressive (and impresed the author), and the coding for the concurrency were done much quicker than initially anticipated. The authors would acknowledge that the claims of ease of concurrency of the Erlang designers are achievable with minimal effort.

The Brozoswki DFA construction algorithm and the methods chosen to do concurrent processing to derive the DFA, was not able to achieve any speedup on the hardware tested. It will be the authors' opinion that the speedups wil not be easily achieved as the processing needs are much less than the message sizes, and the overhead is more than the actual processing required.

8.1 Future studies/work

As our research focussed on threading the processing over the derivation of each sub-derived expresion for each of the alphabet entries, we concluded that it is too fine grained, and research could be looked at to rather spread the concurrency over each derivation with its alphabet as a processing unit.

Appendix A

Listings

```
lists:sum(
 lists:map(
   fun(X) ->
     element(1,
       timer:tc(hvp1,hv\_brz,
        [{concat,
          {union,
           "Is dit nog hierso",
           {kclosure,"Here"}},
          {kclosure,{union,"Testing","My testing"}}}
         ," "++lists:seq($a,$z)++lists:seq($A,$Z)
         ,available,2]))
     end
   ,lists:seq(1,20)
))/20.
```