# BISIMULATION REDUCTION WITH MAPREDUCE

Master's Thesis
of
Jeroen Vonk

UNIVERSITY OF TWENTE.

# Bisimulation reduction with MapReduce

Master's Thesis

of

Jeroen Vonk

born on the 10th of June 1988 in Uitgeest

29 August, 2016

Graduation Committee:
Prof. Dr. J.C. van de Pol
Dr. S.C.C. Blom
Dr.ir. Djoerd Hiemstra

University of Twente
Department of Computer Science

# Contents

# Figures, Code and Tables

# Chapter 1

# Introduction

Today we have the joy of many technological assets. Maybe you are reading this on a laptop, and downloaded this Thesis over the internet. But all these technologies have a drawback; they make life very complex, when you exactly want to know what is happening. Even an ordinary item such as a pencil is very complex due to our modern production processes. It can even be claimed that not a single person on earth has the knowledge and capability to produce a pencil identical to a store-bought pencil.[31] Knowing how the production process of a single pencil works might not be a common thing to question. But what about knowing how the airbags in your car work? How can a consumer, or producer as a matter of fact, know with a 100% certainty that airbags will deploy in case of a crash or will not deploy in case of a rare racing condition in the car computer?

Ironically enough, technology can supply a solution. Within the field of Computer Science a lot of previous and current research is done on model checking[26]. Model checking allows researchers to simulate a process or system, and exhaustively test for wanted or non-wanted properties. Logically, the result of these test are as dependable as your model represents the actual system. The best model then, would be a model representing the system down to its last atom, allowing for every possible interaction with the model. The model of course will become extremely large, a situation known as *state space explosion*.
Current research[20, 35, 17] therefore focuses on:

- Storing larger models
- Processing large models faster and smarter
- Reducing the size of models, whilst keeping the same properties

In this thesis we will focus on reducing the size of the models using bisimulation reduction[25, 29]. Bisimulation reduction allows to identify similar states that can be merged whilst preserving certain properties of the model. These similar, or redundant states will be identified by comparing them with other states in the model using a *bisimulation relation*. The bisimulation relation will identify states showing the same behavior, that therefore can be merged. This process is called bisimulation reduction [25]. A common method to determine the smallest model is using partition refinement[7]. We will elaborate on these techniques is Chapter 2.

In order to use the algorithm on large models it needs to be *scalable*. Therefore we will be using a framework for distributed processing that is part of Hadoop[34], called MapReduce[16]. Using this framework provides us with a robust system that automatically recovers from e.g. hardware faults. The use of MapReduce also makes our algorithm scalable, and easily executed at third party clusters. However, this framework will put several limitations on the design of the algorithm, and the available data structures. It is reported that Hadoop can be slow when using multiple iterations[4, 23].

The main contributions of this project will be bisimulation reduction with MapReduce. This gives us the following research question:

> ***Research Question:*** **To what extent is MapReduce a suitable platform for bisimulation reduction?**

In order to judge the suitability of our solution we will run several tests reducing predetermined models. During these tests we will look at two things that can distinguish our solution opposed to current solutions. Leading to the following subquestions:

> ***Sub-Question 1:*** **How much space does a MapReduce based bisimulation reduction require w.r.t. scaling of the model size?**

> ***Sub-Question 2:*** **How much time does a MapReduce based bisimulation reduction require w.r.t. scaling of the model size?**

This report is structured as followed: Chapter 2 explains some preliminaries like: What is a model exactly? How can you define bisimulation reduction, and what does Hadoop do? Chapter 3&4 contain two different bisimulation relations with their respectable implementations, more information on their implementation is given in 5. In Chapter 6 we put the implementations to the test, and explain something about the test procedure and the used hardware. After the experiments we will provide the Conclusions and Future work in the final Chapter 7.

## 1.1 Related work

Current state of the art tools already have various sequential implementations for bisimulation reduction[26, 13]. We will now list a brief overview of this work. The used terms and techniques will be explained in more depth in Chapter 2. For sequential implementations the work by Groote and Vaandrager[19] has been used which has a complexity of $O(mn)$, where $n$ is the number of states and $m$ the number of transitions. Recently an even more efficient algorithm has been devised with a complexity of $O(m(log|Act| + logn))$[20]. The existing tools make use of *partition refinement*[12, 25]. Partition refinement splits models in equivalent blocks or partitions, with goal to find the *coarsest* partitioning of a model whilst keeping certain properties of the model. To allow distributed partition refinement Blom and Orzan introduced the use of signatures[8]. Signatures allow for quick identification of partitions for states with less centralized information. Since then several tools have been developed that allow efficient distributed bisimulation reduction, for example using mpi [26]. With the current shift to large clusters and data analysis in the cloud on services like Amazon[30] it is possible to use large clusters without a big investment. Current research includes using MapReduce for state space generation [6], (limited) state space reduction[27], and model checking[5, 22, 1].

# Chapter 2

# Preliminaries

## 2.1 Model Checking

For the last decade it has become clear that our current society depends more and more on software, often to the point that lives can be at stake when the software does not work correctly. At the same time, software becomes more and more complex, increasing the odds for errors. One way to increase software quality is to use model checking. Using model checking, a model representing for example a piece of software or a protocol is either manually designed or automatically generated using software. This model can then be used to explore whether certain desired properties or undesired properties hold for this model, and thus for the program represented by the model. A way to do model checking is to generate the complete set of states the program can be in when the model of the software is used and to check if one of those states conflicts with predefined rules. Each state in the model represents a unique condition of the software, and the model describes the possible relation between each of these states. The most common problems when model checking are:

- Knowing how to model your system
- State space explosion

State space explosion describes the phenomenon that the amount of states of a model can grow exponentially with the size of the model. State space explosion is a logical result of trying to represent an intricate system by defining all the states a system can be in, e.g. by introducing parallelism.

Bisimulation reduction is a well known technique to reduce this state space [25, 7]. In this thesis we will present a new way to achieve bisimulation reduction using the Hadoop MapReduce framework[11]. MapReduce is a well-established paradigm in the big data community [16]. Using MapReduce allows for both simple scaling of an algorithm w.r.t. the amount of machines, and migration of the algorithm to another computational cluster. Model checking is a process that is inherently demanding on computing power and memory usage. We think that implementing model checking algorithms using the MapReduce framework can allow users with less computational power to evaluate large models by outsourcing the computing to parties like Amazon EMR [21].

## 2.2 Labeled Transition Systems

A way to represent the state space of a model is by using *Labeled Transition Systems (LTS)*[2]. LTSs represent a way to describe a system by its states, ($\mathcal{S}$) and all the transitions between these states. LTSs therefore consist of all possible states $\mathcal{S}$ a system can be in, and all the transitions with their actions $Act$ between those states. We assume that there exists one special label named $\tau$ (tau), this label describes a non-observable or silent step. This denotes that from the outside of the model we cannot see when this step is taken. It is important to note that having two $\tau$-steps from the same state represent a non-deterministic step. E.g. when reducing a state space and not respecting this property the model loses information about branching processes.[15]

By definition[2], a LTS can be described as a tuple

$$\mathcal{TS} = (\mathcal{S}, Act, \rightarrow, i, \mathcal{AP})$$

where

- $\mathcal{S}$ is the set of states of the system
- $Act$ is the set of actions
- $\rightarrow \subseteq \mathcal{S} \times Act \times \mathcal{S}$ represents the transitions ($t \xrightarrow{\alpha} s$ where $\alpha \in Act$ and $s, t \in \mathcal{S}$)
- $i \in \mathcal{S}$ the initial state of our system

### 2.2.1 Beverage machines

To illustrate matters we will be using some (simplified) models of beverage machines in this thesis. Our beverage machine usually behaves in a predictable manner, one can supply a coin with the action ***c**(oin)*, and then select a product ***a**(pple juice)* or ***b**(eer)*. After the selection a $\tau$-step happens in which the machine probably updates its cash registry, and hopefully it will ***d**(ispence)* a cold beverage. The first two versions of this machine can be seen in figure 2.1 and 2.2. In chapter 4 we will introduce a more complex beverage machine to illustrate branching bisimulation (figure 4.1).

We now have an idea of what a model and a state space is, and how to describe it. Next, we are going to see if we can make this state space a bit smaller.

## 2.3 Bisimulation Reduction

To reduce the state space we will try to replace our LTS with a (much) smaller version that has exactly the same behavior as our original LTS. So our new LTS simulates the old LTS and visa versa, or the old LTS *bi-simulates* the new LTS.

A bisimulation relation $\mathcal{R}$ is a set of states $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ where both states in the relation are said to simulate each other (bisimulation). Intuitively, given $(s_1, s_2) \in \mathcal{R}$, $s_1$ could for example do all the actions that $s_2$ could do and end up in a similar state as $s_2$. The exact same goes for the reversed situation. For a formal description of the bisimulation relation we refer to Chapter 3.

If we have two states that can perfectly simulate the other and end up in the same (or a similar) state, we could not distinguish both states. If we replace all these simulating states with one state we have a reduced version of our original LTS. A common way to calculate the reduced LTS, is by using a partitioning algorithm. We will give a short explanation of the idea of the algorithm, more intricate descriptions are given in the respective chapters. The partitioning algorithm states that each state is bi-similar. It then iterates, trying to find states that are in fact *not* similar, partitioning the state space. The iteration terminates when the algorithm cannot find any non bi-similar states. The termination is guaranteed, since we cannot create more partitions than there are states. At this point the algorithm has found the largest (or coarsest) partitions possible, therefore the result of the algorithm will be the *smallest* LTS that is bisimilar to the original LTS. Figure 2.2 shows us the LTS of a beverage machine, we enabled partition refinement to color all the (strongly) bi-similar states. After reduction, by merging the same colored states we have the original machine shown in 2.1. The formal definition of strong bisimulation can be found in the next chapter.

State space reduction has several added advantages for model checking. Generating a state space in a distributed environment (like Hadoop) allows for more memory usage compared to using a single machine. However, exploring the generated state space in-memory will also require a distributed model checker, since the generated state space does not fit in-memory on a single machine. If we can reduce the state space in such a way that the state space fits on a single (high-end) machine, conventional model checking tools can be used to do the model checking of the generated state space. These tools are highly developed, sparing the effort of having to implement them in a distributed manner.

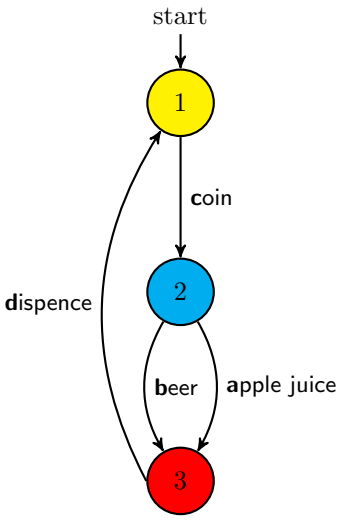



Figure 2.1: Machine 01

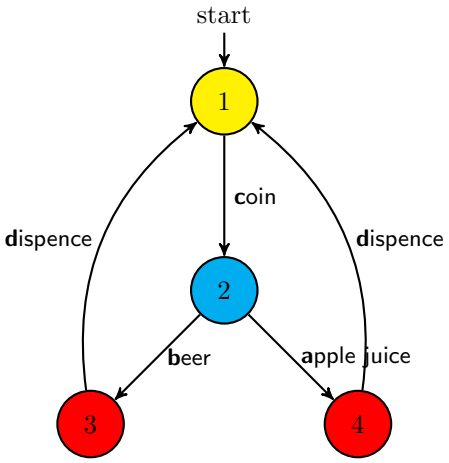



Figure 2.2: Machine 02

### 2.3.1 Signature refinement

The implementations for strong and branching bisimulation that we present in Chapter 3&4 make use of signature refinement. Signature refinement is proposed by Blom and Orzan[8] to allow for more efficient distributed bisimulation reduction. In the algorithm proposed by Kanellakis and Smolka the blocks for each state needs to be calculated and saved in a central table during partitioning[25]. Signature refinement replaces these blocks by defining signatures for each state. States with an equal signature will be assigned to the same partition. The definition for the signature for strong bisimulation is:

$$sig_k(s) = \{(\alpha, part_{k-1}(t)) \mid s \xrightarrow{\alpha} t$$

Where $sig_0(s) = \emptyset$, $part_k(s)$ is the current partition for a given state, and $k$ is the current iteration for the partition refinement. As you can see given a state and its outgoing transitions we can calculate the new signature, and thus partition for this state. In our algorithm we will directly calculate an unique partition number by using a cryptographic hash for the partitions.

$$part_k(s) = hash(sig_k(s))$$

We will lose the option for monotonically increasing partition numbers, but in return all the partitions for a given state can be calculated and assigned solely based on the outgoing transitions for that state and without communication between workers. For branching bisimulation a similar approach will be used based on the work by Blom and Pol[9].

## 2.4 MapReduce

During this project we developed a way to realize partition refinement using the MapReduce framework by Hadoop. Hadoop is a framework initiated by Google to process vast amounts of data[18, 16]. Hadoop offers a framework that handles most of the common tasks needed in distributed implementations, as for example handling the distributed storage of data or errors (e.g. the failure of one of the machines or nodes)[11]. Within the Hadoop-framework there are multiple options for distributed operations, one of these options is the use of distributed algorithms. To develop distributed algorithms, Hadoop has a framework, called MapReduce [16, 38], MapReduce allows easy development of distributed applications. A developer can then easily develop a distributed solution, simply by implementing the MapReduce-framework.

Once an algorithm is implemented in MapReduce it can be run on clusters of various sizes. These clusters can be owned by the researcher, or the researcher could rent time at a large provider like Amazon. In our research we already discovered some promising implementation for CTL-model checking[1, 6, 5, 22]. During this thesis we will focus on algorithms for bisimulation reduction.

MapReduce requires the developer to divide each distributed algorithm in two steps: the Map and the Reduce-phase. The developer writes a function for the Map-phase that accepts a key-value pair ($\langle k, v \rangle$), and outputs another key-value pair ($\langle k', v' \rangle$). In an intermediate step all these key-value pairs are sorted and grouped by $k'$. Custom sorting algorithms will even allow us to do *secondary sorts* on these key-value pairs[34], we will use this in Section 3.3.1. Next, we call the Reducer function with each key and a iteratable with the corresponding values, which we will represent as a list in the pseudo-code ($\langle k', v'[] \rangle$). In some snippets of pseudo-code it is important to note that we loop multiple times over all the values, since `v[]` is an iteratable this is technically not possible. This is fixed either by a carefully chosen secondary sort, guarantying the correct order of `v[]` or by storing the values in `v[]` locally. MapReduce does not allow for communication or shared variables between reducers and mappers, however MapReduce does offer global counters allowing users to keep track of certain statistics. In our pseudo code counters can be identified by the `COUNTER`-prefix. The Reducer-function then uses this data to output a new key-value pair, $\langle k'', v'' \rangle$. Optionally, the result of this reducer can now be fed to another MapReduce function, or iterate the same MapReduce function.
For example, in Listing 2.1 we have a map and reduce function. The goal of the algorithm is to count the total of incoming edges for each state. Given that the map-function is called with $s$ as key and $\xrightarrow{a} t$ as value, where $s \xrightarrow{a} t \in TS$. The map-function than emits $(t, 1)$ for each transition, since we want to count the total amount of incoming edges. After grouping on $t$ all these values are send to the reducer. The reducer only has to sum up all the values and emit the amount of incoming edges together with the corresponding state.

```
input: transition system ⟨s, (a, t)⟩
temp: ⟨s, 1⟩
output: ⟨s, count(s)⟩

MAP(k,v)
    emit(v.t,1)

REDUCE(k,v[])
    emit(k,v.length)
```

Listing 2.1: Counting of incoming transitions per state

# Chapter 3

# Strong Bisimulation

We will present a definition for the given Bisimulation relation followed by a regular (naive) algorithm for the sequential case. Then we show the structure of our algorithm in a flow graph, accompanied by the pseudo code for each of the MapReduce-blocks. Ending with a short discussion on the implementation with design choices and encountered difficulties. We will adhere to the same structure for branching bisimulation in chapter 4.

The first bisimulation reduction algorithm we developed is for **strong bisimulation**.

## 3.1 Definition

Strong bisimulation states that two states are bi-similar if, and only if, they can execute the same action and end up in a bi-similar state.

More formally given two transition systems[2]:

$$\mathcal{TS}_\delta = (\mathcal{S}_\delta, Act_\delta, \rightarrow_\delta, i_\delta), \delta = 1, 2$$

The strong bisimulation relation $\mathcal{R}$ over $\mathcal{TS}_\delta$ is defined as $\mathcal{R} \subseteq \mathcal{S}_1 \times \mathcal{S}_2$

A. $(i_1, i_2) \in \mathcal{R}$

B. for all $(s_1, s_2) \in \mathcal{R}$ it holds:

   1. if $s_1 \xrightarrow{\alpha} s_1'$ then there exists $s_2 \xrightarrow{\alpha} s_2'$ with $(s_1', s_2') \in \mathcal{R}$ and $\alpha \in Act_\delta$
   2. if $s_2 \xrightarrow{\alpha} s_2'$ then there exists $s_1 \xrightarrow{\alpha} s_1'$ with $(s_1', s_2') \in \mathcal{R}$ and $\alpha \in Act_\delta$

The above definition states that the two initial states in both transition systems need to be bisimilar in condition A. Condition B guarantees that each pair of states $(s_1, s_2)$ in $\mathcal{R}$ is actually strongly bisimilar. Meaning that $s_1$ is bisimilar to $s_2$ if and only if each action $\alpha$ to a state $s_1'$ by $s_1$ can be simulated by $s_2$. $s_2$ is not required to make the same $\alpha$ transition to $s_1'$, but it has to make a $\alpha$-step to a state that is bisimilar to $s_1'$ which can be either $s_1'$ itself or $s_2'$ (given that $(s_1', s_2') \in \mathcal{R}$).

```
input: transition system TS = (S, →)
output: table pi, containing the new partitions

reduce()
    for all states s ∈ S do pi[s]:=0 end for
    repeat
        // compute signatures
        for all states s do sig[s]:=∅ end for
        for all transitions (s,α,t) ∈→ do
            insertSig(s,α,pi[t]) end if
        end for
        // reassign pi according to sig
        hashtable := ∅
        count:=0
        for all states s do
            if not sig[s] in hashtable.keys() then
                hashtable.insert(sig[s],count)
                inc(count)
            end if
        end for
        for all states s do
            pi[s]:= hashtable.lookup(sig[s])
        end for
    until pi is stable

insertSig(t,a,ID)
    if not((α,ID) ∈ sig[t]) then
        sig[t]:=sig[t] ⋃ {(α,ID)}
    end if
```

Listing 3.1: sequential implementation

## 3.2 Regular algorithm

In order to minimize a state space using bisimulation reduction we have to find the largest set ($\mathcal{R}$) of bisimulation relations. A way to do this, is using *partition refinement*[2, 25]. A simple representation of such an algorithm is given in Listing 3.1. This is a simplified representation of the algorithm proposed by Blom and Orzan[8]. The algorithm works with signatures represented by the variable *sig*, initially each state has an empty and thus equal signature. States of equal signature will be in equal partitions, these partitions will be stored in *pi*. The algorithm iteratively refines these partitions by calculating new signatures, based on the outgoing transitions for each state. The partition numbers are unique since each new signature is given an unique identifier by *count*. The algorithm terminates when the amount of partitions (or unique signatures) stays stable. Termination is guaranteed, because the amount of signatures will increase or stay stable for each iteration, and the total amount of unique signatures can not exceed the total amount of states.

In order to see how this algorithm works we will look at our beverage machines. In figure 2.2 we can see our less optimal beverage machine, which we claimed is strongly bisimilar with the beverage machine next to it. We will go through each iteration of our algorithm and calculate the signatures for each iteration. In table 3.1 we show these steps. The columns represent each state of the beverage machine. The iterations are represented by the rows. We can see that after two iterations we can conclude that we have found the coarsest partitioning. We reassign state numbers according to the generated signatures, then we can see that we have created the LTS shown in figure 2.1.

Table 3.1: Strong bisimulation reduction on the beverage machine

| | State | 1 | 3 | 4 | 5 | Partitions |
|---|---|---|---|---|---|---|
| **Iteration 0** | **Pi** | 0 | 0 | 0 | 0 | **1** |
| **Iteration 1** | **Signature** | {(c,0)} | {(a,0),(b,0)} | {(d,0}) | {(d,0)} | |
| | **Pi** | 0 | 1 | 2 | 2 | **3** |
| **Iteration 2** | **Signature** | {(c,1)} | {(a,2),(b,2)} | {(d,0}) | {(d,0}) | |
| | **Pi** | 0 | 1 | 2 | 2 | **3 (stable)** |
| | **New States** | **1** | **2** | **3** | **3** | |

## 3.3 MapReduce implementation

The MapReduce implementation is based on the work by Blom and Orzan[8] and Schätzle et al.[32]. As stated before, the algorithm in listing 3.1 is similar to the algorithm proposed by Blom and Orzan. For our MapReduce implementation we can not make use of a shared table to store the variables *sig* or *pi*. For the calculation of the partition table *pi* a shared table would be an easy solution. In order to be able to do distributed strong bisimulation reduction without a shared hashtable some non-trivial problems need to be solved. In listing 3.1 on line 28 we construct a signature by creating a set of the actions and previous partition for the target state of all the outgoing transitions, creating inductive signatures[9]. On line 15 we use these sets to determine new partitions. Alternatively we could also store this whole set for each state, but this would result in a gigantic blowup of the size of the model. Another way to generate a unique partition number based on the set of states and partition numbers is the use of a cryptographic hash. This guarantees an unique partition number, without communication between workers during the partitioning. A disadvantage of this method is that we lose the fact that $n$ partitions will be numbered $0..n-1$. The signatures for a given state $s$ are thus generated by hashing the set of all outgoing transitions being: $sig^{k+1} = hash(\{(\alpha, sig^k(t)) | s \xrightarrow{\alpha} t \in \rightarrow\})$.

An overview of our algorithm is shown in the flowchart in figure 3.1. Each of the steps in the graph are than described in subsection 3.3.2. In the flowgraph the *createSigs* step executes the MapReduce version for the inside of the loop (line 7-23). The *SigCount* counts the signatures, which is needed for the condition on line 24 in the sequential implementation.

### 3.3.1 Flowchart

In the flowchart in figure 3.1 we have outlined the developed algorithm. Between each Map and Reduce action we have placed the submitted key-value pairs. Often the keys consist of two values, allowing for secondary sorting on the flags supplied with the key. The most important step in the flowchart is the *BiSim*-step, here we calculate the new signatures for each state. To calculate these signatures the Mapper provides the reducer with the signature of a given state, and all the outgoing transitions for that state. After the generation of the new signatures a separate Map and Reduce function counts the amount of unique signatures in *SigCount*. If we look at the original algorithm in listing 3.1 *CreateSigs* is similar to the step taken on lines 8-23, where the use of *pi* is replaced with a hashing function. *CountSigs* represents the terminating condition on line 24.

A further explanation for each key-value pair and MapReduce function can be found in the next section (3.3.2).

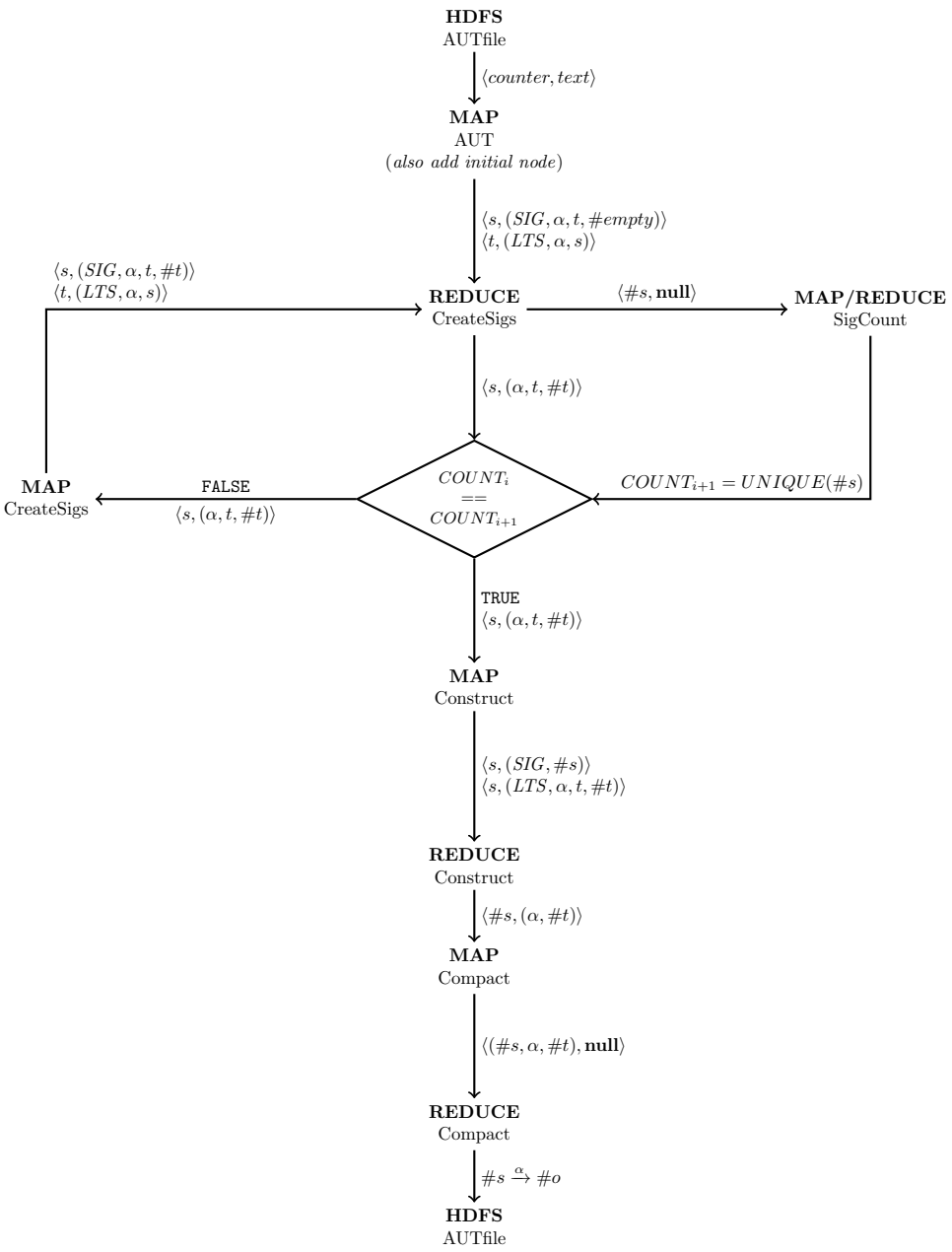



Figure 3.1: Flowchart - Strong Bisimulation with MapReduce

### 3.3.2 Pseudo code

This section contains pseudo code for the used Mappers and Reducers for the strong bisimulation reduction. Each listing starts with the expected input. The intermediary format that is emitted by the Mapper, which will be sorted on both the state and the supplied flag, before it is offered as input to the Reduce function. The output(s) specify the expected out for the reducer which will be offered to another MapReduce task (except for the very last run). Along with each listing an explanation of that step is provided, and the listing will also contain some additional comments.

#### Loading the model

We read the transitions from an existing state space in the Aldebaran (*.AUT) format[26]. The use of a pre-defined format for allows our tools interoperability with already existing tools. The Aldebaran format consists of a header containing the name of the start state, the amount of states and the number of transitions. Each following line describes a transition, we could state that we simply get supplied the set $\rightarrow$ from $\mathcal{TS}$. Each transition is emitted with the label "$LTS$", for each state we also emit the signature, which in the initial mapping is the hash for an empty set. The signature for a given state $s$ will be shorthanded #s.

```
input: transition system (counterk, textv) // k is not used, v contains a line of text
output: ⟨t, (SIG, α, t, #t)⟩              // for the signature of state t
        ⟨s, (LTS, α, t, #t)⟩              // a transition for state s


MAP(k,v)
    #empty = []
    // retrieve the transition from v
    (s, α, t) ← parse_text(v)
    // emit the signature for t
    emit(t, (SIG, t, #empty))
    // emit the transition
    emit(s, (LTS, α, t, #empty))
```

Listing 3.2: Reading the AUT file (MAP)

#### Creating new signatures

All the keys will be automatically sorted on the state and have a secondary sort on the type (*SIG*,LTS). Therefore the reducer will first receive all the values with *SIG* for a given state, and then all the remaining values (with *LTS*). The reduce function will therefore first receive each outgoing transition for a given key. Based on these transitions a new signature can be calculated. The calculated signature will then be emitted for every incoming transition of this state. This way the signature will propagate backwards through the LTS. Additionally, we also emit the calculated signature to a separate file. It is important to note that we loop twice over all the values, since `v[]` is an iteratable this is technically not possible, as stated in section 2.4. In this case this is solved by the secondary sorting on the type (*SIG*,*LTS*).

```
input: ⟨s, (α, t, #t)⟩                    // transition as input
temp: ⟨s, (SIG, α, t, #t)⟩  // for the signature of state t
      ⟨t, (LTS, α, s)⟩ // a transition for state s
output_1: ⟨s, (α, t, #t)⟩                 // transition with new signature
output_2: ⟨#s, null⟩                      // signature for counting


MAP(k,v)
    // emit the signature for outgoing transitions from s
    emit(k.s, (SIG, v.t, v.#t))
    // emit the reversed transitions
    emit(v.t, (LTS, v.α, k.s))

REDUCE(k,v[])
    new = ∅
    for v in v[] where v.type==SIG
        // add the outgoing signatures to the new−set
        new = new ⋃(v.α, v.#t)
        // emit the transitions with #new
    for v in v[] where v.type==LTS
        emit_1(v.s, (v.α, k.t, hash(new)))
    // emit our new signature for counting
    emit_2(hash(new), null)
```

Listing 3.3: CreateSigs

#### Statecount

The emitted signatures from listing 3.3 can now be counted. The output of the map function is sorted and all the duplicate keys are removed. In the reducer we can therefore simply update a global counter, counting the total of discovered states. The algorithm will compare this counter to the old amount of sigs. When the number of signatures is not stable we will continue to the map function in listing 3.3, according to the flowchart in figure 3.1. When the amount of signatures s stabilized we can construct the reduced LTS.

```
input: ⟨#new, null⟩ // new signature
temp: ⟨#new, null⟩
output: null

MAP(k,v)
    emit(k, null)

REDUCE(k,v[])
    SIG_COUNTER.inc();
```

Listing 3.4: CountSigs

#### Constructing the reduced LTS

To construct the new LTS we need to convert the original LTS consisting of the triples $s \xrightarrow{\alpha} t$ to a minimized LTS according to the partitions. We can achieve that by substituting the partition numbers creating the set of triples $\#s \xrightarrow{\alpha} \#t$. For a given transition the map function will pass the signature for that state, and all the outgoing transitions, already containing the signature

for the destination. The reduce function will than construct $\#s \xrightarrow{\alpha} \#t$ for each given transition. To remove the duplicate transitions we have the additional task in listing 3.6.

```
input: ⟨s, (α, t, #t)⟩                    // transition
temp: ⟨t, (SIG, #t)⟩          // signature
      ⟨s, (LTS, α, t, #t)⟩ // transition
output: ⟨#s, (α, #t)⟩                    // transition with signatures


MAP(k,v)
    // emit the signature for t
    emit(v.t, (SIG, v.#t))
    // emit the transition
    emit(k.s, (LTS, v.α, v.t, v.#t))

REDUCE(k,v[])
    select v from v[] where v.type==SIG
        #s = v.#t
    for v in v[] where v.type==LTS
        emit(#s, (v.α, v.#t))
```

Listing 3.5: Construct LTS

### Output to AUT

Using the sorting function of MapReduce we can remove all duplicate transitions. We automatically sort and remove duplicate keys between the mapper and the reducer. The result will be the reduced transition system.

```
input: ⟨#s, (α, #t)⟩          // transition
temp: ⟨(#s, α, #t), null⟩ // sort by transition
output: ⟨#s, (α, #t)⟩       // output transition


MAP(k,v)
    emit((k.#s, v.α, v.#t), null)

REDUCE(k,v[])
    emit(k,null)
```

Listing 3.6: Compact LTS

# Chapter 4

# Branching Bisimulation

## 4.1 Definition

Branching bisimulation is quite similar, only it preserves the branching structure of the TS better. Internally, the system can take as many *unobservable* steps ($\tau$-steps) as needed, as long as the observable or visible steps are identical, and when the starting and the end states of the $\tau$-steps are in the bisimulation relation with the same state. We will write the definition by [33] in the style used in Chapter 3. [2, 33] More formally given two transition systems:

$$\mathcal{TS}_\delta = (\mathcal{S}_\delta, Act_\delta, \rightarrow_\delta, i_\delta), \delta = 1, 2$$

The branching bisimulation relation $\mathcal{R}$ over $\mathcal{TS}_\delta$ is defined as $\mathcal{R} \subseteq \mathcal{S}_1 \times \mathcal{S}_2$

A. $(i_1, i_2) \in \mathcal{R}$

B. for all $(s_1, s_2) \in \mathcal{R}$ it holds:

1. if $s_1 \xrightarrow{\alpha} s_1'$ then there exists $s_2 \xrightarrow{\tau}{}^* s_2'' \xrightarrow{\alpha} s_2''' \xrightarrow{\tau}{}^* s_2'$
   with $(s_1, s_2'') \in \mathcal{R} \wedge (s_1', s_2''') \in \mathcal{R} \wedge (s_1', s_2') \in \mathcal{R}$ and $\alpha \in Act_\delta \backslash \{\tau\}$
2. if $s_2 \xrightarrow{\alpha} s_2'$ then there exists $s_1 \xrightarrow{\tau}{}^* s_1'' \xrightarrow{\alpha} s_1''' \xrightarrow{\tau}{}^* s_1'$
   with $(s_1'', s_2) \in \mathcal{R} \wedge (s_1''', s_2') \in \mathcal{R} \wedge (s_1', s_2') \in \mathcal{R}$ and $\alpha \in Act_\delta \backslash \{\tau\}$
3. if $s_1 \xrightarrow{\tau} s_1'$ then there exists $s_2 \xrightarrow{\tau}{}^* s_2'$ with $(s_1', s_2') \in \mathcal{R}$
4. if $s_2 \xrightarrow{\tau} s_2'$ then there exists $s_1 \xrightarrow{\tau}{}^* s_1'$ with $(s_1', s_2') \in \mathcal{R}$

The above definition states that the two initial states in both transition systems need to be bisimilar in condition A. Condition B guarantees that each pair of states $(s_1, s_2)$ in $\mathcal{R}$ is branching bisimilar. This means that $s_1$ is bisimilar to $s_2$ if and only if each action $\alpha$, where $\alpha \neq \tau$, to a state $s_1'$ by $s_1$ can be simulated by $s_2$, possibly by first taking an arbitrary amount of $\tau$-steps, then taking the $\alpha$-action followed by zero or more $\tau$-steps. The step needs to be simulated, meaning that the transitions need to end in a bisimilar state ($(s_1', s_2') \in \mathcal{R}$).

If $s_1$ takes a $\tau$-step to $s_1'$ then $s_1$ and $s_2$ are bisimilar if and only if $s_2$ also takes zero or more $\tau$-steps to a state that is bisimilar to $s_1'$.

```
input: transition system TS = (S, →)
output: table pi, containing the new partitions

reduce()
    for all states s ∈ S do pi[s]:=0 end for
    repeat
        // compute signatures
        for all states s do sig[s]:=∅ end for
        for all transitions (s,α,t) ∈→ do
            // select all invisible steps
            if not (α=τ and pi[s]=pi[t]) then
                insertSig(s,α,pi[t]) end if
        end for
        // reassign pi according to sig
        hashtable := ∅
        count:=0
        for all states s do
            if not sig[s] in hashtable.keys() then
                hashtable.insert(sig[s],count)
                inc(count)
            end if
        end for
        for all states s do
            pi[s]:=hashtable.lookup(sig[s])
        end for
    until pi is stable

insertSig(t,a,ID)
    if not((α,ID) ∈ sig[t]) then
        sig[t]:=sig[t] ⋃ {(α,ID)}
        for all s ∈ S such that s --τ--> t and pi[s]=pi[t] do
            insertSig(s,α,ID)
        end for
    end if
```

Listing 4.1: sequential implementation

## 4.2 Regular algorithm

The single threaded algorithm shown in listing 4.1 is the sequential algorithm in the work by Blom and Orzan [7]. For each state a set is created similar as in listing 3.1 but with an additional constraint. This constraint tells us to ignore all states that are *invisible*, meaning all transitions containing a $\tau$-step where the source and destination are in a shared partition. This set of *visible* transitions is than propagated back along all *invisible* $\tau$-transitions. The moment the propagation along the *invisible* step is finished the signature for each state is created based on these (propagated) sets.

In order to see how this algorithm works we will introduce a new beverage machine. In figure 4.1 we can see our new beverage machine, which we claim is branching bisimilar with the beverage machines we have seen before. We will go through each iteration of our algorithm and calculate the signatures for each iteration, our steps can be seen in table 4.1. In this table the columns represent each state of the beverage machine. The iterations are represented by the rows. During the signature phase we create a signature for all the visible transitions. At the transition phase you can see the exchange of the signatures between state 2 and 3. After renumbering we will have a reduced LTS similar to machines shown in figure 2.1.

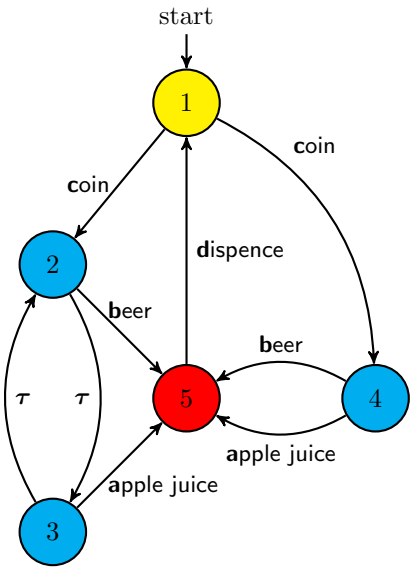



Figure 4.1: Machine 03

## 4.3 MapReduce implementation

For our MapReduce implementation we needed a more elaborate scheme, allowing for the nested loops in the original algorithm. The outer loop is similar to the original loop in the strong bisimulation implementation. More specifically lines 7-13 correspond to the *Label*-step in the flowgraph. The inner loop provides the propagation along the invisible transitions. In the sequential implementation this happens on lines 30-32. Figure 4.2 shows this more intricate scheme. The propagation step of the algorithm requires us to keep a set of all the states ($\Sigma s$), opposed to only passing a hash ($\#s$) of the set.

Table 4.1: Branching bisimulation on the new beverage machine

| | **State** | **1** | **2** | **3** | **4** | **5** | **Partitions** |
|---|---|---|---|---|---|---|---|
| **Iteration 0** | **Pi** | 0 | 0 | 0 | 0 | 0 | **1** |
| **Iteration 1** | **Signature** | {(c,0)} | {(b,0)} | {(a,0)} | {(a,0),(b,0})} | {(d,0)} | |
| | **Traversed** | - | {(a,0),(b,0})} | {(a,0),(b,0})} | - | - | |
| | **Pi** | 0 | 1 | 1 | 1 | 2 | **3** |
| **Iteration 2** | **Signature** | {(c,1)} | {(b,2)} | {(a,2)} | {(a,2),(b,2})} | {(d,0)} | |
| | **Traversed** | - | {(a,2),(b,2})} | {(a,2),(b,2})} | - | - | |
| | **Pi** | 0 | 1 | 1 | 1 | 2 | **3** |
| | **New States** | **1** | **2** | **2** | **2** | **3** | |

### 4.3.1 Flowchart

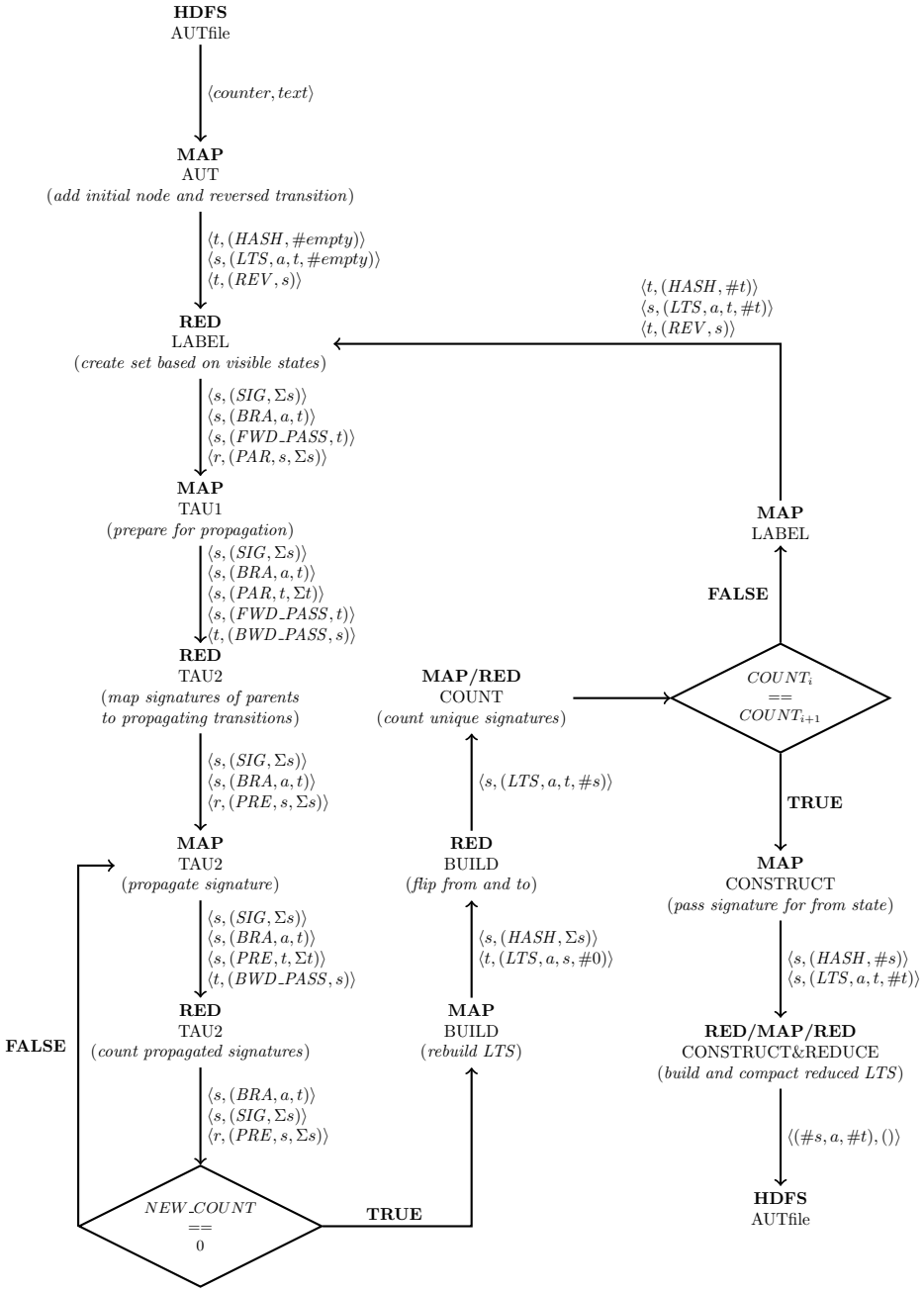



Figure 4.2: Flowchart - Branching Bisimulation with MapReduce

### 4.3.2 Pseudo code

#### Reading from AUT file

A difference with branching simulation is that we have to pass the reversed transition to be able to propagate the signature we are going to create along the *invisible* steps.

```
input: transition system counterk, textv) // we only use v, which contains our text
output: ⟨t, (HASH, #empty)⟩              // pass the signature for state t
        ⟨s, (LTS, a, t, #empty)⟩         // pass the transition
        ⟨t, (REV, s)⟩                    // pass the reversed transition

MAP(k,v)
    #empty = []
    // retrieve the transition from v
    (s, α, t) ← parse_text(v)
    // pass hash of the state
    emit(t, (HASH, #empty))
    // pass the transition
    emit(s, (LTS, a, t, #empty))
    // pass reverse transitions for later use
    emit(t, (REV, s))
```

Listing 4.2: Reading the AUT file (MAP)

**Creating new signatures**

We split up our LTS in the visible and invisible part. The invisible and visible part we respectively output with the type *FWD_PASS* and *BRA*. We use all the visible transitions for a given state to create the set sigset. This set we emit for this state and also coupled to all the incoming transitions.

```
input: ⟨s, (LTS, α, t, #t)⟩        // the transition
temp: ⟨s, (HASH, #s)⟩              // pass the hash
      ⟨s, (LTS, a, t, #t)⟩         // pass the transition
      ⟨t, (REV, s)⟩                // pass the reversed transition
output: ⟨s, (SIG, Σs)⟩             // the signature set for s
      ⟨s, (BRA, a, t)⟩             // the visible transition
      ⟨s, (FWD_PASS, t)⟩           // the invisible transition
      ⟨r, (PAR, s, Σs)⟩            // the reversed transitions

MAP(k,v)
    // pass hash of the state
    emit(t, (HASH, #t))
    // pass the transition
    emit(s, (LTS, a, t, #t))
    // pass reverse transitions for later use
    emit(t, (REV, s))

REDUCE(k,v[])
    // get the signature for the current state
    select v ∈ v[] where v.type==HASH
        fromsig = v.#s
    // emit the invisible transitions
    for v ∈ v[] where  v.type==LTS ∧
                      (v.a == τ ∧ fromsig == v.#t)
        emit(s, (FWD_PASS, t))
    // calculate the signatureset
    for v ∈ v[] where  v.type==LTS ∧
                      not (v.a == τ ∧ fromsig == v.#t)
        sigset = sigset⋃(v.a, v.#t)
    // emit the signatureset for this state
    emit(k.s, (SIG, sigset))
    // emit the visible transitions
    for v ∈ v[] where  v.type==LTS ∧
                      not (v.a == τ ∧ fromsig == v.#t)
        emit(k.s, (BRA, v.a, v.t))
    // emit the reversed transitions with the signatureset for propagation purposes
    for v ∈ v[] where v.type==REV
        emit(v.r, (PAR, k.s, sigset))
```

Listing 4.3: Creating new signatures

**First tau step**

The first $\tau$-step looks up the passed back signatures in $PAR$ for all the transitions in $FWD\_PASS$. We add all the new signatures to this state and pass this new set to all the propagating invisible transitions. We count all the new signatures that are added in a global counter.

```
input: ⟨s, (SIG, Σs)⟩          // signatureset for s
       ⟨s, (BRA, a, t)⟩        // visible transition
       ⟨s, (FWD_PASS, t)⟩      // invisible transition
       ⟨r, (PAR, s, Σs)⟩       // reversed transition for propagation
temp: ⟨s, (SIG, Σs)⟩           // signatureset for s
       ⟨s, (BRA, a, t)⟩        // visible transition
       ⟨s, (PAR, t, Σt)⟩       // invisible transition for propagation
       ⟨s, (FWD_PASS, t)⟩      // invisible transition
       ⟨s, (BWD_PASS, r)⟩      // reversed invisible transition
output: ⟨s, (SIG, Σs)⟩         // signatureset for s
       ⟨s, (BRA, a, t)⟩        // visible transition
       ⟨r, (PRE, s, Σs)⟩       // invisible transition

MAP(k,v)
    // emit the invisible transitions reversed
    if (v.type==FWD_PASS)
        emit(v.t, (BWD_PASS, k.s))
    // pass through
    emit(k,v)

REDUCE(k,v)
    sigs = ∅
    newsigs = ∅
    // pass the visible transitions
    for v ∈ v[] where v.type==BRA
        emit(k,v)
    // get the signatureset for s
    select v ∈ v[] where v.type==SIG
        sigs = v.Σs
    // get all the invisible transitions
    for v ∈ v[] where  v.type==FWD_PASS
        // get the signatureset from the reversed transitions
        for w ∈ v[] where  v.type==PAR ∧ v.t == w.t
            newsigs = newsigs ⋃(w.Σt)
    // emit how much new signatures we found
    SIG_COUNTER.inc(|newsigs\sigs|);
    // add the new signatureset to the signatureset
    sigs = sigs ⋃ newsigs
    // emit our new signatureset
    emit(s, (SIG, sigs))
    // emit our invisible transitions with the signatureset
    for v ∈ v[] where v.type==BWD_PASS
        emit(v.r, (PRE, k.s, sigset))
```

Listing 4.4: First tau step

**Propagating along tau steps**

The map-function copies the invisible sets for the propagation to *BWD_PASS*. In the reducer we emit the visible states. We calculate the propagating signatures and pass these for the current state, and also coupled to the invisible transitions for propagation. This step will be repeated until no more changes in the signatures occur.

```
input: ⟨s, (SIG, Σs)⟩            // signatureset for s
       ⟨s, (BRA, a, t)⟩           // visible transition
       ⟨s, (PRE, t, Σt)⟩         // invisible transition
temp: ⟨s, (SIG, Σs)⟩             // signatureset for s
       ⟨s, (BRA, a, t)⟩           // visible transition
       ⟨s, (PRE, t, Σt)⟩         // invisible transition
       ⟨s, (BWD_PASS, r)⟩        // reversed invisible transition for propagation
output: ⟨s, (SIG, Σs)⟩           // signatureset for s
       ⟨s, (BRA, a, t)⟩           // visible transition
       ⟨r, (PRE, s, Σs)⟩         // invisible transition

MAP(k,v)
    for v ∈ v[] where v.type==PRE
        emit(v.t, (BWD_PASS, k.s))
    emit(k,v)

REDUCE(k,v)
    sigs = ∅
    newsigs = ∅
    // pass the visible transitions
    for v ∈ v[] where v.type==BRA
        emit(k,v)
    // get the signatureset for s
    select v ∈ v[] where v.type==SIG
        sigs = v.Σs
    // get all the invisible transitions
    for v ∈ v[] where  v.type==PRE
        newsigs = newsigs ⋃(w.Σt)
    // emit how much new signatures we found
    SIG_COUNTER.inc(|newsigs\sigs|);
    // add the new signatureset to the signatureset
    sigs = sigs ⋃ newsigs
    // emit our new signatureset
    emit(s, (SIG, sigs))
    // emit our invisible transitions with the signatureset
    for v ∈ v[] where v.type==BWD_PASS
        emit(v.r, (PRE, k.s, sigset))
```

Listing 4.5: Propagating along tau

### Reconstructing LTS

Based on the set of signatures for a given set we create a hash. This hash is emitted for the state. Both the visible and the invisible transitions are emitted with the to-state as key. The reducer retrieves the hash for a given state s. This hash is then added to all the transitions, and the to- and from-states are flipped to recreate the original transitions with the new signatures added.

```
input: ⟨s, (SIG, Σs)⟩                // signatureset for s
       ⟨s, (BRA, a, t)⟩               // visible transition
       ⟨s, (PRE, t, Σt)⟩              // invisible transition
temp: ⟨s, (HASH, #s)⟩                // hash for s
       ⟨s, (LTS, a, r, #empty)⟩      // transitions for s
output: ⟨r, (LTS, a, s, #s)⟩         // transitions for s

MAP(k,v)
    sigset = ∅
    // get the signatureset for s
    select v ∈ v[] where v.type==SIG
        sigset = v.Σs
    // create a new signature for s and emit it
    emit(s, (HASH, hash(sigset)))
    // emit all visible transitions reversed
    for v ∈ v[] where v.type==PRE
        emit(v.t, (LTS, v.a, k.s, #empty))
    // emit all invisible transitions reversed
    for v ∈ v[] where v.type==BRA
        emit(v.t, (LTS, τ, k.s, #empty))

REDUCE(k,v)
    // get the signature for s
    select v ∈ v[] where v.type==HASH
        sig = v.\#s
    // add the signature to the transitions and un−reverse them
    for v ∈ v[] where v.type==LTS
        emit(v.r, (LTS, v.a, k.s, sig))
```

Listing 4.6: Reconstructing LTS

### Counting signatures

We emit all the signatures and after sorting we can simply count all the unique signatures.

```
input: ⟨s, (LTS, a, t, #t)⟩ // transitions
temp: ⟨(#s), )                // signature for s
output: null

MAP(k,v)
    // emit the signature
    emit((#t), )

REDUCE(k,v[])
    // count the signatures
    SIG_COUNTER.inc()
```

Listing 4.7: Counting signatures

### Constructing the LTS

To construct the new LTS is similar to the construction for the strong bisimulation: we need to convert the original LTS consisting of the triples $s \xrightarrow{\alpha} t$ to a minimized LTS according to the partitions. We can achieve that by substituting the partition numbers creating the set of triples $\#s \xrightarrow{\alpha} \#t$. For a given transition the map function will pass the signature for that state, and all the outgoing transitions, already containing the signature for the destination. The reduce function will then construct $\#s \xrightarrow{\alpha} \#t$ for each given transition.

```
input: ⟨s, (LTS, a, t, #t)⟩   // transition for s
temp: ⟨s, (HASH, #s)⟩    // signature for s
      ⟨s, (LTS, a, t, #t)⟩ // transitions for s
output: ⟨(#s, a, #t), ()⟩     // transition for s

MAP(k,v)
    for v ∈ v[]
        // emit the transition
        emit(k,v)
        // emit the signature for t
        emit(t, (HASH, #t))

REDUCE(k,v)
    // retrieve the hash for
    select v ∈ v[] where v.type==HASH
        sig = v.\#s
    // emit the transitions
    for v ∈ v[] where v.type==LTS
        emit((sig, v.a, v.#t), ())
```

Listing 4.8: Constructing the LTS

### Compacting the LTS

For the compacting we can leverage MapReduce to output a list without duplicate transitions.

```
input: ⟨(#s, a, #t), ()⟩ // transition for s
temp: ⟨(#s, a, #t), ()⟩ // transition for s
output: ⟨(#s, a, #t), ()⟩ // transition for s

MAP(k,v)
    emit(k,null)

REDUCE(k,v[])
    emit(k,null)
```

Listing 4.9: Compacting the LTS

# Chapter 5

# Implementation & Optimization

We would like to elaborate a bit on the more technical details of the code of both the algorithms. The code as shown in the pseudo code above, along with the flow graphs should give an impression on how to implement bisimulation reduction with MapReduce. However the actual implementation in Java with the MapReduce framework is a bit more elaborate.

## 5.1 Implementation

### 5.1.1 File format

Reading the model requires us to have the model in a format that is readable using the MapReduce framework. A common and simple way to read files in MapReduce is to retrieve data from a text file. Therefore we have chosen to first format our models in the Aldebaran format. Between each iteration however it is not recommended to use a text based file format. Hadoop offers the use of sequence files. The sequence files can contain the key value pairs in a binary format, which saves a lot of space. For the signature counting in the strong bisimulation we have chosen a separate approach. To reduce the bandwidth used for the counting we output a separate file containing only the signatures during the signature creation. In a small scale experiment we experienced a minor speed-up of up to 2x for large models.

### 5.1.2 Signatures

The signatures themselves needed to have a few properties. First of all, the signatures need to be relatively small, we achieved that by using a hashing function. However, we want to be sure we do not have any hash collisions. A hash collision could mean that two states suddenly could have a similar signature when they are not supposed to have that. An easy way to make sure that collisions are virtually impossible is to use a cryptographic hash. The cryptographic hash we chose (SHA-256) has such low chances on a hash collision, that we are safe to assume that it will never happen. A quick calculation, assuming that SHA-256 is a proper hashing algorithm, learns that based on the birthday problem[3] we can approximate the chance $p$ on a collision given $n$ signatures to be:

$$p \approx \frac{1}{2}\left(\frac{n}{2^{128}}\right)^2$$

Meaning that when we have significantly less states than $2^{128}$ we can safely ignore collisions. The disadvantage of using a cryptographic hash is that they are often designed to be slow in order to prevent attacks on e.g. hashed passwords[28]. We decided that SHA-256 is still reasonably fast with the advantage that we can be sure that we will not have hash collisions.

### 5.1.3 Technicalities

As you might remember, between each map and reduce phase the keys get sorted and coupled to a list of values. As can be seen in the flowcharts in figure 3.1&4.2 we have keys that are pairs with a type and a state. To order these keys correctly we had to rewrite a custom partitioner

and grouper. We have written this partitioner in such a way that all key value pairs with the same state in the key will be offered to the same reducer. However, the keys will be sorted based on the type of the key-value pair. This way we can ensure that the values will always arrive in a predetermined order. Otherwise we could for example not retrieve the signature of a key before we attach it to later transitions as can be seen in listing 3.3.

If you look at listing 4.3 you can see that we output the key value pair $(\{PAR, r\}, \{s, \Sigma s\})$. The $\Sigma s$ however is quite a large variable. It basically is a set containing pairs with a label and a byte array. To make sure that our value is correctly written to disk by Hadoop required research in the data structures in Hadoop that allow this.

## 5.2 Potential Optimizations

Designing the branching bisimulation algorithm we thought about some worst case scenarios. A possible scenario is that we might get very long invisible tau steps, which requires a lot of iterations. Observing that iterations are quite expensive with our algorithm, we tried to devise a solution. Our proposed solution was to try and cut some of these long tau steps in pieces. We decided to focus at states that have both visible and invisible outgoing transitions. By simply ignoring all the invisible steps for these states we could stop the propagation of long tau steps. Our conjecture was, that in subsequent iterations these signatures would still propagate and thus the branching bisimulation would still be correct. However this suspicion proved wrong. As a counter example we will take the beverage machine from figure 4.1. When we would ignore the silent steps in state 2 and 3 they would be split up after the first iteration. This is a coarser partition than what should be allowed, because state 2,3&4 belong to the same partition. Therefore we will not further pursue this option.

Discussing the previous optimization, another possible optimization came up. It would of course be allowed to make our partition less coarse, if done the correct way. A possibility is to not use all the possible labels in the calculation of a signature. As long as our last (and stabilizing) refinement does use all the labels. While propagating along the silent steps we could keep track which labels are still propagating. Labels that take a long time to propagate could be ignored for that round. In later rounds the long tau chains responsible for this behavior might been broken up and allow for a faster propagation off the labels.

A different approach is to use the transitive closure for all the (non-branching) tau-steps. This would prevent the massive amount of inner iterations, since all the signatures propagate at once. The time-complexity for n states is $O(log(n))$. However, in the worst case the required space could be up to $O(n^2)$ [24]. We have done a small scale experiment and calculated the transitive closure for a subset of models. On average the required space for the transitive closure was in excess of 100 times the amount of non-branching tau-steps. Therefore we concluded that the use of the full transitive closure for the propagation along the tau-steps is not a promising optimization.

# Chapter 6

# Experiments

We have benchmarked both of our algorithms. To run the models we needed a cluster of several computers to run this on. Luckily we were in the position to make use of the CTIT-cluster of the University of Twente. We have benchmarked both the strong and branching bisimulation algorithms on a set of models. This set of models describes several protocols and models suited for benchmarking. The size of the models varied from small to very large. The smallest model only has 289 states and 1224 transitions. The largest model available has 7.041.674.929 states, this is already beyond the maximal value of an unsigned 32 bit integer which can be maximal $2^{32} - 1$ (4.294.967.295).

## 6.1 Experimental setup

For our setup we have used the cluster of CTIT. This cluster consists of 44 nodes for the calculation of our MapReduce tasks. The nodes each have a single Opteron 4386 processor and 64GB of memory. Our main program runs on a central node, from where it offers the MapReduce tasks for processing by the cluster. This main program also logs our benchmark information. We keep track of information relevant about the model like the amount of signatures after each iteration. We also keep track of the time spend for each iteration.

### 6.1.1 Models

Our models are part of the VLTS-benchmark[14]. The state space is converted using LTSmin[10] to the Aldebaran format. The Aldebaran format is very suitable to read with MapReduce since it is a plain text file containing each transition on a separate line. For the larger models this is a serious drawback, since the files will get too large for storing as a simple text file. Since for larger models we can therefore not use the Aldebaran format we looked at other formats offered by LTSmin. We eventually decided on using the ".dir" format, which is an uncompressed variant of the Generic Container Format used by LTSmin[10]. We have developed a separate program that can load these uncompressed state spaces and convert them directly to the sequence file format we already use internally in our algorithm. We have added the lift_7-model, which is similar to cwi_2165_8723 (lift 5) and cwi_33949_165318 (lift 6), except for the size of the model. This allows us for easy comparison on the scaling of the algorithm with respect to the state space. The VLTS-benchmark also includes vasy_40_60, a model that has a state space specifically designed to require a lot of transitions with 20002 transitions for strong bisimulation reduction. The full information on all the models can be found in table 8.1.

### 6.1.2 Results

By looking at the metrics for several models we quickly could deduce a few crucial facts about model checking with hadoop. The execution time for each iteration of a given model is roughly similar. This can be explained by the fact that for each iteration we pass the whole transition system as supplied at the start of the reduction process. Since we cannot know how the coarsest partition looks like there is no possibility to prune duplicate transitions *before* the final coarsest

partition is found. This can be easily shown by defining a LTS where there exist two labels, **a** and **b**. Initially all the states belong to one single partition. Now we could merge all the duplicate transitions we would have a lot of **a**-transitions going from this single partition to this same partition. It is easy to see that without knowing the final (coarsest) partitioning we might merge a lot of transitions that in the final LTS should be separate transitions. The similar calculation times per iteration therefore tells us that the main time spend in each iteration is corresponding to the amount of transitions of our original model. An advantage of this is that with a single execution of the algorithm we still can have a statistically fair estimation of the execution time for each transition, by taking the average of the time spend for each iteration of a given model. In the tables below you can see the average execution time for a iteration on a given model, together with the standard deviation over this set. The small deviation tells us that data-throughput is the main bottleneck for our implementation, and we thus have a fair approximation. Taking the average for the MapReduce part of the algorithm also has another advantage. Since the cluster is shared with other users the scheduler may schedule some other tasks for other users in between our execution. In the averaging the incidental interleaving for other tasks is therefore ignored.

## 6.2 Strong bisimulation

We ran our bisimulation algorithm for the whole suite, increasing the size of the model for each run. In table 6.1 the results for these runs can be viewed. Each row starts with the model name and the amount of states for the given model, we than have the averaged time for each "BiSim"-action (see figure 3.1). The column "Sig" contains the time used for each signature counting. The last three columns show the total amount of iterations used for the calculation, the cumulative time for the reduction of the model, and the amount of states in the reduced model.

### 6.2.1 Results

When we look at our table we can see that the time spent in signature counting takes slightly less time than the signature creation. However, the time needed per iteration does scale with the amount of initial transitions. We can explain the difference by looking at the amount of data that is transferred for the signature calculation opposed to the signature counting. For the signature counting we read and write for each transition a single signature in the mapping phase. The reducer only counts the amount of signatures. The signature creation for each transition we read and write the whole transition, and all the incoming transitions for that transition. We can conclude that we have to transfer more than twice the amount of data.

We now want to have a more detailed look at how our algorithm scales. In order to create a clear picture we have decided to plot the initial amount of transitions against the time spend *per iteration*. We have chosen for the initial amount of transitions because it is a clear indicator of the model size. Since the total time spend on calculation increases with the amount of iterations needed for the reduction we can not do a fair comparison between the models. Therefore a clearer indication of scalability is the time needed per iteration. Figure 6.1 shows this in two figures. The first figure shows us all the models on a logarithmic scale for both axis allowing for the larger models to fit within the frame. The second figure shows us the models with less than $10^8$ transitions.

### 6.2.2 Conclusion

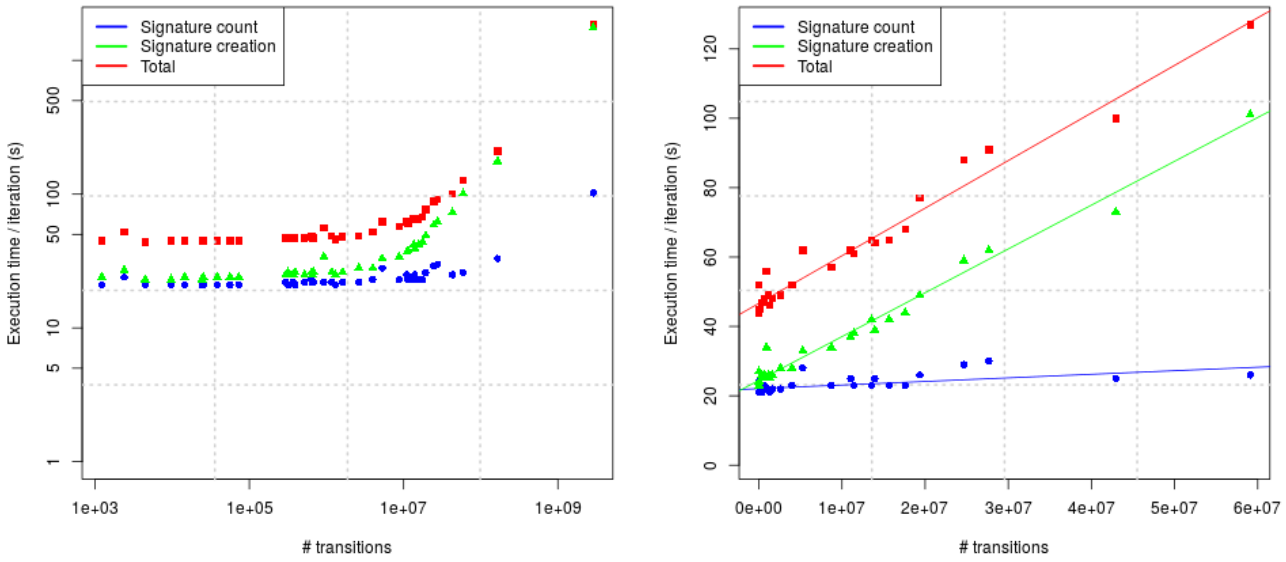
From the figures we can see that up to 10 million transitions the execution time per iteration is not increasing exponentially. We can also see that the minimum time per iteration is around 30 seconds, probably this has to do with the startup costs for each MapReduce phase. Above $10^7$ transitions the amount of transitions per second we process seems to be increasing. Our second figure shows a close-up for all the models with less than $10^8$ transitions. We indeed see a linear increase in this lower region.

| model name | transitions | BiSim | Sig | Iterations | Time | Reduced |
|---|---|---|---|---|---|---|
| cwi_1_2 | 2387 | 27 ± 3s | 24 ± 4s | 27 | 0:23:35 | 1132 |
| cwi_142_925 | 925.429 | 34 ± 5s | 22 ± 3s | 16 | 0:15:41 | 3410 |
| cwi_214_684 | 684.419 | 26 ± 1s | 22 ± 0s | 86 | 1:08:02 | 77.292 |
| cwi_2165_8723 | 8.723.465 | 34 ± 1s | 23 ± 0s | 66 | 1:03:57 | 31.906 |
| cwi_2416_17605 | 17.605.592 | 44 ± 3s | 23 ± 1s | 42 | 0:49:35 | 95.610 |
| cwi_3_14 | 14.552 | 24 ± 1s | 21 ± 0s | 61 | 0:45:33 | 62 |
| cwi_33949_165318 | 165.318.222 | 176 ± 21s | 33 ± 2s | 91 | 5:23:22 | 122.035 |
| cwi_371_641 | 641.565 | 26 ± 1s | 23 ± 1s | 73 | 0:59:25 | 33.994 |
| cwi_566_3984 | 3.984.157 | 28 ± 1s | 23 ± 0s | 16 | 0:15:04 | 15.518 |
| cwi_7838_59101 | 59.101.007 | 101 ± 11s | 26 ± 1s | 94 | 3:23:45 | 966.470 |
| lift7 | 2.875.174.785 | 1751 ± 472s | 102 ± 15s | 120 | 53:29:21 | 441.015 |
| vasy_0_1 | 1224 | 24 ± 0s | 21 ± 0s | 5 | 0:03:48 | 9 |
| vasy_10_56 | 56.156 | 24 ± 1s | 21 ± 0s | 33 | 0:24:46 | 2112 |
| vasy_11026_24660 | 24.660.513 | 59 ± 4s | 29 ± 2s | 50 | 1:15:24 | 882.341 |
| vasy_1112_5290 | 5.290.860 | 33 ± 0s | 28 ± 3s | 4 | 0:05:10 | 265 |
| vasy_116_368 | 368.569 | 25 ± 1s | 22 ± 1s | 29 | 0:22:43 | 116.456 |
| vasy_12323_27667 | 27.667.803 | 62 ± 7s | 30 ± 2s | 35 | 0:55:59 | 996.774 |
| vasy_1_4 | 4464 | 23 ± 1s | 21 ± 0s | 7 | 0:05:11 | 28 |
| vasy_157_297 | 297.000 | 25 ± 1s | 22 ± 0s | 27 | 0:21:12 | 4289 |
| vasy_164_1619 | 1.619.204 | 26 ± 1s | 22 ± 2s | 6 | 0:05:19 | 1136 |
| vasy_166_651 | 651.168 | 25 ± 1s | 22 ± 1s | 10 | 0:08:01 | 83.436 |
| vasy_18_73 | 73.043 | 24 ± 1s | 21 ± 0s | 22 | 0:16:26 | 4087 |
| vasy_25_25 | 25.216 | 24 ± 0s | 21 ± 0s | 2 | 0:01:32 | 25.217 |
| vasy_2581_11442 | 11.442.382 | 38 ± 2s | 23 ± 0s | 21 | 0:22:58 | 2.581.374 |
| vasy_386_1171 | 1.171.872 | 26 ± 1s | 22 ± 1s | 8 | 0:06:48 | 113 |
| vasy_4220_13944 | 13.944.372 | 39 ± 2s | 25 ± 1s | 39 | 0:43:25 | 1.356.477 |
| vasy_4338_15666 | 15.666.588 | 42 ± 2s | 23 ± 0s | 21 | 0:24:22 | 2.581.374 |
| vasy_52_318 | 318.126 | 26 ± 1s | 21 ± 0s | 15 | 0:11:44 | 8142 |
| vasy_574_13561 | 13.561.040 | 42 ± 3s | 23 ± 0s | 5 | 0:06:54 | 3577 |
| vasy_5_9 | 9676 | 23 ± 0s | 21 ± 1s | 5 | 0:03:45 | 145 |
| vasy_6020_19353 | 19.353.474 | 49 ± 4s | 26 ± 8s | 9 | 0:13:36 | 7168 |
| vasy_6120_11031 | 11.031.292 | 37 ± 2s | 25 ± 1s | 21 | 0:23:29 | 5199 |
| vasy_65_2621 | 2.621.480 | 28 ± 1s | 22 ± 1s | 4 | 0:03:54 | 65.536 |
| vasy_66_1302 | 1.302.664 | 25 ± 1s | 21 ± 0s | 4 | 0:03:25 | 66.929 |
| vasy_69_520 | 520.633 | 25 ± 0s | 22 ± 1s | 12 | 0:09:27 | 69.754 |
| vasy_720_390 | 390.999 | 26 ± 0s | 21 ± 1s | 5 | 0:04:00 | 3292 |
| vasy_8082_42933 | 42.933.110 | 73 ± 8s | 25 ± 1s | 10 | 0:18:05 | 408 |
| vasy_8_24 | 24.411 | 23 ± 1s | 21 ± 1s | 14 | 0:10:25 | 416 |
| vasy_83_325 | 325.584 | 25 ± 1s | 21 ± 1s | 10 | 0:07:50 | 83.436 |
| vasy_8_38 | 38.424 | 24 ± 1s | 21 ± 1s | 5 | 0:03:48 | 219 |

Table 6.1: Results for the strong bisimulation

## 6.3 Branching bisimulation

We ran our branching bisimulation algorithm for the same models as for the strong bisimulation. We sorted the models based on the iterations needed for the strong bisimulation. In table 6.2 the results for these runs can be viewed. Each row starts similar to the set-up in table 6.1, with the model name and the amount of transitions for the given model, we than have the averaged time for each "LABEL"-action (see figure 4.2). The column "tau" contains the average time over all the tau-steps, and the column "build" is the time taken for actual signature generation. The last three columns show the total amount of iterations used for the calculation, the cumulative time for the reduction of the model, and the amount of states in the reduced model.




(a) Logarithmic plot

(b) Linear plot (under $10^8$)

Figure 6.1: Iteration-times vs state space for Strong bisimulation

### 6.3.1 Results

When we look at our table we can see result quite similar to that shown for strong bisimulation. In general the "label"-step takes the most time, this is when the signature sets are created and a lot of tuples are passed. During the "tau"-steps we have a large signature set that slowly grows. This set is not trivially small, so we infer that the time spent is in part because of the traversal of and read/write-operations on this large set. The "build"-step has to read this large signature set, but only has to write one hash for each set, which might explain the quicker iterations by the "build"-step. Overall time is significantly higher than the strong bisimulation implementation, which is not surprising given the addition of several tau-steps. Using tables 6.1,6.2 and 8.1 we can focus on few models that have zero $\tau$-transitions. In theory given the absence of $\tau$-transitions, the strong and branching implementation should give the same result. Moreover, since the branching implementation always executes one (unnecessary) $\tau$-step we should expect the branching implementation to take twice as long as the strong implementation. If we look at models `vasy_0_1`, `vasy_1112_5290`,`vasy_25_25`,`vasy_574_13561`, and `vasy_65_2621` we can indeed see that there is a factor 2 between the total execution time for branching and strong bisimulation for these models.

### 6.3.2 Conclusion

From the figures we can see that for branching bisimulation up to 1 million transitions the execution time per iteration a linear fit can be made. We can also see that the minimum time per iteration is still around 30 seconds. Since this was also the case with the strong bisimulation reduction we are strengthened in our conjecture this has to do with the startup costs for each MapReduce Job. Above $10^6$ transitions the amount of transitions per second we process seems to be increasing. Our second figure shows a close-up for all the models with less than $10^7$ transitions. We indeed see a linear increase in this lower region.

## 6.4 Additional experiments

To check the speedup using a single file in sequential file-format vs the Aldebaran file-format we ran the lift5 model (`cwi_2165_8723`) from both formats for comparison. We found that

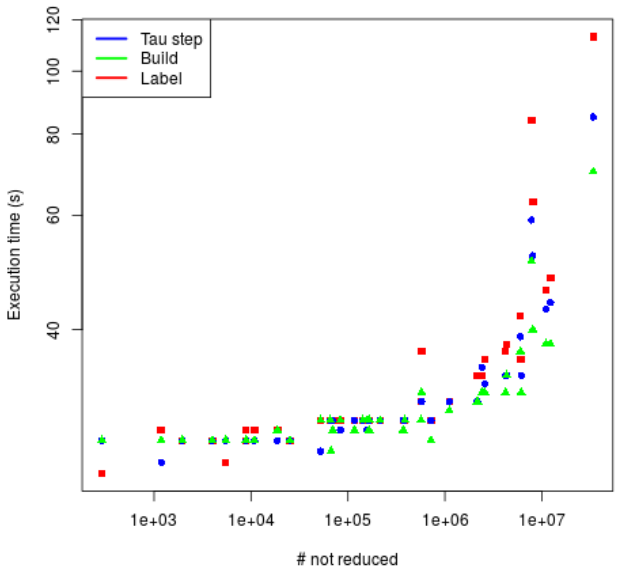
| model name | transitions | calc | tau | build | Iterations (tau) | Time | Reduced |
|---|---|---|---|---|---|---|---|
| cwi_1_2 | 2387 | 27 ± 2s | 27 ± 2s | 27 ± 2s | 8(128) | 1:07:38 | 67 |
| cwi_142_925 | 925.429 | 29 ± 1s | 29 ± 2s | 29 ± 2s | 6(30) | 0:22:58 | 23 |
| cwi_214_684 | 684.419 | 29 ± 2s | 29 ± 2s | 29 ± 1s | 18(840) | 7:00:02 | 478 |
| cwi_2165_8723 | 8.723.465 | 34 ± 2s | 31 ± 2s | 31 ± 3s | 14(696) | 6:20:39 | 4256 |
| cwi_2416_17605 | 17.605.592 | 34 ± 0s | 35 ± 3s | 32 ± 2s | 2(51) | 0:34:22 | 730 |
| cwi_3_14 | 14.552 | 27 ± 0s | 27 ± 2s | 27 ± 0s | 2(122) | 0:57:08 | 2 |
| cwi_33949_165318 | 165.318.222 | 113 ± 3s | 85 ± 5s | 70 ± 3s | 16(1080) | 26:39:44 | 12.463 |
| cwi_371_641 | 641.565 | 29 ± 1s | 29 ± 2s | 28 ± 2s | 6(195) | 1:40:50 | 2134 |
| cwi_566_3984 | 3.984.157 | 31 ± 0s | 31 ± 2s | 29 ± 2s | 6(33) | 0:26:41 | 198 |
| cwi_7838_59101 | 59.101.007 | 84 ± 5s | 59 ± 3s | 51 ± 3s | 46(2016) | 35:06:43 | 62.031 |
| vasy_0_1 | 1224 | 24 ± 2s | 27 ± 0s | 27 ± 0s | 5(5) | 0:08:58 | 9 |
| vasy_10_56 | 56.156 | 28 ± 2s | 27 ± 2s | 27 ± 3s | 33(42) | 1:02:34 | 2112 |
| vasy_11026_24660 | 24.660.513 | 46 ± 2s | 43 ± 2s | 38 ± 2s | 44(201) | 3:52:30 | 775.618 |
| vasy_1112_5290 | 5.290.860 | 31 ± 3s | 31 ± 2s | 30 ± 2s | 4(4) | 0:09:14 | 265 |
| vasy_116_368 | 368.569 | 29 ± 2s | 29 ± 2s | 28 ± 2s | 17(406) | 3:31:49 | 22.398 |
| vasy_12323_27667 | 27.667.803 | 48 ± 2s | 44 ± 2s | 38 ± 2s | 31(149) | 2:53:32 | 876.944 |
| vasy_1_4 | 4464 | 28 ± 0s | 25 ± 3s | 27 ± 0s | 2(2) | 0:03:38 | 4 |
| vasy_157_297 | 297.000 | 29 ± 3s | 28 ± 3s | 29 ± 1s | 21(63) | 0:58:04 | 3038 |
| vasy_164_1619 | 1.619.204 | 29 ± 1s | 29 ± 2s | 28 ± 2s | 5(7) | 0:10:41 | 992 |
| vasy_166_651 | 651.168 | 29 ± 4s | 29 ± 2s | 29 ± 2s | 6(18) | 0:16:43 | 42.195 |
| vasy_18_73 | 73.043 | 28 ± 3s | 27 ± 2s | 28 ± 2s | 14(81) | 0:54:48 | 2326 |
| vasy_25_25 | 25.216 | 27 ± 0s | 27 ± 0s | 27 ± 0s | 2(2) | 0:03:42 | 25.217 |
| vasy_2581_11442 | 11.442.382 | 36 ± 2s | 33 ± 2s | 32 ± 2s | 14(56) | 0:53:17 | 704.737 |
| vasy_386_1171 | 1.171.872 | 29 ± 1s | 29 ± 1s | 29 ± 0s | 5(20) | 0:16:57 | 71 |
| vasy_4220_13944 | 13.944.372 | 37 ± 3s | 34 ± 2s | 32 ± 2s | 27(124) | 1:54:49 | 1.186.266 |
| vasy_4338_15666 | 15.666.588 | 38 ± 1s | 34 ± 2s | 34 ± 2s | 14(56) | 0:57:11 | 704.737 |
| vasy_52_318 | 318.126 | 29 ± 0s | 26 ± 3s | 29 ± 2s | 4(11) | 0:10:33 | 66 |
| vasy_574_13561 | 13.561.040 | 37 ± 3s | 31 ± 2s | 32 ± 2s | 5(5) | 0:12:19 | 3577 |
| vasy_5_9 | 9676 | 25 ± 4s | 27 ± 2s | 27 ± 0s | 5(10) | 0:11:13 | 112 |
| vasy_6020_19353 | 19.353.474 | 42 ± 0s | 39 ± 3s | 37 ± 0s | 2(12) | 0:12:51 | 256 |
| vasy_6120_11031 | 11.031.292 | 36 ± 1s | 34 ± 2s | 32 ± 3s | 16(64) | 1:03:25 | 2505 |
| vasy_65_2621 | 2.621.480 | 29 ± 2s | 29 ± 0s | 29 ± 0s | 4(4) | 0:08:25 | 65.536 |
| vasy_66_1302 | 1.302.664 | 29 ± 0s | 29 ± 2s | 26 ± 2s | 3(7) | 0:07:47 | 51.128 |
| vasy_69_520 | 520.633 | 29 ± 2s | 29 ± 2s | 28 ± 3s | 12(24) | 0:27:25 | 69.753 |
| vasy_720_390 | 390.999 | 29 ± 3s | 29 ± 4s | 27 ± 3s | 5(5) | 0:08:45 | 3292 |
| vasy_8082_42933 | 42.933.110 | 63 ± 1s | 52 ± 2s | 40 ± 2s | 6(18) | 0:30:21 | 290 |
| vasy_8_24 | 24.411 | 27 ± 2s | 27 ± 1s | 27 ± 0s | 10(62) | 0:41:29 | 170 |
| vasy_83_325 | 325.584 | 29 ± 1s | 28 ± 3s | 29 ± 2s | 6(18) | 0:16:43 | 42.195 |
| vasy_8_38 | 38.424 | 28 ± 2s | 27 ± 2s | 27 ± 0s | 5(5) | 0:08:56 | 193 |

Table 6.2: Results for the branching bisimulation

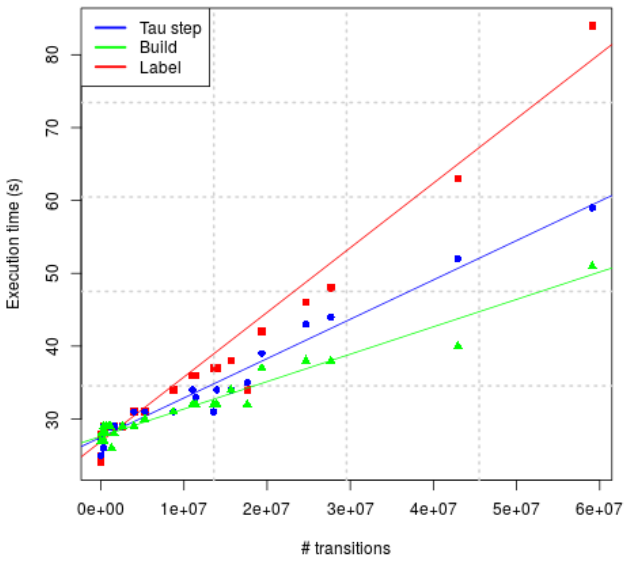
both the implementations took respectively 129 and 140 seconds for the first iteration (where the model is loaded). We attribute the lack of difference between the two implementations to the fact that both files are not offered in a split form to Hadoop, but instead are both a single big file.

We have ran a small benchmark on the speedup for using smaller key-value pairs in the signature counting for the strong bisimulation. As stated in subsection 5.1.1 the use of smaller key-value pairs led to a maximum 2x-speedup.

In figure 7.1 we have plotted the execution time for branching and strong bisimulation for the lift-models. We have done this both for our implementation and the existing distributed bisimulation reduction implemented in LTSmin[26] using a similar algorithm[8, 9].




(a) Logerithmic plot




(b) Linear plot (under $10^7$)

Figure 6.2: Iteration-times vs state space for Branching bisimulation

# Chapter 7

# Conclusion and Future Work

## 7.1 Conclusion

In Chapter 3 we introduced an algorithm for strong bisimulation reduction using MapReduce. The algorithm is based on the work by Blom and Orzan. We have studied the distributed strategy used by Blom and Orzan and created an algorithm without the need for central tables. We have introduced the use of cryptographic hashes for the calculation of partitions to circumvent the lack of communication between workers. In Chapter 4 we have introduced Branching bisimulation and created an additional algorithm to allow Branching bisimulation with MapReduce. The differences in complexity of these implementations can be seen when comparing figure 3.1 and 4.2. The resulting algorithms were benchmarked on the CTIT cluster using the VLTS-benchmark set[14]. The resulting reduced models validated in both cases against models that are reduced with the existing toolset LTSmin[26]. From all the metrics we have compiled two tables (tabel 6.1&6.2) and created two graphs showing the scaling of time versus the state space (figure 6.1&6.2).

During our experiments we saw that the execution-time for a MapReduce job takes a relatively long time. We have estimated that there is a startup cost for each job of circa 30 seconds. This means that the reduction of transition systems that need a lot of iterations can be very high. Extreme cases such as the `vasy_40_60` which take over 20.000 iterations therefore could not be benchmarked within an acceptable time-frame. Each iteration all of our data is passed over the disk. Therefore it is not unreasonable to see a factor 10-100 slow down compared to a mpi-based implementation (e.g. LTSmin). From our experiments we have concluded that the separate iteration times of our algorithm scale linearly up to $10^8$ transitions for strong bisimulation and $10^7$ for branching bisimulation. On larger models the iteration time increases exponentially, therefore we where not able to benchmark our largest model (`lift8`).

## 7.2 Comparison with existing tools

We have taken the MapReduce benchmark for the `lift5`, `lift6` and `lift7`-model and compared it with the execution time for the existing distributed implementation in LTSmin. The results are shown in figure 7.1. Note that the vertical axis is logarithmic. Here we can clearly see the price of the (slow) iteration in MapReduce, showing a factor 100 between the MapReduce implementation and an existing state-of-the-art tool. Based on this graph our conjecture is that our MapReduce implementation will not be a viable alternative for existing tools, given our current framework.

## 7.3 Hadoop and MapReduce as a Programming Paradigm for Model Checking

In two recent papers Banhos Filho and Yero and Hinkka, Lehto, and Heljanko proposed that both Hadoop and Spark[37] are indeed not suited for tasks that require a great amount of iterations[4, 23]. Therefore we feel safe to conclude that at the moment frameworks implementing

MapReduce are not a suitable platform for high-iterative tasks, even with solutions optimized for iterative tasks[36]. Since a lot of model checking algorithms have an iterative nature[2] these are not suited for MapReduce. Bisimulation reduction (e.g. strong, branching, weak) is a great example. But also algorithms using least fix point or greatest fix point calculations, such as Computational Tree Logic (CTL) or Linear Temporal Logic (LTL)[2]. A model checking technique that is a possibility is the exploration of large state spaces. However, creation of large state spaces that will not fit on a researchers machine have limited value, since we concluded that bisimulation reduction or full fledged checkers using LTL or CTL is not feasible using MapReduce. However, by severely restricting oneself, it could be possible to check properties written in First Order Logic[2] that do not require large amounts of iterations.

We conclude that MapReduce could be a valuable paradigm for model checking tools, given that:

- The overhead per iteration is very low.
- Map-tasks are able to merge input streams, allowing us to temporally "park" data.
- Additionally central data structures are allowed to prevent large data-throughputs.

## 7.4 Future Work

The availability and accessibility of commercial clusters such as Amazon EC2 is an exciting development. Researchers not possessing access to large clusters can now run very CPU- or memory-intensive tasks at a relatively low cost. With model checking we have chosen a field that uses intensive tasks and vast amounts of data. The choice for MapReduce could open possibilities for algorithms that are easily scalable and deployed on third party clusters. During our research we found that the performance of MapReduce in Hadoop is heavily bound by the overhead on separate iterations. Since most algorithms in model checking are dependent on iterative processes we can therefore advice that the use of MapReduce is promising, given that the future frameworks offer less time-expensive iterations.

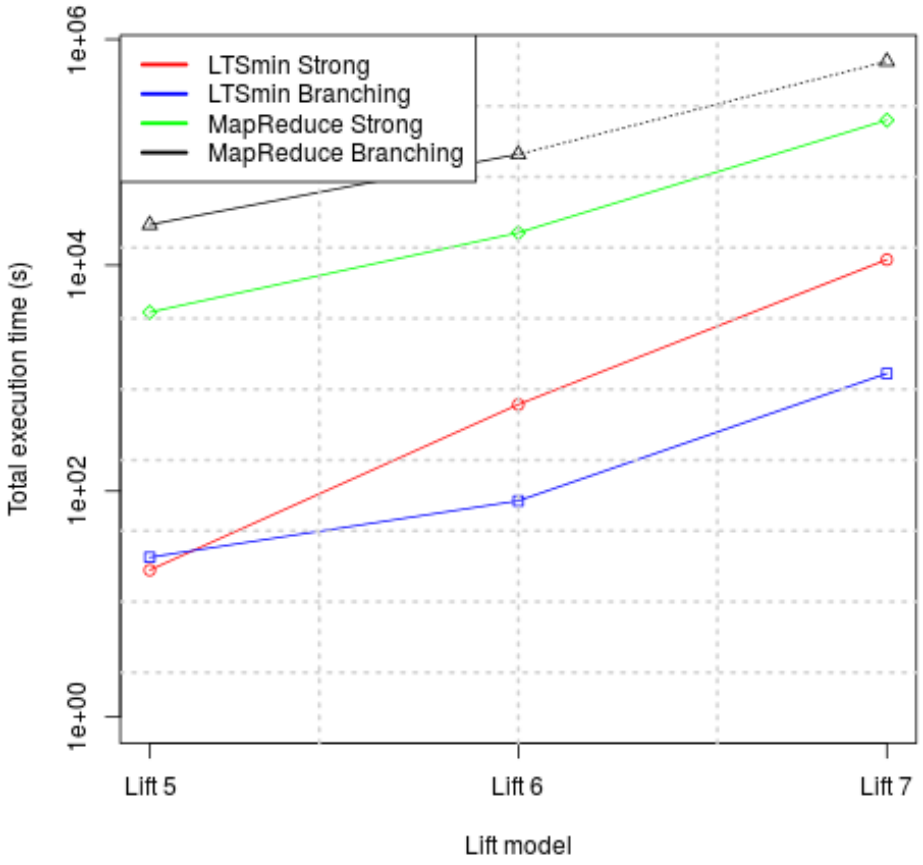



Figure 7.1: LTSmin vs MapReduce for lift models

# Chapter 8

# Appendices

| model name | states | transitions | tau-transitions | labels | branchingfactor avg [min-max] | deadlocks | livelocks | deterministic |
|---|---|---|---|---|---|---|---|---|
| cwi_1_2 | 1.952 | 2.387 | 2.215 | 26 | 1,22 [1-16] | - | - | - |
| cwi_142_925 | 142.472 | 925.429 | 862.298 | 7 | 6,50 [0-9] | X | - | - |
| cwi_214_684 | 214.202 | 684.419 | 550.611 | 5 | 3,20 [0-7] | X | X | - |
| cwi_2165_8723 | 2.165.446 | 8.723.465 | 3.830.225 | 26 | 4,03 [1-14] | - | X | - |
| cwi_2416_17605 | 2.416.632 | 17.605.592 | 17.490.904 | 15 | 7,29 [0-14] | X | X | - |
| cwi_3_14 | 3.996 | 14.552 | 14.551 | 2 | 3,64 [0-6] | X | - | - |
| cwi_33949_165318 | 33.949.609 | 165.318.222 | 74.133.306 | 31 | 4,87 [1-17] | - | X | - |
| cwi_371_641 | 371.804 | 641.565 | 445.600 | 61 | 1,73 [1-25] | - | X | - |
| cwi_566_3984 | 566.64 | 3.984.157 | 3.666.614 | 11 | 7,03 [0-10] | X | - | - |
| cwi_7838_59101 | 7.838.608 | 59.101.007 | 22.842.122 | 20 | 7,54 [3-13] | - | X | - |
| lift_7 | 501.505.138 | 2.875.174.785 | NB | 36 | NB | - | X | - |
| vasy_0_1 | 289 | 1.224 | 0 | 2 | 4,24 [4-8] | - | - | - |
| vasy_1_4 | 1.183 | 4.464 | 1.213 | 6 | 3,77 [2-5] | - | - | - |
| vasy_10_56 | 10.849 | 56.156 | 2.680 | 12 | 5,18 [4-6] | - | - | X |
| vasy_11026_24660 | 11.026.932 | 24.660.513 | 2.748.559 | 119 | 2,24 [0-13] | X | - | - |
| vasy_1112_5290 | 1.112.490 | 5.290.860 | 0 | 23 | 4,76 [3-6] | - | - | X |
| vasy_116_368 | 116.456 | 368.569 | 263.296 | 21 | 3,16 [1-8] | - | - | - |
| vasy_12323_27667 | 12.323.703 | 27.667.803 | 3.153.502 | 119 | 2,25 [0-13] | X | - | - |
| vasy_157_297 | 157.604 | 297 | 31.798 | 235 | 1,88 [0-48] | X | - | X |
| vasy_164_1619 | 164.865 | 1.619.204 | 109.910 | 37 | 9,82 [1-16] | - | - | - |
| vasy_166_651 | 166.464 | 651.168 | 91.392 | 211 | 3,91 [0-96] | X | - | - |
| vasy_18_73 | 18.746 | 73.043 | 39.217 | 17 | 3,90 [1-6] | - | - | - |
| vasy_25_25 | 25.217 | 25.216 | 0 | 25.216 | 1,00 [0-1] | X | - | X |
| vasy_2581_11442 | 2.581.374 | 11.442.382 | 2.508.518 | 223 | 4,43 [0-97] | X | - | - |
| vasy_386_1171 | 386.496 | 1.171.872 | 122.976 | 73 | 3,03 [1-38] | - | - | - |
| vasy_40_60 | 40.006 | 60.007 | 20.003 | 3 | 1,50 [1-2] | - | - | X |
| vasy_4220_13944 | 4.220.790 | 13.944.372 | 2.546.649 | 223 | 3,30 [0-97] | X | - | - |
| vasy_4338_15666 | 4.338.672 | 15.666.588 | 3.127.116 | 223 | 3,61 [0-97] | X | - | - |
| vasy_5_9 | 5.486 | 9.676 | 2.094 | 31 | 1,76 [0-6] | X | - | - |
| vasy_52_318 | 52.268 | 318.126 | 130.752 | 17 | 6,09 [1-17] | - | X | - |
| vasy_574_13561 | 574.057 | 13.561.040 | 0 | 141 | 23,62 [1-64] | - | - | X |
| vasy_6020_19353 | 6.020.550 | 19.353.474 | 17.526.144 | 511 | 3,21 [2-260] | - | X | - |
| vasy_6120_11031 | 6.120.718 | 11.031.292 | 3.152.976 | 125 | 1,80 [0-16] | X | - | - |
| vasy_65_2621 | 65.537 | 2.621.480 | 0 | 72 | 40,00 [40-40] | - | - | X |
| vasy_66_1302 | 66.929 | 1.302.664 | 117.866 | 81 | 19,46 [2-42] | - | - | - |
| vasy_69_520 | 69.754 | 520.633 | 1 | 135 | 7,46 [0-35] | X | - | - |
| vasy_720_390 | 720.247 | 390.999 | 1 | 49 | 0,54 [0-39] | X | - | X |
| vasy_8_24 | 8.879 | 24.411 | 8.534 | 11 | 2,75 [1-5] | - | - | - |
| vasy_8_38 | 8.921 | 38.424 | 2.916 | 81 | 4,31 [0-10] | X | - | X |
| vasy_8082_42933 | 8.082.905 | 42.933.110 | 2.535.944 | 211 | 5,31 [0-48] | X | - | X |
| vasy_83_325 | 83.436 | 325.584 | 45.696 | 211 | 3,90 [0-96] | X | - | - |

Table 8.1: Models used during experimentation [14]

# Bibliography


[1] Rehan Abdul Aziz. *Distributed Model Checking Using Hadoop*. Tech. rep. Department of Computer Science, Aalto University, Finland, 2010.

[2] Christel Baier, Joost-Pieter Katoen, et al. *Principles of model checking*. Vol. 26202649. MIT press Cambridge, 2008.

[3] WW Rouse Ball. "Mathematical recreations and essays". In: *Bull. Amer. Math. Soc. 46 (1940), 211-213 DOI: http://dx. doi. org/10.1090/S0002-9904-1940-07170-8 PII* (1940), pp. 0002–9904.

[4] Francisco Sanches Banhos Filho and Eduardo Javier Huerta Yero. "Exact Vs. Approximated Diameter Calculation in Large Graphs". In: *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*. IEEE. 2016, pp. 256–263.

[5] Carlo Bellettini et al. "Distributed CTL model checking in the cloud". In: *arXiv preprint arXiv:1310.6670* (2013).

[6] Carlo Bellettini et al. "Mardigras: Simplified building of reachability graphs on large clusters". In: *Reachability Problems* (2013), pp. 83–95.

[7] Stefan Blom and Simona Orzan. "Distributed branching bisimulation reduction of state spaces". In: *Electronic Notes in Theoretical Computer Science* 89.1 (2003), pp. 99–113.

[8] Stefan Blom and Simona Orzan. "Distributed state space minimization". In: *International Journal on Software Tools for Technology Transfer* 7.3 (2005), pp. 280–291.

[9] Stefan Blom and Jaco van de Pol. "Distributed branching bisimulation minimization by inductive signatures". In: *arXiv preprint arXiv:0912.2550* (2009).

[10] Stefan Blom, Jaco van de Pol, and Michael Weber. "LTSmin: Distributed and symbolic reachability". In: *Computer Aided Verification*. Springer Berlin Heidelberg. 2010, pp. 354–359.

[11] Dhruba Borthakur. "The hadoop distributed file system: Architecture and design". In: *Hadoop Project Website* 11.2007 (2007), p. 21.

[12] Michael C. Browne, Edmund M. Clarke, and Orna Grümberg. "Characterizing finite Kripke structures in propositional temporal logic". In: *Theoretical Computer Science* 59.1-2 (1988), pp. 115–131.

[13] Sjoerd Cranen et al. "An overview of the mCRL2 toolset and its recent advances". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2013, pp. 199–213.

[14] INRIA/VASY CWI/SEN2. *The VLTS benchmark*. URL: `http://cadp.inria.fr/resources/vlts/`.

[15] Rocco De Nicola and Frits Vaandrager. "Three logics for branching bisimulation". In: *Journal of the ACM (JACM)* 42.2 (1995), pp. 458–487.

[16] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[17] Tom van Dijk and Jaco van de Pol. "Sylvan: Multi-core decision diagrams". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2015, pp. 677–691.

[18] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google file system". In: *ACM SIGOPS operating systems review*. Vol. 37. 5. ACM. 2003, pp. 29–43.

[19] Jan Friso Groote and Frits Vaandrager. "An efficient algorithm for branching bisimulation and stuttering equivalence". In: *International Colloquium on Automata, Languages, and Programming*. Springer. 1990, pp. 626–638.

[20] Jan Friso Groote and Anton Wijs. "An O (m\ log n) Algorithm for Stuttering Equivalence and Branching Bisimulation". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2016, pp. 607–624.

[21] Thilina Gunarathne et al. "MapReduce in the Clouds for Science". In: *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*. IEEE. 2010, pp. 565–572.

[22] Feng Guo et al. "Ctl model checking algorithm using mapreduce". In: *Emerging Technologies for Information Systems, Computing, and Management* (2013), pp. 341–348.

[23] Markku Hinkka, Teemu Lehto, and Keijo Heljanko. "Assessing Big Data SQL Frameworks for Analyzing Event Logs". In: *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*. IEEE. 2016, pp. 101–108.

[24] Robert Kabler, Yannis E Ioannidis, and Michael J Carey. "Performance evaluation of algorithms for transitive closure". In: *Information systems* 17.5 (1992), pp. 415–441.

[25] Paris C Kanellakis and Scott A Smolka. "CCS expressions, finite state processes, and three problems of equivalence". In: *Proceedings of the second annual ACM symposium on Principles of distributed computing*. ACM. 1983, pp. 228–240.

[26] Gijs Kant et al. "LTSmin: High-Performance, Language-Independent Model Checking". In: (2015).

[27] Yongming Luo et al. "I/O-efficient algorithms for localized bisimulation partition construction and maintenance on massive graphs". In: *CoRR, abs/1210.0748* (2012).

[28] Katja Malvoni and Josip Knezovic. "Are your passwords safe: Energy-efficient bcrypt cracking with low-cost parallel hardware". In: *8th USENIX Workshop on Offensive Technologies (WOOT 14)*. 2014.

[29] Robin Milner. *Communication and concurrency*. Vol. 84. Prentice hall New York etc., 1989.

[30] Brian Pratt et al. "MR-tandem: parallel X! tandem using hadoop MapReduce on amazon Web services". In: *Bioinformatics* 28.1 (2012), pp. 136–137.

[31] Lawrence W Reed and Milton Friedman. *I, Pencil: My Family Tree as Told to Leonard E. Read*. Foundation for Economic Education, 2008.

[32] Alexander Schätzle et al. "Large-scale bisimulation of RDF graphs". In: *Proceedings of the Fifth Workshop on Semantic Web Information Management*. ACM. 2013, p. 1.

[33] Rob J Van Glabbeek and W Peter Weijland. "Branching time and abstraction in bisimulation semantics". In: *Journal of the ACM (JACM)* 43.3 (1996), pp. 555–600.

[34] Tom White. *Hadoop: The definitive guide.* " O'Reilly Media, Inc.", 2012.

[35] Anton Wijs and Dragan Bošnački. "Many-core on-the-fly model checking of safety properties using GPUs". In: *International Journal on Software Tools for Technology Transfer* 18.2 (2016), pp. 169–185.

[36] Min Yoon et al. "Performance analysis of MapReduce-based distributed systems for iterative data processing applications". In: *Mobile, Ubiquitous, and Intelligent Computing*. Springer, 2014, pp. 293–299.

[37] Matei Zaharia et al. "Spark: cluster computing with working sets." In: *HotCloud* 10 (2010), pp. 10–10.

[38] Jerry Zhao and Jelena Pjesivac-Grbovic. "MapReduce: The programming model and practice". In: *Tutorials of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. 2009, p. 105.