Graph algorithms to determine sufficient connectivity of collaborative autonomous systems

Master's thesis

Master's thesis in Computer science and engineering

Edvin Alestig
Max Hvid-Hansen

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2024

Master's thesis 2024

Graph algorithms to determine sufficient connectivity of collaborative autonomous systems

Master's thesis

Edvin Alestig
Max Hvid-Hansen

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Graph algorithms to determine sufficient connectivity of collaborative autonomous systems
Master's thesis
Edvin Alestig & Max Hvid-Hansen

© Edvin Alestig & Max Hvid-Hansen, 2024.

Supervisor: Yehia Abd Alrahman, Department of Computer Science and Engineering
Examiner: Carl-Johan Seger, Department of Computer Science and Engineering

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2024

Graph algorithms to determine sufficient connectivity of collaborative autonomous systems
Master's thesis
Edvin Alestig & Max Hvid-Hansen
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Systems of interacting agents such as robots are becoming increasingly common. The problem is that the available resources are often limited and there is a need for agents to interact so that they accomplish their mission. This thesis is a part of the SynTM project which aims at tackling this problem by producing cooperative agents with minimal interactions. The idea is to go from a high-level specification of some agents' joint mission to an implementation of a loosely coupled set of cooperative agents with minimal interaction. Our contribution consists of designing algorithms for the reduction of the required interactions among the agents while still fulfilling the original specification. An algorithm using a technique named $\mathcal{E}$-cooperative bisimulation, was proposed in the SynTM project as a proof of concept. An implementation of this algorithm features poor performance due the high computational complexity of the algorithm both in space and time. We set out to create algorithms with better performance so we propose and implement new algorithms followed by doing computational complexity analysis. The algorithms are compared using both their practical performance and complexity. Our results show that the first proposed algorithm features a time complexity one degree less than the existing algorithm together with a large speed increase, especially for large problems. The second proposed algorithm features the same complexity as the first, with equal or slightly better performance except for very large problems.

Keywords: Bisimulation, graph theory, autonomous system, computer science, algorithm, project, thesis.

Acknowledgements

We would like to thank our supervisor, Dr. Yehia Abd Alrahman, for his help with this thesis. It has been invaluable.

Edvin Alestig & Max Hvid-Hansen, Gothenburg, 2024-06-13

Contents

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**

2 Background **7**

2.1 $\mathcal{E}$-Cooperative Bisimulation 7

2.2 Solving Classic Bisimulation Problems Using Partition Refinement . . 16

2.2.1 The Kanellakis-Smolka Algorithm 17

2.2.2 The Paige-Tarjan Algorithm 19

3 Methods **21**

3.1 Algorithmic Solutions for $\mathcal{E}$-Cooperative Bisimulation 21

3.2 Study of Computational Complexity 22

3.3 Performance Evaluation 23

4 Results **25**

4.1 Applying $\mathcal{E}$-Cooperative Bisimulation 25

4.2 The Existing Proof of Concept Algorithm 25

4.2.1 Computational Complexity 27

4.2.2 Alternative Version 29

4.2.3 Performance 29

4.3 Improved Algorithm I 29

4.3.1 Implementation 30

4.3.2 Computational Complexity 32

4.3.3 Performance 35

4.4 Improved Algorithm II 36

4.4.1 Implementation 36

4.4.2 Computational Complexity 38

4.4.3 Performance 39

5 Discussion and Conclusion **45**

5.1 Discussion 45

5.1.1 Selection of Specifications 45

5.1.2 Performance 45

5.1.3 Computational Complexity 48
5.1.4 Potential Improvements and Future Work 49
5.2 Conclusion 50

Bibliography **51**

A Specifications Used in Tests **I**
A.1 LTL4simple I
A.2 AMBADecompArb3 II
A.3 AMBADPA10 III
A.4 LilyDemoV18 IV
A.5 LTL3Arb V
A.6 Priority3Arb VI
A.7 LTL2+3Arb VII
A.8 Priority2+3Arb VIII
A.9 Comb3Arb IX

List of Figures

1.1 Central model T 3
1.2 System T_1 3
1.3 System T_2 4
1.4 Reduced system T_1 (solution) 5

2.1 Central system T 12
2.2 System T_1 12
2.3 System T_2 13
2.4 Pseudocode of the parallel proof of concept algorithm 13
2.5 Minimised system T_1 (solution) 15
2.6 Example transition system 18
2.7 The reduced bisimilar system 19
2.8 Diagram of a three-way split 20

4.1 Pseudocode of the proof of concept algorithm using relations running in parallel 26
4.2 DAG of the execution of the proof of concept algorithm 28
4.3 Pseudocode of *Alg I* 31
4.4 DAG of the execution of *Alg I* 33
4.5 DAG of the execution of 'Run queue' 34
4.6 Pseudocode of *Alg II* 37
4.7 DAG of the execution of *Alg II* 40
4.8 DAG of the execution of 'Preprocess' 41
4.9 DAG of the execution of 'Refinement' 42

5.1 Diagram of the execution time for each algorithm for a selection of the tests 47
5.2 A log-log diagram of the execution time for each algorithm for all tests 47
5.3 Logarithmic diagram of the execution time for *Alg I* 49

List of Tables

4.1 Performance of the proof of concept algorithm 29
4.2 Performance of *Alg I* 35
4.3 Performance comparison between *Alg I* and proof of concept 35
4.4 Performance of *Alg II* 43
4.5 Performance comparison between *Alg I*, *Alg II* and the proof of concept algorithm 43

5.1 Performance comparison between *Alg I*, *Alg II* and the proof of concept algorithm 46

1

Introduction

Distributed systems are becoming more and more common. One such example is warehouse robots moving objects around efficiently. These systems rely on minimal communication and efficient algorithms for distribution and are usually modelled by transition systems or more generally graphs. The focus of our thesis is on systems of autonomous agents (e.g., robots) with joint goals. That is, autonomous agents that are required to communicate and collaborate to achieve joint goals. The main difficulty in designing these systems is that joint goals cannot be expressed in terms of the knowledge of individual agents, and thus message exchange and coordination is inevitable.

There are different techniques to develop such systems. One way is to develop individual agents and decide how they must interact, and another is to specify the joint behaviour of the whole system and use it to automatically generate matching individual behaviours. This thesis is part of a VR (Vetenskapsrådet) project, called SynTM, led by Dr. Yehia Abd Alrahman and focuses on the latter method.

This problem was previously considered undecidable, but thanks to a reformulation of the problem, that obstacle has been removed [1]. The new approach in SynTM consists of specifying the joint behaviour of the system using declarative logical formulas that are later used to synthesise a central model (or a graph) that is correct with respect to the specification. In other words, the approach tells a computer *what* we want to achieve in the form of declarative specifications and the computer decides if this is possible (or realisable). If it is possible, the computer has to generate a central model that invariably satisfies the specification. This model says *how* the agents should behave.

Individual agents cannot use the central model as it is, so for it to be useful, we want to decompose it for each agent so that their joint behaviour satisfies the specification. This can be done using a simple but naïve method, often resulting in unnecessary communication and interactions. Namely, by duplicating and projecting the original central model into a set of individual ones synchronising on every step. The goal of this thesis is to reduce such communication to a minimal level while still satisfying the original specification.

To achieve this goal, a new technique called $\mathcal{E}$-cooperative bisimulation was proposed as part of the SynTM project. This is a major extension of state of the art bisimulation theories. The idea is to consider the simple decomposition method of an

individual agent and minimise it with respect to the rest of the system, thus making sure no errors are introduced such as necessary communication being removed.

$\mathcal{E}$-cooperative bisimulation uses specifications defined using linear temporal logic (LTL) which is later converted into a deterministic Transition System (TS) or a specialised directed graph as will be explained later. This process is described in detail in [1]. The resulting TS describes the behaviour of all agents as a single unit, and must be decomposed for each agent.

To make it easier to understand the theory, we will use a running example throughout the thesis to illustrate the concept of $\mathcal{E}$-cooperative bisimulation. The example is quite simple as to help guide the reader through the most important parts.

Running Example (Step 1):

Consider the transition system in Fig. 1.1 which represents a central model T implementing some specification about two interacting agents T_1 and T_2. A transition system is a directed graph where each node label represents the current state of the system and each transition label represents communication with the rest of the world. Moreover, each state is decorated with a set of channels Y that the system listens to. A system can gain/lose access to a channel by changing state. The label of the state itself is decomposed into two parts: the last channel used in communication and the current output of state of the system, e.g., $\perp/00$ where $\perp$ means there was no communication or it was initiated by another system; while 00 represents the current state of the system represented by some local variables. Due to the way the TS is created, a state cannot have incoming transitions on different channels as it cannot have more than one last used channel in its label [1]. The TS must be deterministic, i.e., each state has at most one outgoing transition on each channel.

All states in the example have the same output values. The only effect different values would have is changing the labels of the states which could lead to fewer equivalent states. It is also much easier to disprove equivalence of states with different labels, thus not showing the core part of the algorithm. In order to show how a simple decomposition can be done, we choose a small example system with similar labels.

In this example we consider the set of channels the system listens to as $Y = \{a, b, c, d, e, f, g\}$ and the outputs to be the assignment to two local variables $O = \{o_1, o_2\}$.

A typical question is that how to decompose the central model T into two individual models for the robots T_1 and T_2, such that when they run in parallel they still satisfy the same specification.

Each agent is given an interface. An interface (Int) contains a set of channels that the agent can initiate communication on, and its outputs/state.

$$\begin{array}{llll} \textsc{Int}_1 = \langle Y_1, O_1 \rangle & \text{where} & Y_1 = \{e, f, g\}, & O_1 = \{o_1\} \\ \textsc{Int}_2 = \langle Y_2, O_2 \rangle & \text{where} & Y_2 = \{a, b, c, d\}, & O_2 = \{o_2\} \end{array}$$

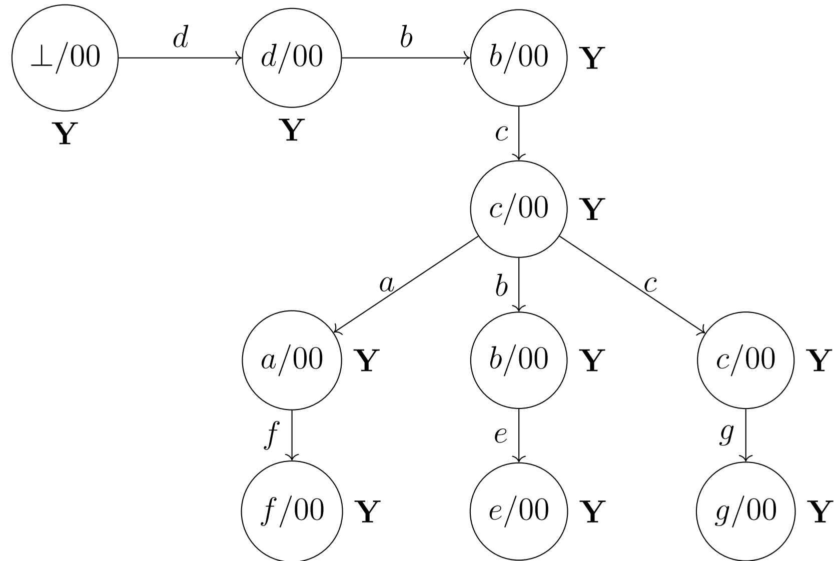

Figure 1.1: Central model T

By splitting T (Fig. 1.1) based on the interfaces $\textsc{Int}_1$ and $\textsc{Int}_2$, we can create the transition systems T_1 (Fig. 1.2) and T_2 (Fig. 1.3). This process uses a naïve method described in detail in [1]. Simply put, the output variables not in the system's interface are removed and the last used channels are changed to $\perp$ if the original channel is not included in its interface (i.e., the agent cannot initiate communication on it). The set of channels a state can listen to is replaced by a set containing only the channels with transitions from that state.

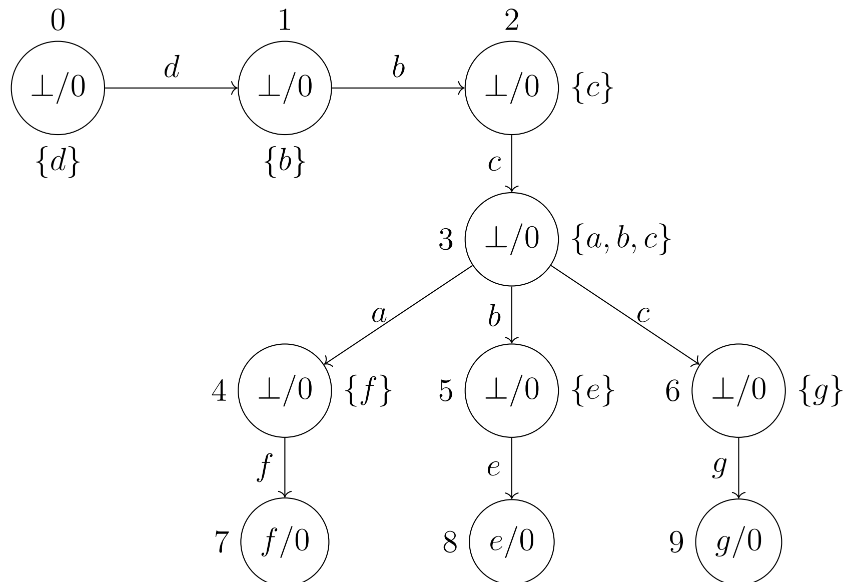

Figure 1.2: System T_1

If we apply the SynTM approach, namely by reducing one agent model with respect to another, we get a less coupled model with minimal interaction. For instance, if we reduce T_1 with respect to T_2. That is, by considering T_2 as a parameter, we get the

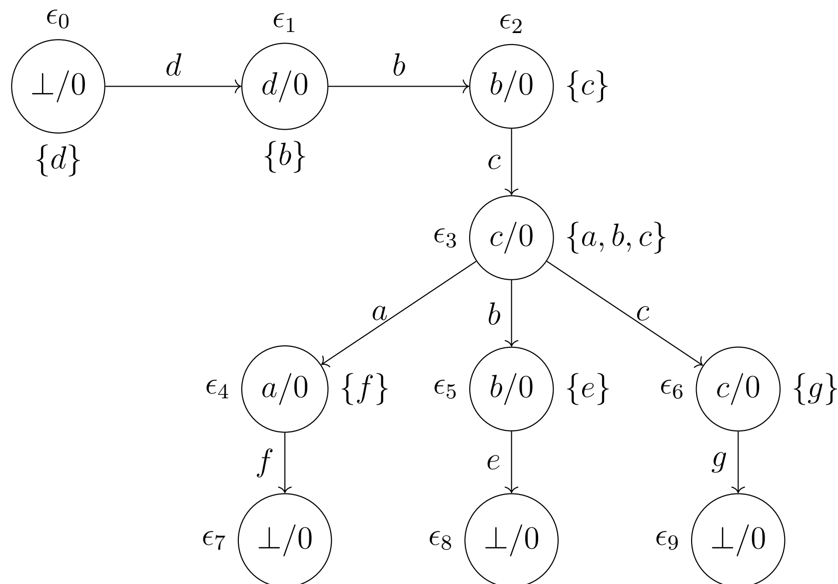

Figure 1.3: System T_2

transition system shown in Fig. 1.4. As it can be seen, the resulting TS requires less communication. In fact, the first two edges (labelled d and b) have been removed and were thus deemed unnecessary. A detailed description of how this was achieved can be found in Section 2.1.

We use the proposed $\mathcal{E}$-cooperative bisimulation technique as a starting point for our thesis work. A proof of concept algorithm was created as a part of the SynTM project which we initially studied the computational complexity of. Based on the results, we designed our own optimised bisimulation algorithm and compared it to the existing one. We considered both sequential and parallel algorithms to solve this problem and studied their computational complexities. The complexity is essential to understand how our algorithms perform in general, but in practice this might give a skewed perception of the performance considering different instances with varying size and the fact that unlikely edge cases can worsen the complexity while being performant in the average case. One such example is the problem of solving parity games, where an exponential algorithm performs better than quasi-polynomial alternatives despite the inferior complexity [2].

Computational complexity only informs about the asymptotic complexity of the algorithm and is more useful when considering large instances. However, in practice there can be different algorithms that are efficient for a specific scale. To get a good understanding of the practical performances, we evaluated the proposed algorithms in practice by studying their performance considering different input scales. One of the challenges of the latter goal is that we did not have a test bench to benchmark the algorithms and thus some time was dedicated to design such a test bench based on our observations during the development.

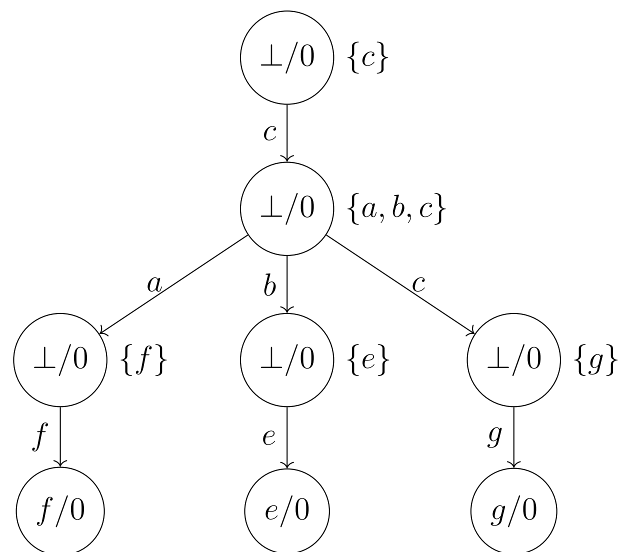

Figure 1.4: Reduced system T_1 (solution)

2

Background

This chapter covers the material needed to understand this thesis. We introduce concepts such as bisimulation as well as important definitions.

2.1 $\mathcal{E}$-Cooperative Bisimulation

Bisimulation is an established method in graph theory that is normally used to simplify graphs by exploiting symmetries. Simplification may lead to a reduction in the number of states in the graph, and thus facilitates more efficient analysis [3].

Two graphs are called bisimilar if they are indistinguishable from an external observer point of view. External observers can only observe the labels of the edges/ transitions in the graphs while state labels are private information. The number of states and edges in the systems may vary, but as long as the behaviour is externally indistinguishable, they are called bisimilar. This type of bisimulation is what exists in the current literature.

Traditional bisimulation techniques have been used to minimise system models, represented as transition systems or directed graphs. There, state labels are used to encode the current state of the system, and edge labels are used to encode communication capabilities. That is, a system in its current state can enable communication with other systems using specific communication links (the edge labels).

Traditional state of the art bisimulation algorithms focus on reducing the number of states in a transition system [3]. However, the focus of this thesis is rather on communication reduction. That is, we want to remove unnecessary edges (representing communication between agents). This means that existing bisimulation techniques will not work in this case, and a major adaptation is required.

Building on that idea, the SynTM project introduced $\mathcal{E}$-cooperative bisimulation which permits communication reduction by relying on parameterised algorithms. That is, given a model of the system, we need to come up with an algorithm that will decide what communication edges can be removed if this system is expected to collaborate with another system (i.e., the parameter). A simple proof of concept algorithm has been proposed, but its computational and practical complexity had not been investigated.

It is worth mentioning that the support of parameterised algorithms is a major modification to traditional bisimulation algorithms. The difficulty stems from the fact that the solution of the parameterised algorithm will feature a simultaneous solution of a set of interdependent fixed point algorithms, i.e., a joint fixed point.

We are working within the context of autonomous systems where traditional bisimulation would not take the connectivity between the agents into consideration. Since these agents often have limited resources, optimisation is of utmost importance. Connectivity between agents is costly so we would want to reduce it. The core work of our thesis is to first study the computational and practical complexity of the proof of concept algorithm and come up with efficient variants that can solve the problem in practical settings. We also provide prototype implementations for all of our proposed algorithms.

Firstly, we will formally show how to model autonomous systems (Definition 2.1.1). All of these definitions can be found in [1]. Secondly, we will present the notion of $\mathcal{E}$-cooperative bisimulation (Definition 2.1.4) and demonstrate how to apply it using our running example.

Definition 2.1.1 (Transition system [1]). A transition system (TS) T_k is $\langle S_k, s_0^k, \textsc{Int}_k, \textsc{ls}^k, L_k, \Delta_k \rangle$, where:

- S_k is the set of states of T_k and $s_0^k \in S_k$ its initial state.
- $\textsc{Int}_k = \langle Y_k, O_k \rangle$ is the interface of T_k, where
  - $Y_k \subseteq Y$ is the only set of channels that agent$_k$ can use to initiate communication, where Y is the set of channels that can be used by all agents. All other channels in $Y \setminus Y_k$ can only be used to react to communication from other agents.
  - O_k is an output (or actuation) alphabet.
- $\textsc{ls}^k : S_k \to 2^Y$ is a channel listening function such that for every state $s \in S_k$, channel $y \in Y$ we have that $\Delta_k(s, y) \neq \varnothing$ implies $y \in \textsc{ls}^k(s)$. That is, $\textsc{ls}^k$ defines (per state) the channels that T_k listens to, always including the *enabled* channels (i.e., $\Delta_k(s, y) \neq \varnothing$ for all $y \in Y_k$) in Y.
- $L_k : S_k \to 2^{Y_k} \times O_k$ is a labelling function. We use L_k to label states with recent used channels from Y_k and also the produced output. We will use $L_k^y(s) \subseteq Y_k$ and $L_k^o(s) \in O_k$ to denote the channel label of s (and correspondingly the output label of s). Note that unlike $\textsc{ls}^k$, we have that $L_k^y(s) \not\subseteq Y \setminus Y_k$, i.e., cannot contain channel labels in $Y \setminus Y_k$.
- $\Delta_k \subseteq S_k \times Y_k \times S_k$ is the transition relation of T_k, satisfying the following:
  - For every state $s \in S_k$ and every channel $y \in Y_k$, if there exists $s' \in S_k$ such that $(s, y, s') \in \Delta_k$ then $y \in L_k^y(s')$.
  - For every state $s \in S_k$, if there exists $s' \in S_k$ such that $(s, y, s') \in \Delta_k$ for some $y \in (Y \setminus Y_k)$ then $y \notin L_k^y(s')$.

The formal definition of the example system in Fig. 1.2 is $T_1 = \langle S_1, s_0^1, \textsc{Int}_1, \textsc{ls}^1, L_1, \Delta_1\rangle$ where

$S_1 = \{0, 1, \dots, 9\}$

$s_0^1 = 0$

$\textsc{Int}_1 = \langle\{e, f, g\}, \{o_1\}\rangle$

$$\textsc{ls}^1 = \left\{ \begin{array}{ll} \textsc{ls}^1(0) & = \{d\} \\ & \vdots \\ \textsc{ls}^1(9) & = \varnothing \end{array} \right\}$$

$$L_1 = \left\{ \begin{array}{ll} L_1^y(0) & = \bot \\ & \vdots \\ L_1^y(9) & = g \\ L_1^o(0) & = 0 \\ & \vdots \\ L_1^o(9) & = 0 \end{array} \right\}$$

$\Delta_1 = \{(0, d, 1), \dots, (6, g, 9)\}$

TSs can be combined to form a team as shown in Definition 2.1.2:

Definition 2.1.2 (Team semantics [1])**.** Given a set $K = \{1, \dots, n\}$ of TSs $T_k = \langle S_k, s_0^k, \textsc{Int}_k, \textsc{ls}^k, L_k, \Delta_k\rangle$ where $k \in K$, their composition is the team $T = \langle S, s_0, \textsc{Int}, \textsc{ls}, L, \Delta\rangle$ where

$S = \prod_{k\in K} S_k$ and $s_0 = (s_0^1, \dots, s_0^n)$,

$\textsc{Int} = \langle Y, O\rangle$ such that $Y = \bigcup_k Y_k$ and $O = \prod_{k\in K} O_k$,

$$\Delta = \left\{ \left(\begin{array}{c} (s_1, \dots, s_n), \\ y, \\ (s_1', \dots, s_n') \end{array} \right) \middle| \begin{array}{l} \exists k.\ y \in Y_k, (s_k, y, s_k') \in \Delta_k \text{ and } \forall j \neq k \\ (1)\ (s_j, y, s_j') \in \Delta_j \text{ and } y \in \textsc{ls}^j(s_j) \text{ or} \\ (2)\ y \notin \textsc{ls}^j(s_j) \text{ and } s_j' = s_j \end{array} \right\},$$

$\textsc{ls}((s_1, \dots, s_n)) = \bigcup_k \textsc{ls}^k(s_k)$ and

$L((s_1, \dots, s_n)) = (\bigcup_{k=1}^n L_k^y(s_k), \prod_{k=1}^n L_k^o(s_k))$.

Intuitively, multicast channels are blocking. That is, if there exists an agent$_k$ with a transition $(s_k, y, s_k') \in \Delta_k$ on channel $y \in Y_k$ (i.e., agent$_k$ is the initiator) then every other parallel agent$_j$, s.t. $j \neq k$, who listens to y in its current state, i.e., $y \in \textsc{ls}(s_j)$, must supply a matching transition $(s_j, y, s_j') \in \Delta_j$ or otherwise the sender is blocked. Other parallel agents that do not listen to y simply cannot observe the interaction, and thus cannot block it.

In the example above, T_1 (Fig. 1.2) and T_2 (Fig. 1.3) can form a team equivalent to T (Fig. 1.1) using these semantics. This shows that decomposed systems can be recomposed as a team to form the original central model.

Definition 2.1.3 (Notation [1])**.** For a transition system (TS) A, the following notation will be used:

- A can initiate communication on y:
  $(s \xrightarrow{y}_! s')$ iff $y \in Y_A$ and $(s, y, s') \in \Delta_A$, i.e., A is initiating an exchange (or communication) on y from state s.

- A can directly react to communication on y:
  $(s \xrightarrow{y}_? s')$ iff $y \notin Y_A, y \in \text{LS}^A(s), (s, y, s') \in \Delta_A$, and $L(s) \neq L(s')$, i.e., A, in state s, only reacts to an exchange on y from other agents, and updates its next state's label as a side effect.

- A reacts silently to communication on y:
  $(s \xrightarrow{y}_\tau s')$ iff $y \notin Y_A, y \in \text{LS}^A(s), (s, y, s') \in \Delta_A$, and $L(s) = L(s')$. Note the state's label did not change due to exchange on y, i.e., A, in state s, may not be required to participate in this exchange.

Recall that, by the semantics of TS in Definition 2.1.2, exchanges of the type $\xrightarrow{y}_\tau$ or $\xrightarrow{y}_?$ cannot happen without a corresponding initiator, i.e., $\xrightarrow{y}_!$.

- We use $(\xrightarrow{\tau}{}^\star_y)^{\epsilon'}_\epsilon$ to denote a sequence (possibly empty) of arbitrary silent reactions ($\xrightarrow{y'}_\tau$ for any $y' \neq y$), starting when the cooperative parameter state is ϵ and ending with ϵ'.

- We will use $s \xrightarrow{y}$ when $\Delta_A(s, y) \neq \varnothing$, i.e., s enables a y-transition, and $s \not\xrightarrow{y}$ when $\Delta_A(s, y) = \varnothing$, i.e., s does not enable y-transitions.

Definition 2.1.4 ($\mathcal{E}$-Cooperative Bisimulation [1])**.** Consider the TS $T = \langle S, S_0, \text{INT}, \text{LS}, \text{L}, \Delta \rangle$ to be minimised (reduce communication) with respect to the cooperative parameter TS $\mathcal{C} = \langle \mathcal{E}, \epsilon_0, \text{INT}_\mathcal{E}, \text{LS}^\mathcal{E}, \text{L}_\mathcal{E}, \Delta_\mathcal{E} \rangle$. An $\mathcal{E}$-Cooperative bisimulation relation $\mathcal{R}$ is a *symmetric* $\mathcal{E}$-indexed family of relations $\mathcal{R}_\epsilon \subseteq S \times S$ for $\epsilon \in \mathcal{E}$ such that whenever $(s_1, s_2) \in \mathcal{R}_\epsilon$ then:

1. $L(s_1) = L(s_2)$

   and for all $y \in Y$, we have that:

2. if $\nexists \epsilon'.\ (\epsilon, y, \epsilon') \in \Delta_\mathcal{E}$ and $y \notin Ls(\epsilon)$ then
   $s_1 \xrightarrow{y}_! s'_1$ implies $\exists s'_2.\ s_2 \xrightarrow{y}_! s'_2$ and $(s'_1, s'_2) \in \mathcal{R}_\epsilon$;

3. if $(\epsilon, y, \epsilon') \in \Delta_\mathcal{E}$ then

   (a) $s_1 \xrightarrow{y}_! s'_1$ implies $\exists s'_2.\ s_2 \xrightarrow{y}_! s'_2$ and $(s'_1, s'_2) \in \mathcal{R}_{\epsilon'}$

   (b) $y \notin (\text{LS}(s_1) \cup \text{LS}(s_2))$ implies $(s_1, s_2) \in \mathcal{R}_{\epsilon'}$

   (c) $s_1 \xrightarrow{y}_\tau s'_1$ implies $s_2 \not\xrightarrow{y}_?$ and

   $$\text{if } s_2 \xrightarrow{y}_\tau \text{ then } \exists s'_2.\ s_2 \xrightarrow{y}_\tau s'_2 \text{ and } (s'_1, s'_2) \in \mathcal{R}_{\epsilon'}$$
   $$\text{else } (s_1, s_2) \in \mathcal{R}_{\epsilon'} \text{ and } (s'_1, s_2) \in \mathcal{R}_{\epsilon'}$$

   (d) $s_1 \xrightarrow{y}_? s'_1$ implies $s_2 \not\xrightarrow{y}_\tau$ and

   $$\text{if } s_2 \xrightarrow{y}_? \text{ then } \exists s'_2.\ s_2 \xrightarrow{y}_? s'_2 \text{ and } (s'_1, s'_2) \in \mathcal{R}_{\epsilon'}$$
   $$\text{else } \exists s'_2, s''_2, \epsilon''.\ s_2 (\xrightarrow{\tau}{}^\star_y)^{\epsilon}_{\epsilon''} s''_2, s''_2 \xrightarrow{y}_? s'_2, (s_1, s''_2) \in \mathcal{R}_\epsilon \text{ and } (s'_1, s'_2) \in \mathcal{R}_{\epsilon'}$$

Two states s_1 and s_2 of T are $\mathcal{E}$-cooperative bisimilar with respect to a parameter state $\epsilon \in \mathcal{E}$, written $s_1 \sim_\epsilon s_2$, **iff** there is an $\mathcal{E}$-Cooperative bisimulation $\mathcal{R}$ such that $(s_1, s_2) \in \mathcal{R}_\epsilon$, where ϵ is a state of $\mathcal{C}$.

To put Definition 2.1.4 into words, condition 1 ensures the two states are equivalent if they have the same label, i.e., they have the same output and recently used channel. Condition 2 ensures that two states are equivalent if they can both initiate on the same channel without the participation of the parameter and they reach two equivalent states under the current parameter state. Note that in this case the parameter is not listening to the channel. Condition 3.a is similar to Condition 2, but here the parameter participates and the next states must be equivalent under the next parameter state. Condition 3.b considers the case when the parameter initiates and neither state listens, and in this case both states must be equivalent in the next parameter state. Condition 3.c considers the case when one state reacts silently to a parameter message, and in this case the other state has to match either with an exact silent move or by not listening to the channel. In both cases the resulting states must be equivalent in the next parameter state.

Condition 3.d is similar to 3.c, but it handles the actual reaction to messages from the parameter. Here, if one state can directly react to a message then the other state must either react directly or otherwise react through a sequence of silent moves followed by a matching reaction. In all cases, the reached states must be equivalent in the next parameter state.

This clearly shows the recursive nature of the definition where the equivalence checking terminates eventually when a fixed point is reached. To make it even clearer, an example application of it is presented below.

Running Example (Step 2):

Recall the system model T (Fig. 2.1) in our running example in the introduction with channels $Y = \{a, b, c, d, e, f, g\}$ and outputs $O = \{o_1, o_2\}$. Now, we want to split it for the two agents using the interfaces $\textsc{Int}_1 = \langle Y_1, O_1 \rangle$ where $Y_1 = \{e, f, g\},\ O_1 = \{o_1\}$ and $\textsc{Int}_2 = \langle Y_2, O_2 \rangle$ where $Y_2 = \{a, b, c, d\},\ O_2 = \{o_2\}$.

To provide agent-specific system models, the central TS is split with respect to the agents' interfaces using an algorithm presented in [1]. In this example, the TS T is split using the interfaces $\textsc{Int}_1$ and $\textsc{Int}_2$. The resulting system models, T_1 and T_2, contain all the states and edges in T but with the state labels changed to only concern that particular agent. Naturally, these models are not optimal and may contain unnecessary communication (edges). To optimise the system, the $\mathcal{E}$-cooperative bisimulation technique is utilised.

Consider the case where we want to minimise T_1 in Fig. 2.2 with respect to T_2 in Fig. 2.3. In that case, we need to build an indexed family of binary relations over the states of T_1. The indices here are the states of T_2, i.e., the parameter. The algorithm is shown in Fig. 2.4.

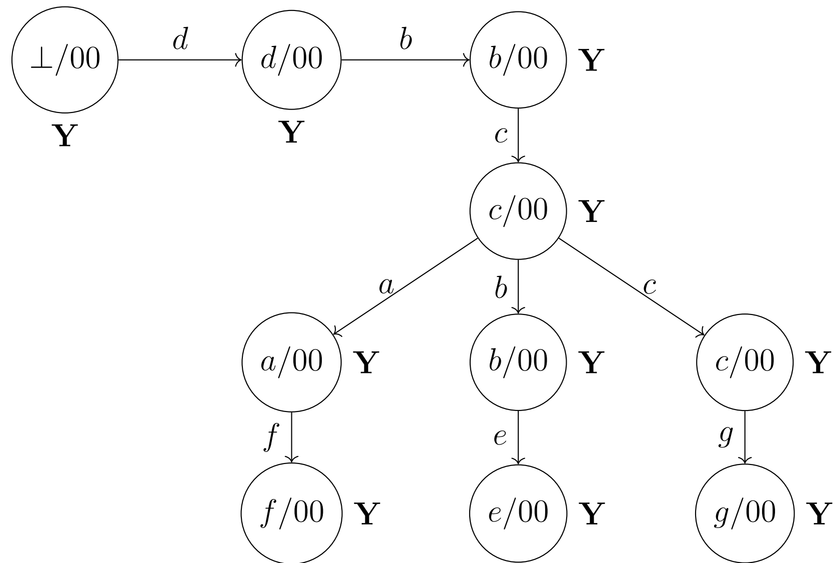

Figure 2.1: Central system T

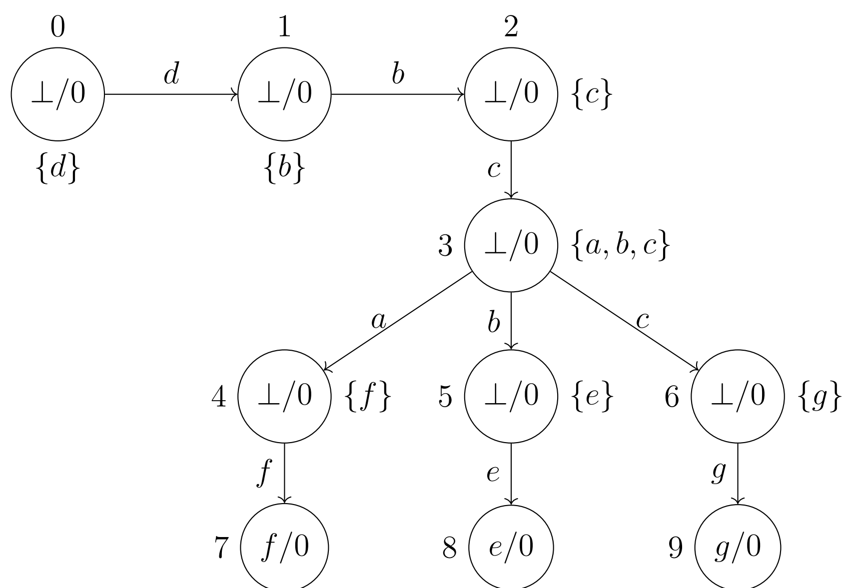

Figure 2.2: System T_1

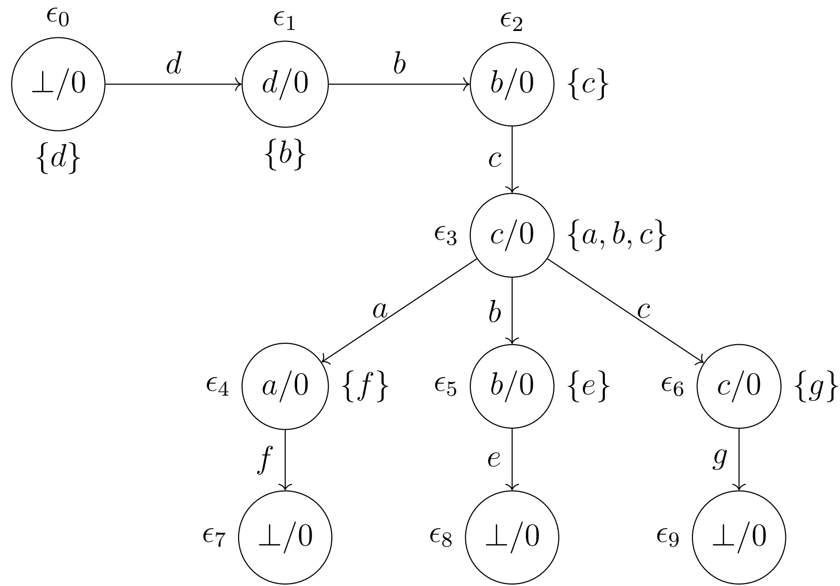

Figure 2.3: System T_2

```
1   input: T = ⟨S, s_0, INT, LS, L, Δ⟩ to be minimised with respect to the
2          parameter TS C = ⟨E, ε_0, INT_E, LS^E, L_E, Δ_E⟩
3   for all ε ∈ E, create R_ε = S × S, tmp_ε = ∅
4   while true
5     par ∀ε ∈ E
6       for all (s_1, s_2) in R_ε
7         if (s_1, s_2) not in f(R)_ε then
8           add (s_1, s_2) to tmp_ε
9         end
10      end
11    sync
12    if all tmp_ε are empty then
13      return {R_ε for each ε ∈ E}
14    end
15    for all ε in E
16      R_ε = R_ε \ tmp_ε
17    end
18  end
```

Figure 2.4: Pseudocode of the parallel proof of concept algorithm

First, we need to build the set $\{\mathcal{R}_{\epsilon_0}, \ldots, \mathcal{R}_{\epsilon_9}\}$ where each $\mathcal{R}_{\epsilon_i}$ for $0 \leq i \leq 9$ is defined over $S \times S$, containing all possible pairs of states in T_1. The reduction is done by repeatedly checking Definition 2.1.4 on each pair in each state's relation. If a pair does not satisfy all conditions, it is removed from the relation. Remember that the definition is recursive, thus the states' relations depend on each other. When all pairs in all states' relations satisfy Definition 2.1.4, i.e., a fixed point has been reached and the algorithm terminates. The starting state's (specified in the interface) relation now contains all pairs of equivalent states that can be combined to create a minimised system model. This will now be applied to the running example.

$$
\begin{aligned}
&S = \{0,1,2,3,4,5,6,7,8,9\}\\
&\text{For all } i \in [0,9]:\\
&\mathcal{R}_{\epsilon_i} = S \times S = \{(0,1),(0,2),(0,3),(0,4),(0,5),(0,6),(0,7),(0,8),(0,9),\\
&\qquad (1,2),(1,3),(1,4),(1,5),(1,6),(1,7),(1,8),(1,9),\\
&\qquad (2,3),(2,4),(2,5),(2,6),(2,7),(2,8),(2,9),\\
&\qquad \ldots\\
&\qquad (8,9)\} \cup Id \cup \mathcal{R}_{\epsilon_i}^{-1}
\end{aligned}
$$

We start with the states and relations defined above. Using condition 1 in Definition 2.1.4, we check the label of all states. States 0-6 all have the same label ($\perp/0$) while 7, 8 and 9 are different. Because equivalent states must have the same label, no pairs containing states 7, 8 and 9 satisfy all the conditions in Definition 2.1.4 and can be removed. This gives the updated relations

$$
\begin{aligned}
&\text{For all } i \in [0,9]:\\
&\quad \mathcal{R}_{\epsilon_i} = \{(0,1),(0,2),(0,3),(0,4),(0,5),(0,6),\\
&\qquad (1,2),(1,3),(1,4),(1,5),(1,6),\\
&\qquad (2,3),(2,4),(2,5),(2,6),\\
&\qquad \ldots\\
&\qquad (5,6)\} \cup Id \cup \mathcal{R}_{\epsilon_i}^{-1}.
\end{aligned}
$$

Then we check condition 2. The parameter TS (T_2) does not contain any transitions from a state to itself so the implication is always true. Condition 3 is checked on all transitions (ϵ, y, ϵ') in T_2 where y is the channel and ϵ, ϵ' are states.

All transitions have to be checked (in no particular order) in each iteration so in this example we begin with $(\epsilon_0, d, \epsilon_1)$. The implication in condition 3a is always true as $d \notin Y_1$ (T_1 cannot initiate communication on d). Condition 3b checks the channels each state is listening to. The channels listened to are denoted $\textsc{ls}(s)$ and are shown as sets below/to the right of each state in Fig. 2.2. If $d \notin \textsc{ls}(s_1) \cup \textsc{ls}(s_2)$, then $(s_1, s_2) \in \mathcal{R}_{\epsilon_0}$ must exist in $\mathcal{R}_{\epsilon_1}$. This is true for all states in T_1.

Condition 3c checks silent reactions. Only $(0,1) \in \mathcal{R}_{\epsilon_0}$ has a silent reaction on channel d. State 0 does not have a direct reaction and because state 1 does not have a silent reaction on d, we check if $d \notin \text{LS}(1)$ and $(1,1) \in \mathcal{R}_{\epsilon_1}$. This is true because $\mathcal{R}$ is reflexive (a state is always equivalent to itself).

Condition 3d checks direct reactions. No $(s_1, s_2) \in \mathcal{R}_{\epsilon_0}$ has a direct reaction on channel d so the implication is true. As a result, we were not able to remove any more pairs from $\mathcal{R}_{\epsilon_0}$ using the transition $(\epsilon_0, d, \epsilon_1)$ in this iteration. Because the relations depend on other states' relations, this can change in later iterations.

If we move to the transition $(\epsilon_4, f, \epsilon_7)$ in the parameter T_2, we can look at the pair $(4,5) \in \mathcal{R}_{\epsilon_4}$. When checking condition 3a, we find $4 \xrightarrow{f}_! 7$. This implies that there must exist an initiating transition on channel f from state 5 which is not the case. Thus we can remove the pair $(4,5)$ from $\mathcal{R}_{\epsilon_4}$. The same applies to the pair $(4,6)$. The pair $(5,6)$ can be removed from $\mathcal{R}_{\epsilon_5}$ using the same logic. Because of the symmetry of $\mathcal{R}$, the symmetrical pairs $(5,4), (6,4), (6,5)$ can be removed as well.

These steps will be performed iteratively for all transitions (ϵ, y, ϵ') in T_2 until no further changes can be made, i.e., a fixed point has been reached. The changes made in $\mathcal{R}_{\epsilon_4}$ and $\mathcal{R}_{\epsilon_5}$ will bubble up to the top due to the recursive definition, where we will end up with the relation $\mathcal{R}_{\epsilon_0} = \{(0,1), (0,2), (1,2)\} \cup Id \cup \mathcal{R}_{\epsilon_0}^{-1}$. We can conclude that the states 0, 1 and 2 are equivalent and can be combined in the minimised TS, see Fig. 2.5. The system has been reduced and requires less communication than before which was the objective.

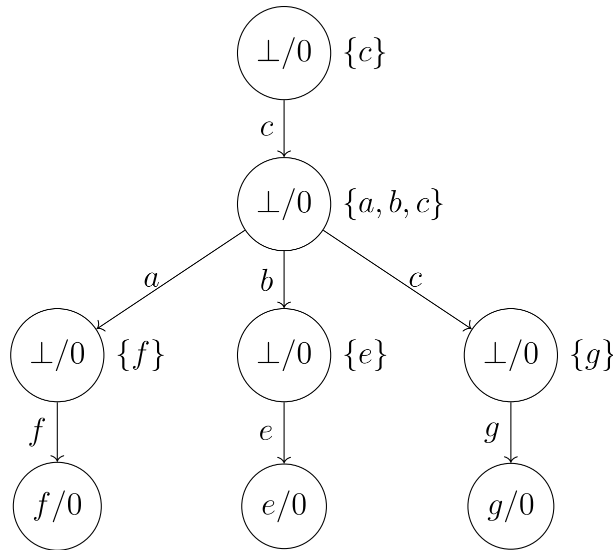

Figure 2.5: Minimised system T_1 (solution)

Notice that the key difficulty here is that we are solving a simultaneous set of interdependent fixed point algorithms, one for each $\mathcal{R}_{\epsilon_i}$. That is, we need a global termination criterion for our algorithm. Also, it gets complicated when the parameter does not move. In that case, we need to evaluate the definition at two different stages in the same algorithm which is computationally expensive.

This method is quite simple but its performance is lacking. A more efficient method to calculate $\mathcal{E}$-cooperative bisimulation is to use partition refinement algorithms instead of working with relations. When using relations, all equivalent states are stored as pairs. For instance, if states 1, 2 and 3 are equivalent as well as 4 and 5 being equivalent, the relation would be equal to {(1,1), (1,2), (1,3), (2,1), (2,2), (2,3), (3,1), (3,2), (3,3), (4,4), (4,5), (5,4), (5,5)}. It is an inefficient structure where a set of n equivalent states needs n^2 pairs in the relation.

A more efficient structure is a partition of all states, i.e., a set of sets containing equivalent states. The example relation above would be equal to the partition $\{\{1, 2, 3\}, \{4, 5\}\}$. The sets in the partition are called blocks. Any state not equivalent to any other forms a singleton block containing only that state. This structure always includes each state once and once only, thus replacing a structure of size $|S \times S|$ with one of size $|S|$. It also leads to much fewer calculations necessary to refine a graph using bisimulation which is why the most common existing bisimulation algorithms utilise it. We will also use this structure in all algorithms proposed in this thesis.

Unfortunately, existing partition refinement algorithms performing traditional bisimulation cannot be directly applied as they have different objectives and cannot solve parametric problems. Classic bisimulation takes in one system to be minimised and by repeated partitioning, a fixed point is reached for that system. $\mathcal{E}$-cooperative bisimulation on the other hand, in addition to the system to be minimised, also takes in a parameter system. Each state in the parameter has its own partition of the system to be minimised. The objective is to find a joint fixed point, i.e., a point when all partitions have reached fixed points. The partitions are interdependent, i.e., they depend on each other in a recursive fashion. Thus, instead of having to find one fixed point, we need to find as many fixed points as there are states in the parameter system. Because of the partitions' dependency on each other, it is not as straightforward as finding a fixed point in the traditional setting with only a singular independent system.

Because of these different objectives and capabilities, we need to create new adaptations of existing classic bisimulation algorithms that executes $\mathcal{E}$-cooperative bisimulation. The next section introduces some existing state of the art bisimulation algorithms that could be adapted for our purposes.

2.2 Solving Classic Bisimulation Problems Using Partition Refinement

There are two famous algorithms for solving classic bisimulation problems using partition refinement, the Kanellakis-Smolka and the Paige-Tarjan algorithms. They are both able to reduce the amount of states by partitioning the partition representation of the original graph until they reach fixed points. A fixed point is reached when a partition cannot be refined further.

2.2.1 The Kanellakis-Smolka Algorithm

The Kanellakis-Smolka algorithm was created in 1983 and is able to reduce the number of states while making sure the graphs are bisimilar, giving a correct solution. The steps to do so are defined in Definition 2.2.2 below.

The point of the algorithm is to create a simpler but bisimilar graph. This is done by finding equivalent states and combining them, thus reducing the number of states without altering the behaviour of the system. Equivalence is decided by looking at each state's outgoing transitions. Two states are considered equivalent if each transition from one state has a corresponding transition with the same action (the equivalent of a channel in $\mathcal{E}$-cooperative bisimulation) from the other. The corresponding transition must reach a state equivalent to the first state's transition, i.e., they must point to the same block in the partition.

Definition 2.2.1 ([3])**.** The *stratified bisimulation relations* $\sim_k \subseteq S \times S$ for $k \in \mathbb{N}$, where S is the set of states and Act is the set of actions, are defined as follows:

- $s_0 \sim_0 s_1$ for all $s_0, s_1 \in S$
- $s_0 \sim_{k+1} s_1$ iff for each $a \in Act$: if $s_0 \xrightarrow{a} s_0'$ then there is $s_1' \in S$ such that $s_1 \xrightarrow{a} s_1'$ and $s_0' \sim_k s_1'$; and if $s_1 \xrightarrow{a} s_1'$ then there is $s_0' \in S$ such that $s_0 \xrightarrow{a} s_0'$ and $s_0' \sim_k s_1'$

Lemma 2.2.1 ([3])**.** *Assume that $(S, Act, \rightarrow)$ is a labelled transition system and let $s_0, s_1 \in S$. Then $s_0 \sim s_1$ iff $s_0 \sim_k s_1$ for all $k \in \mathbb{N}$.*

To find all equivalent states, the algorithm starts by assuming all states are equivalent, i.e., they all belong to the same block, followed by repeatedly splitting blocks containing states with a non-corresponding transition whenever they are encountered. Because of this, it does not have to combine equivalent states, only split when necessary, ensuring we get a correct solution.

Definition 2.2.2 (The Kanellakis-Smolka Algorithm [3])**.** Let π be a set with a block containing all states in the graph, then follow these steps:

1. For all blocks $B \in \pi$, do the following:
   (a) For each action a, lexicographically sort the a-labelled transitions from the states in block B.
   (b) Then all blocks in π are partitioned using B with consideration to a with the help of a split function.
   (c) If a new partition $\{B_1, B_2\}$ is given by the split function, then B is replaced with the new partition $\{B_1, B_2\}$ in π.
2. If no updates have been made to π, the algorithm terminates and π is the resulting partition. Otherwise repeat steps 1 and 2.

The split function in the algorithm picks an arbitrary state $S \in B$ and goes through all states $D \in B$. Then it checks whether the transitions from D go to the same

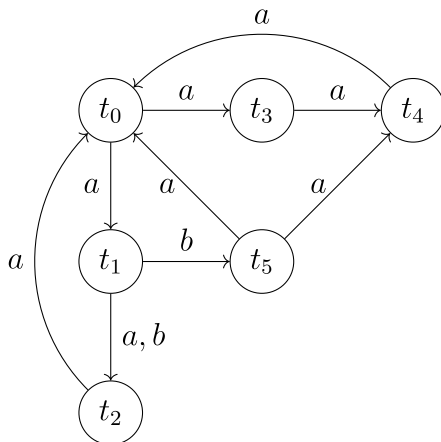

Figure 2.6: Transition system from example 0.2.5 in [3]

blocks as the transitions from S. If this is the case then D is added to a block B_1, otherwise it is added to a block B_2. If B_2 is empty after all states in the block have been checked, i.e., no states were found to not be equivalent, B cannot be refined further and $\{B_1\}$ containing the equivalent states is returned. Otherwise the partitioned set $\{B_1, B_2\}$ is returned.

The split function takes $O(m)$ time and iterates $n-1$ times in the algorithm, where m is the number of transitions and n the number of states. The use of lexicographic sorting seen in step 1.a in Definition 2.2.2 helps with the efficiency of the splitting. Lexicographic sorting in this case means sorting the transitions based on the channel name in alphabetical order, followed by grouping the transitions whose channels are equal by which block the transition ends up in. When searching for transitions with a certain channel, they are all grouped together and early termination of the search is possible, increasing the efficiency. The sorting requires $O(m)$ time using an algorithm such as the one presented in [4]. All elements in the partition are iterated over, making the total complexity $O(mn)$ [3].

This algorithm is not optimal which will be expanded on in the next section. They also conjectured that a solution with complexity $O(m \log n)$ for the problem was possible [5].

Example application of the algorithm

An example application of this algorithm is given in [3] where we have the transition system shown in Fig. 2.6. The calculation is a bit simplified as to not bore the reader with lots of loops and comparisons leading nowhere. The lexicographic sorting is also omitted due to the small size of the problem.

Let π be the set with a block containing the initial partition associated with this labelled transition, where

$$\pi = \{\{t_0, t_1, t_2, t_3, t_4, t_5\}\}.$$

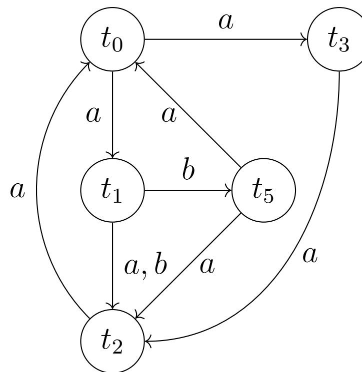

Figure 2.7: The reduced bisimilar system

The single block in π is a splitter for itself. Some states can do b-labelled transitions while others cannot. If we split the block in π using itself with respect to action b we obtain a new partition consisting of the blocks

$$\{\{t_1\}, \{t_0, t_2, t_3, t_4, t_5\}\}.$$

Since t_1 is the only state whose outgoing transitions are b-labelled it becomes its own block. Next we will iterate through all states whose outgoing transitions are a-labelled. We start with state t_0 and see that it has two outgoing transitions, one to block $\{t_1\}$ and another to $\{t_0, t_2, t_3, t_4, t_5\}$. No other state goes to both blocks, thus t_0 becomes it own singleton block and π becomes

$$\{\{t_0\}, \{t_1\}, \{t_2, t_3, t_4, t_5\}\}.$$

Then we continue with states t_2 and t_4 which has outgoing transitions going to $\{t_0\}$ only. We also see that t_5 goes to both $\{t_0\}$ and $\{t_2, t_3, t_4, t_5\}$ and that t_3 goes to $\{t_2, t_3, t_4, t_5\}$. Since they do not go to the same blocks, we split the block into three parts (done in two iterations of the algorithm as it can only split into two parts per iteration). The resulting partition is

$$\{\{t_0\}, \{t_1\}, \{t_2, t_4\}, \{t_3\}, \{t_5\}\}.$$

The partition cannot be refined further from this point so we conclude that states t_2 and t_4 can be combined while the others must stay as they are. The resulting transition system can be seen in Fig. 2.7.

2.2.2 The Paige-Tarjan Algorithm

Kanellakis and Smolka conjectured that the algorithm could be improved to only need $O(m \log n)$ time [5]. This was proved by Paige and Tarjan in 1987 [6].

To make the algorithm more efficient, the Paige-Tarjan algorithm keeps track of the partitions from the previous iteration. Instead of going through every block, it only looks at the blocks not present in the previous iteration. This makes sure that

already stable partitions (partitions that cannot be refined) are not looked at again as it is not possible to split them further. The algorithm's approach to refining blocks is based on Hopcroft's technique of 'processing the smaller half' [6]. This approach says that when a block has been partitioned with respect to a channel it does not need to be partitioned again until it is split. A core part is that when splitting further, only a block smaller than half the size of the original block from a previous split is processed, thus halving the size of the processed blocks each time. Together with the use of a more efficient splitting method called three-way splitting, see Fig. 2.8, this technique allows the partitioning of a block to take logarithmic time, $O(\log n)$ [6], [7]. The three-way splitting method splits a carefully chosen block B into either one, two or three blocks depending on the stability of the resulting partition, something that can be done efficiently using set operations.

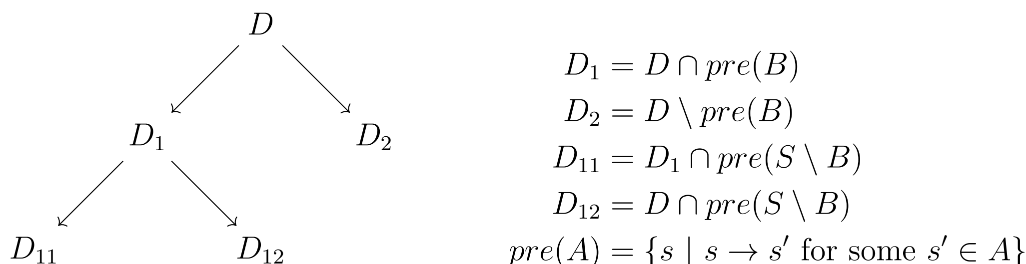

Figure 2.8: Three-way split. D, B and S are blocks. D_1 and D_2 are only calculated if D is not stable with regards to B, i.e., cannot be refined further. D_{11} and D_{12} are only calculated if D_1 is not stable with regards to $S \setminus B$.

Both the Paige-Tarjan and Kanellakis-Smolka algorithms handle only the simple bisimulation mentioned in the existing literature which cannot be used for our purposes. This is because minimising communication between agents is a parametric problem. In traditional bisimulation, only the graph to be reduced has to be inspected as the equivalence of states is determined only by the graph they belong to. This is not the case here. As mentioned in Section 2.1, because other agents have the ability to influence the behaviour of the system to be minimised, we have to minimise with respect to a parameter system that can communicate with it. For this, a parametric algorithm is necessary. We propose such algorithms with similarities to the above mentioned algorithms but using $\mathcal{E}$-cooperative bisimulation instead.

3

Methods

3.1 Algorithmic Solutions for $\mathcal{E}$-Cooperative Bisimulation

The main focus of this thesis is the creation of algorithms for reducing unnecessary communication. To accomplish this we use $\mathcal{E}$-cooperative bisimulation due to the reasons explained in Section 2.1. To be able to create an efficient parametric algorithm, we wanted to study existing (traditional) bisimulation algorithms and see how they function, as well as other work within the area of bisimulation. We wanted to explore what existing techniques were available that we could incorporate in our proposed algorithms. This was an important part so we would completely understand the topic. We concluded that some techniques used in the Kanellakis-Smolka and Paige-Tarjan algorithms could most likely be used in an implementation of a parametric $\mathcal{E}$-cooperative bisimulation algorithm, such as lexicographic sorting and three-way splitting.

The algorithms proposed in this thesis were primarily created as sequential algorithms with parallelised versions proposed afterwards. This was done because it is easier to implement sequential algorithms than parallel algorithms, since if parallel executions are not handled correctly, it can become difficult to debug and find what is causing possible errors. After the proposal of each algorithm, they were implemented, analysed and tested. The processes of calculating the complexity and evaluating the performance are described in Section 3.2 and Section 3.3 respectively.

The implementation was done using an existing Java framework named *teamwork*, created as a part of the SynTM project[1]. The framework provides algorithms and data structures for the purpose of creating individual system models from a specification. By using and extending this framework, we could focus on implementing and testing our proposed algorithms instead of needing to create all necessary data structures and algorithms from scratch. The test bench we created was made as a part of the framework, something that will be described in Section 3.3.

[1]Source code available at `https://github.com/SynTEAM-Y/teamwork`. Our additions are available at `https://github.com/SynTEAM-Y/teamwork/tree/masterthesis-bisim`

3.2 Study of Computational Complexity

Computational complexity is a concept used for estimating how an algorithm scales based on the input. Computation time (time complexity) and the memory usage (space complexity) are generally of particular interest. In this thesis we primarily focus on the time complexity as the memory available is not a limiting factor. The workflow where our algorithm is included only creates the system models, which is usually done on computers with good hardware.

Computational complexity is a good way of analysing an algorithm's performance but it can sometimes be misleading. We will be using big O notation which does not show constants and coefficients which can vary greatly in size, making it possible for an algorithm to look worse than another despite being more efficient in all practical cases. It only shows the worst case complexity (i.e., an upper bound) which for some algorithms can be very rare, with the average case being considerably better.

There exists problems where algorithms with better complexity perform worse than ones with inferior complexity. One such case is Zielonka's algorithm for solving parity games [2]. Its complexity is exponential while alternative algorithms exist running in quasi-polynomial time. Despite this, the implementation of Zielonka's exponential algorithm has the best performance. It is thus very important to not just look at the computational complexities when evaluating an algorithm. Thus, studying the computational complexity gives a fair estimate of how the algorithm will perform on large inputs.

When calculating the complexities, we follow a few assumptions. We assume that the number of channels (denoted k) cannot be greater than the number of transitions (denoted m). If k were to be greater than m, there would be unused channels that could be disregarded. We also assume that the number of states (denoted n) cannot be more than one greater than the number of transitions in the system as the system must be connected. The possible extra state is due to the fencepost problem [8]. We get that $k \leq m$ and $n \leq m + 1$, and thus $O(k) \leq O(m)$ and $O(n) \leq O(m)$.

To help illustrate the implementations of the algorithms, we present pseudocode for each one to make them more easily understood. This allows easy analysis of what the computational complexity is, as well as what parts contribute the most to the total complexity.

Calculating the complexity of a parallel algorithm is not as straightforward as for a sequential one. Two important measurements are presented by Vishkin in [9]; the work and the time (later referred to as span) of the algorithm. The work of the algorithm is all operations done by the algorithm (denoted T_1), i.e., the complexity of the algorithm running completely sequentially. The span is the longest possible execution path of the parallel algorithm (denoted T_∞). The execution paths are usually illustrated as a DAG (Acyclic Directed Graph), where the span is its critical path, also called its longest strand. The possible speedup can be calculated by dividing the work with the span ($\frac{T_1}{T_\infty}$). Note that the speedup is shown as a ratio, not as a complexity. The speedup possible with only p processors can be calculated by dividing the work with p, i.e., $\frac{T_1}{p}$.

3.3 Performance Evaluation

To find out whether our implemented algorithms perform better than the existing proof of concept algorithm, and by how much, a test bench was created as none existed previously. The test bench was implemented directly into the *teamwork* framework. It is quite primitive and consists of a bash script executing the different steps, while measuring, going from a specification to individual system models. Previously, the process of going from a specification to system models was continuous with no way to separate the different execution stages. This had to be modified for us to be able to measure the time and memory for only our algorithm.

The time taken to execute the algorithm is measured using the `time` command available on Linux. On Ubuntu (a Linux distribution), it has a precision of 1 ms and runs a specified program or command while measuring its execution time. In some rare cases, when the program to be measured terminates, `time` does not immediately stop the measurement, which can give inaccurate results [10]. The impact is small and can be reduced further by repeating the execution, something that should be done anyway to get accurate results.

The memory usage is harder to measure and necessitated the use of a purpose-made program. The program we settled on is called Valgrind. Valgrind is used for finding bugs and memory leaks but can use a tool called massif to measure the memory used on the heap [11], [12]. The massif tool gives a graph of the heap usage over the course of the execution, showing the maximum memory used and at what point it occurred. With this information we can reason about the algorithm's memory usage. As previously explained, we do not focus on the space complexity so these measurements will only be used to find algorithms with abnormal memory usage.

Using the test bench introduced above, each algorithm was tested using input systems of differing sizes and complexities. We use multiple transition systems to find strengths and weaknesses of the algorithm since some algorithms may perform better only on systems with certain properties. The specifications for these systems were chosen to maximise the possibility of finding these instances.

4

Results

4.1 Applying $\mathcal{E}$-Cooperative Bisimulation

As previously explained, traditional bisimulation cannot be used to reduce transition systems representing the communication between autonomous agents because it is a parameterised problem (a system must be reduced with respect to another). Parameterised problems are not supported so another method must be used instead, such as $\mathcal{E}$-cooperative bisimulation. Thus, traditional bisimulation algorithms cannot be used 'out of the box', instead having to be adapted. Partition refinement algorithms performing $\mathcal{E}$-cooperative bisimulation have lots of similarities with traditional ones, but the methods of finding splitters are different as they must fulfil different criteria.

The method for finding a splitter using $\mathcal{E}$-cooperative bisimulation is intricate and the resulting partition after splitting must satisfy the conditions in Definition 2.1.4. A function f was defined for the purpose of getting a refined relation in the original definition of $\mathcal{E}$-cooperative bisimulation [1]. The function returns a relation where all remaining pairs satisfy the conditions in Definition 2.1.4. The pairs not satisfying the conditions in the input relation are removed, thus giving the refined relation based on the current state of $\mathcal{R}$. This can be adapted to instead return refined partitions.

An implementation of the function f adapted for partitions exists in the *teamwork* framework and is used in the implementations of our proposed algorithms. The function applies conditions 2-3 in Definition 2.1.4 but not condition 1. The first condition is preprocessed before running the refinement algorithm to help increase performance. The condition will not be invalidated by succeeding iterations of the algorithm, thus there is no need to repeatedly verify it.

4.2 The Existing Proof of Concept Algorithm

A proof of concept algorithm using the original concept of relations was presented in [1], proving the computability of the problem and giving a reference point for further work. The algorithm takes two transition systems as inputs; the system to be reduced and a parameter system. The former system is reduced with respect to the latter to ensure correctness. Pseudocode for the algorithm is shown in Fig. 4.1.

The algorithm is parallel and starts with state-indexed relations (each state in the parameter gets its own relation) containing all possible pairs of states that may successively be removed once equivalence is disproved. This is determined by checking the conditions in Definition 2.1.4. The refined relation is given by the function f introduced in Section 4.1 above. When no more pairs in the relations can be removed, i.e., a joint fixed point has been reached, we find all equivalent states that can be combined in the entry state's relation. A joint fixed point occurs when all states' relations have reached individual fixed points. The reason we cannot terminate the algorithm when the entry state's relation has reached a fixed point is because of the recursive definition of f. Updates to other states' relations might make the entry state's relation no longer fixed and thus the computation must continue until all relations have reached fixed points.

```
1  input: T = ⟨S, s_0, Int, ls, L, Δ⟩ to be minimised with respect to the
2         parameter TS C = ⟨E, ϵ_0, Int_E, ls^E, L_E, Δ_E⟩
3  for all ϵ ∈ E, create R_ϵ = S × S, tmp_ϵ = ∅
4  while true
5    par ∀ϵ ∈ E
6      for all (s_1, s_2) in R_ϵ
7        if (s_1, s_2) not in f(R)_ϵ then
8          add (s_1, s_2) to tmp_ϵ
9        end
10     end
11   sync
12   if all tmp_ϵ are empty then
13     return {R_ϵ for each ϵ ∈ E}
14   end
15   for all ϵ in E
16     R_ϵ = R_ϵ \ tmp_ϵ
17   end
18 end
```

Figure 4.1: Pseudocode of the proof of concept algorithm using relations running in parallel

4.2.1 Computational Complexity

To calculate the complexity, we began by creating a DAG representing the execution of the algorithm, seen in Fig. 4.2. As explained in Section 3.2, this can be used to calculate the parallel complexity. By following the critical path of the DAG, we calculate the complexity as follows:

Creating the data structures is done in cubic time, $O(n^3)$ ($\mathcal{R}_\epsilon = S \times S$ for each state). Once again, n represents the number of states, m the number of transitions and k the number of channels. By sorting $\mathcal{R}_\epsilon$ when creating the structure, the checking of membership for each pair in $f(\mathcal{R})_{\epsilon_i}$ can be done in $O(\log n)$ time using binary search. After synchronising all the threads, the function returns if all temporary variables are empty (see lines 12-13 in Fig. 4.1). This check can be done linearly, $O(n)$.

The maximum number of iterations before returning is n^3. In the worst case, only one pair in one relation is removed per iteration. There are n^2 pairs in each relation, and each state has its own relation, resulting in n^3 pairs in total. Necessitating n^3 iterations is the upper bound and is very unlikely to happen in practice. The calculation of the complexity of $f(\mathcal{R})_{\epsilon_i}$ is more complicated.

The function f gives a refined relation using the current relations $\mathcal{R}_{\epsilon_1}, \mathcal{R}_{\epsilon_2}, ..., \mathcal{R}_{\epsilon_n}$. To calculate the refined relation $f(\mathcal{R})_{\epsilon_i}$, all pairs in $\mathcal{R}_{\epsilon_i}$ must be checked against Definition 2.1.4, requiring n^2 iterations. All channels c also have to be iterated over. Condition 2 is checked once for every state. To check condition 3, the function must loop over all transitions in the parameter TS. To check the existence of a transition fulfilling a condition in 3.a-d, all outgoing transitions from the state must be checked. Because the transition system must be deterministic, the maximum number of transitions is k. Condition 3.d requires the use of a search algorithm where breadth first search (BFS) is optimal to find all states reachable by silent transitions. The BFS takes $O(m+n)$ time. Thus the total computational complexity of f is $O(n^2 \cdot k \cdot (n+m \cdot (k+1+k+(k+m+n)))) = O(kn^2 \cdot (n+m+km+m^2+mn)) = O(k^2mn^2+km^2n^2+kmn^3)$. Using the assumptions explained in Section 3.2, we can write the complexity of f as $O(km^2n^2)$.

Together, this forms a span of $T_\infty = O(n + n^3 \cdot (km^2n^2 + \log n + n)) = O(km^2n^5)$ which is the parallel complexity of the algorithm. The complexity of the algorithm running sequentially (the work) is $T_1 = O(km^2n^6)$. By dividing the work by the span, we get the speedup ratio, $\frac{T_1}{T_\infty} = n$, thus giving a possible linear speedup when parallelising.

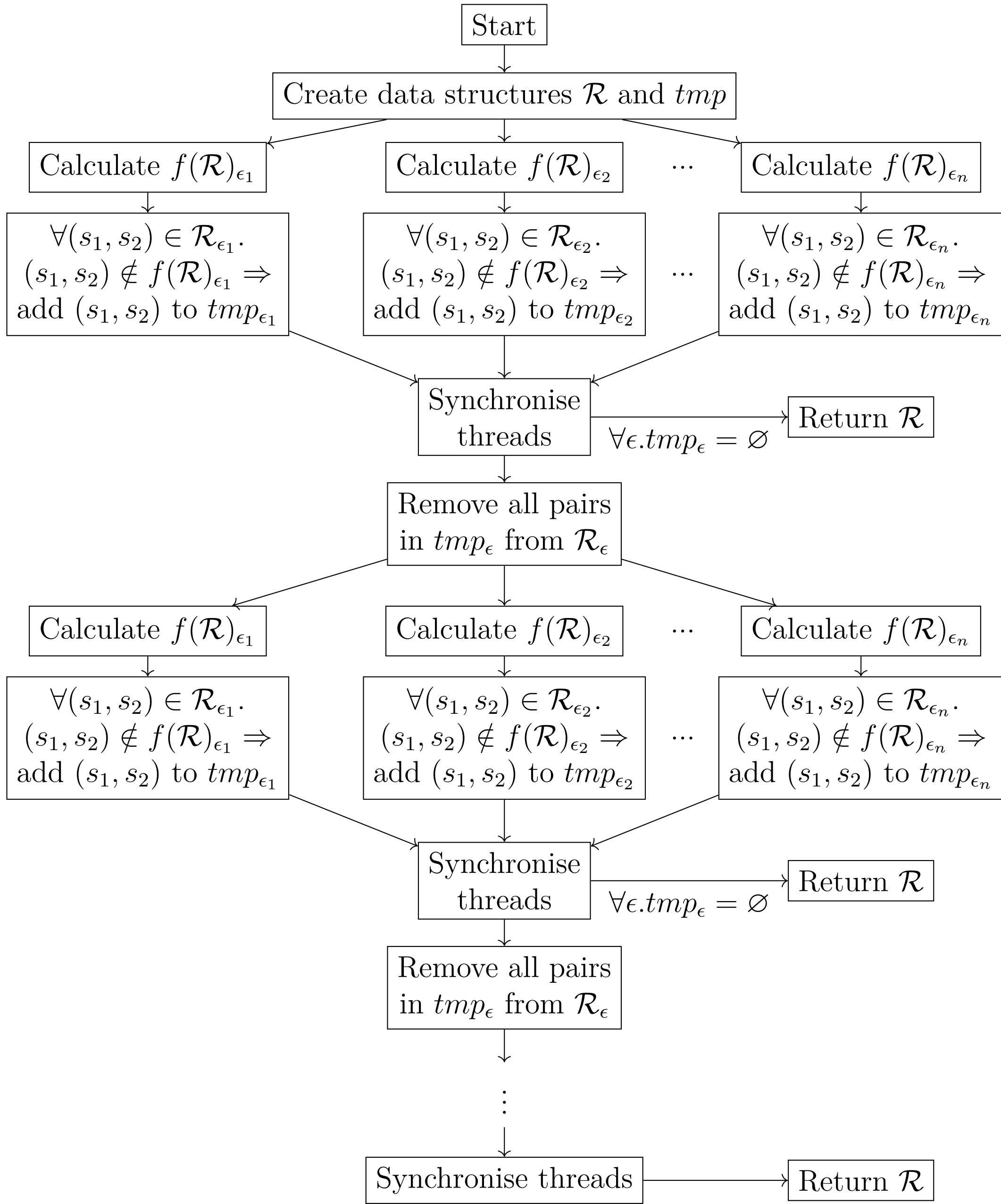

Figure 4.2: DAG of the execution of the proof of concept algorithm

4.2.2 Alternative Version

A simple translation of the proof of concept algorithm can be made to use partition refinement instead of dealing with relations. A version of this algorithm had been created in an earlier part of the SynTM project. This variation of the proof of concept algorithm served as a starting point for further development since we set out to create algorithms utilising the partition refinement technique. It works similarly to the original proof of concept algorithm, but instead of checking membership in the result of $f(\mathcal{R})$, the function f splits a block into two sets, one set with the states fulfilling the conditions in Definition 2.1.4 and one set with the rest. The block to be split is replaced by the two new blocks but only after finishing splitting all blocks in all partitions. The function f only verifies with regards to the previous iteration's resulting partitions, thus each state's partition can be refined in parallel. Note that only this version of the proof of concept algorithm has been implemented and is thus used as the baseline in the comparison with the other algorithms.

4.2.3 Performance

The computational complexity is poor and is thus only practical for smaller systems. The practical performance was tested using specifications listed in Appendix A. The resulting systems are of varying complexities and sizes, with the two largest being too big for the algorithm to handle in a reasonable amount of time. The algorithm did not manage to decompose the systems for a single agent in 24 hours, hence the N/A in Table 4.1. The choice of specifications will be discussed in Section 5.1.1.

Specification	Number of agents	Number of states in the central system	Time taken to optimise using the proof of concept algorithm (s)
LTL4simple (A.1)	4	5	0,264
AMBADecompArb3 (A.2)	4	5	0,270
AMBADPA10 (A.3)	7	13	1,246
LilyDemoV18 (A.4)	3	19	1,510
LTL3Arb (A.5)	3	27	7,641
Priority3Arb (A.6)	3	28	4,838
LTL2+3Arb (A.7)	5	84	N/A
Priority2+3Arb (A.8)	5	244	N/A

Table 4.1: Performance of the proof of concept algorithm

4.3 Improved Algorithm I

The first algorithm we propose is based on the proof of concept algorithm but uses partition refinement techniques instead of working directly with relations. We will refer to this algorithm as *Alg I* and is shown in Fig. 4.3. It is based on the proof of concept algorithm and uses a similar technique. The difference is how often the blocks are replaced and the use of lexicographic sorting. Instead of splitting a block

once before updating the partition for all states, *Alg I* stores the split blocks as intermediate results and tries to split those blocks as well. The splitting of each state's partition is done in parallel so when the partitions are updated, the parallel threads must synchronise. The synchronisation requires all threads to have finished their current tasks, making them wait for each other and are thus wasting time. By reducing the number of times we have to synchronise, a reduced execution time follows.

Only when no more splitting can be done for a state's partition, i.e., a fixed point has been reached, all states' partitions are updated with the intermediate results. This means that all partitions have reached fixed points when synchronising. We know for a fact that a joint fixed point requires all individual partitions to have reached local fixed points. Thus, there is no need to check if a joint fixed point exists unless that is the case. This fact is exploited by *Alg I* unlike the partition refinement translation of the proof of concept algorithm.

But just because each individual partition have reached local fixed points, it does not mean that a joint fixed point has been reached. Because each state's relation is only updated with regards to the previous iteration's resulting partition, other states' newly updated partitions may make the partition no longer be at a fixed point. Only after an iteration where no updates have been made to any partition, we can conclude that a joint fixed point has been reached and the algorithm can terminate, something that will happen eventually by the definition of f [1].

Lexicographic sorting, a technique used in the Kanellakis-Smolka algorithm, was also employed in this algorithm. It is used in the same way as in the Kanellakis-Smolka algorithm, grouping transitions together allowing early termination when searching for transitions while splitting. It does not alter the complexity but should in theory improve the practical performance.

4.3.1 Implementation

Instead of using relations, *Alg I* uses state-indexed partitions to keep track of equivalent states (denoted $\mathcal{H}$). The intermediate results are also stored in a similar data structure. In the implementation, hash maps are used for storing this data, where the keys are states and the values are the partitions. The reason for using hash maps is that they allow lookup in constant time, $O(1)$.

Instead of executing one iteration of the partitioning followed by checking if a fixed point has been found, *Alg I* performs multiple iterations until a local fixed point has been reached. To facilitate this, our implementation of *Alg I* uses a queue Q containing the initial partition of each state index. The refinement is done by taking a block from the queue, sort the block lexicographically and then try to split it. If successful, the new blocks are added to the queue and are added as intermediate results. Otherwise nothing is done and it moves on to the next block in the queue. When there are no more blocks in the queue, no more blocks can be split and a fixed point has been reached.

```
1  input: T = ⟨S, s_0, INT, LS, L, Δ⟩ to be minimised with respect to the
2         parameter TS C = ⟨E, ϵ_0, INT_E, LS^E, L_E, Δ_E⟩
3  for all ϵ ∈ E, create H_ϵ := {S}, tmp_ϵ := ∅
4  while true
5    for each state ϵ ∈ E
6      Q_ϵ, tmp_ϵ := H_ϵ
7      while Q_ϵ ≠ ∅
8        B := Q_ϵ.pop()
9        for each channel c ∈ Y
10         sort the transitions from B lexicographically
11         F := f(B, c, H)
12         if F ≠ B
13           Q_ϵ := Q_ϵ ∪ F
14           replace B with F in tmp_ϵ
15           break
16         end
17       end
18     end
19   end
20   if tmp_ϵ = H_ϵ for all ϵ ∈ E then
21     return H_ϵ1
22   end
23   for all ϵ ∈ E
24     H_ϵ := tmp_ϵ
25   end
26 end
```

Figure 4.3: Pseudocode of *Alg I*

Using this new technique, a fixed point will be found for the current state index before synchronising. After this has been done for all partitions in $\mathcal{H}$, it checks whether a joint fixed point has been reached, i.e., if none of the partitions have been updated in the last iteration. If this is not the case the algorithm repeats the procedure until a joint fixed point is found.

The algorithm was first implemented sequentially and after it had been analysed, a parallel version was created. Because each state has its own independent queue, they can be run in parallel.

4.3.2 Computational Complexity

To estimate the performance we calculate the time complexity for both the sequential and parallel versions of this algorithm.

Sequential version

The sequential *Alg I* proposed in Fig. 4.3 is an improvement of the original proof of concept algorithm. The big difference is the use of partitioning instead of relations. Lexicographic sorting makes the algorithm more efficient and can be done in $O(m)$ time using an algorithm from [4], but does not affect the total complexity. By grouping the transitions, it makes it easier to find the right transitions, but even if finding transitions could be done in constant time it would not make a difference to the overall complexity. This is because of the search algorithm needed to verify condition 3.d in Definition 2.1.4.

The function f is mostly the same as for the original proof of concept algorithm, but in this case it partitions a block into two new blocks, one with states that fulfil Definition 2.1.4 and the other with states that do not. The overall complexity of f is not changed except that the loop over channels is moved outside the function (thus the complexity is $O(m^2n^2)$). The factor k is added subsequently by a loop outside.

Just like in the case of the proof of concept algorithm, in the worst case only one state's equivalence is disproved per iteration. This time we only have n partitions with n elements, thus totalling n^2 instead of the proof of concept's n^3 possible iterations. Thus, the loops on lines 4 and 7 in Fig. 4.3 add a factor of $O(n^2)$ to the total complexity. The for-loop on line 5 is parallelisable but in a sequential setting adds a factor $O(n)$. In total, the sequential algorithm requires $O(km^2n^5)$ time, equalling the span of the parallel proof of concept algorithm.

Note that the new technique of reducing the number of synchronisations required does not alter the complexity of the algorithm. This is because it is possible for each partition to reach a local fixed point every iteration. However, this is unlikely.

Parallel version

The parallelised version of *Alg I* replaces the for-loop on line 5 in Fig. 4.3 with parallel execution for each state. The DAG in Fig. 4.4 shows the execution paths of the algorithm. The work is the same as the sequential version's complexity, $T_1 = O(km^2n^5)$. As the executions for each state index are done in parallel, we can see in the DAG that a factor n is not included in the span. Thus, the span is $T_\infty = O(km^2n^4)$ which is one degree better than the parallel proof of concept algorithm. The speedup ratio is the same as for the proof of concept algorithm, namely n.

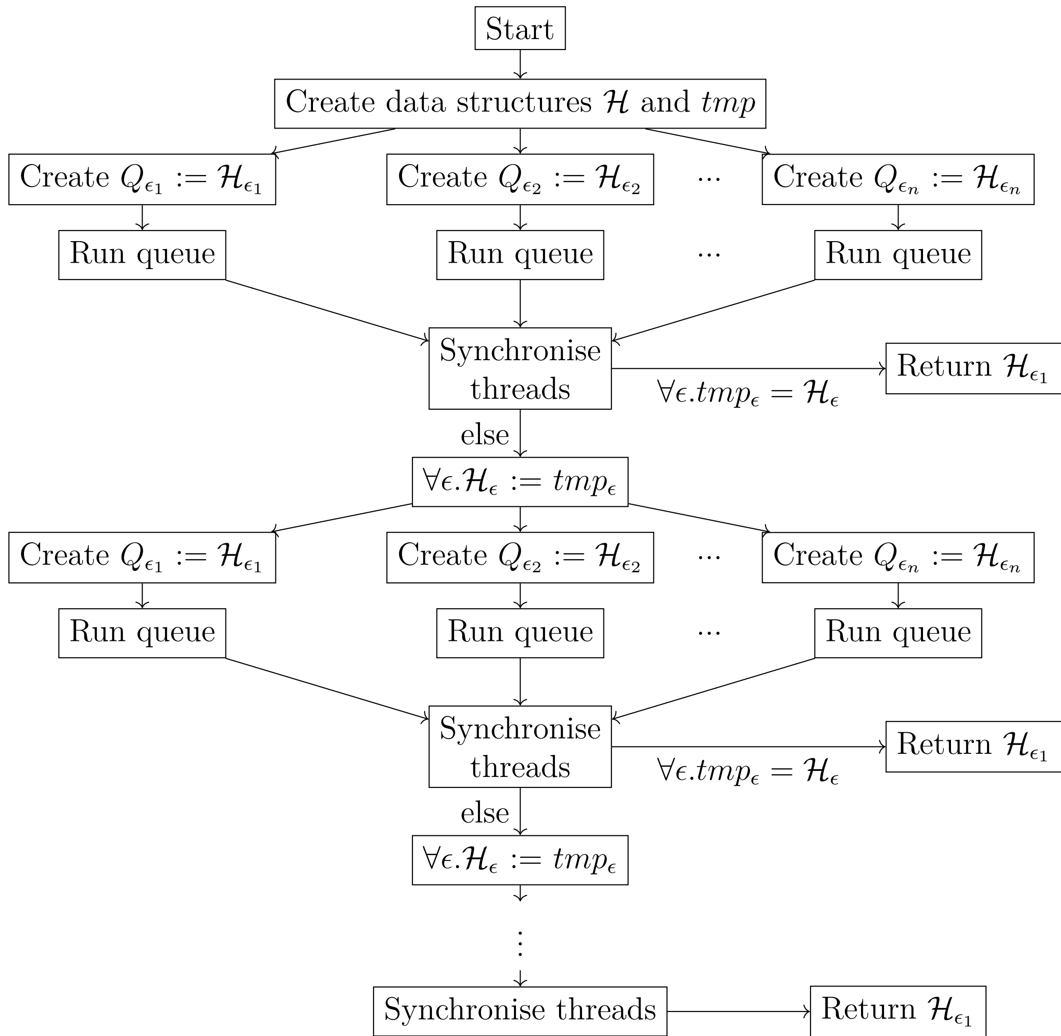

Figure 4.4: DAG of the execution of *Alg I*. The DAG for ‘Run queue’ can be found in Fig. 4.5

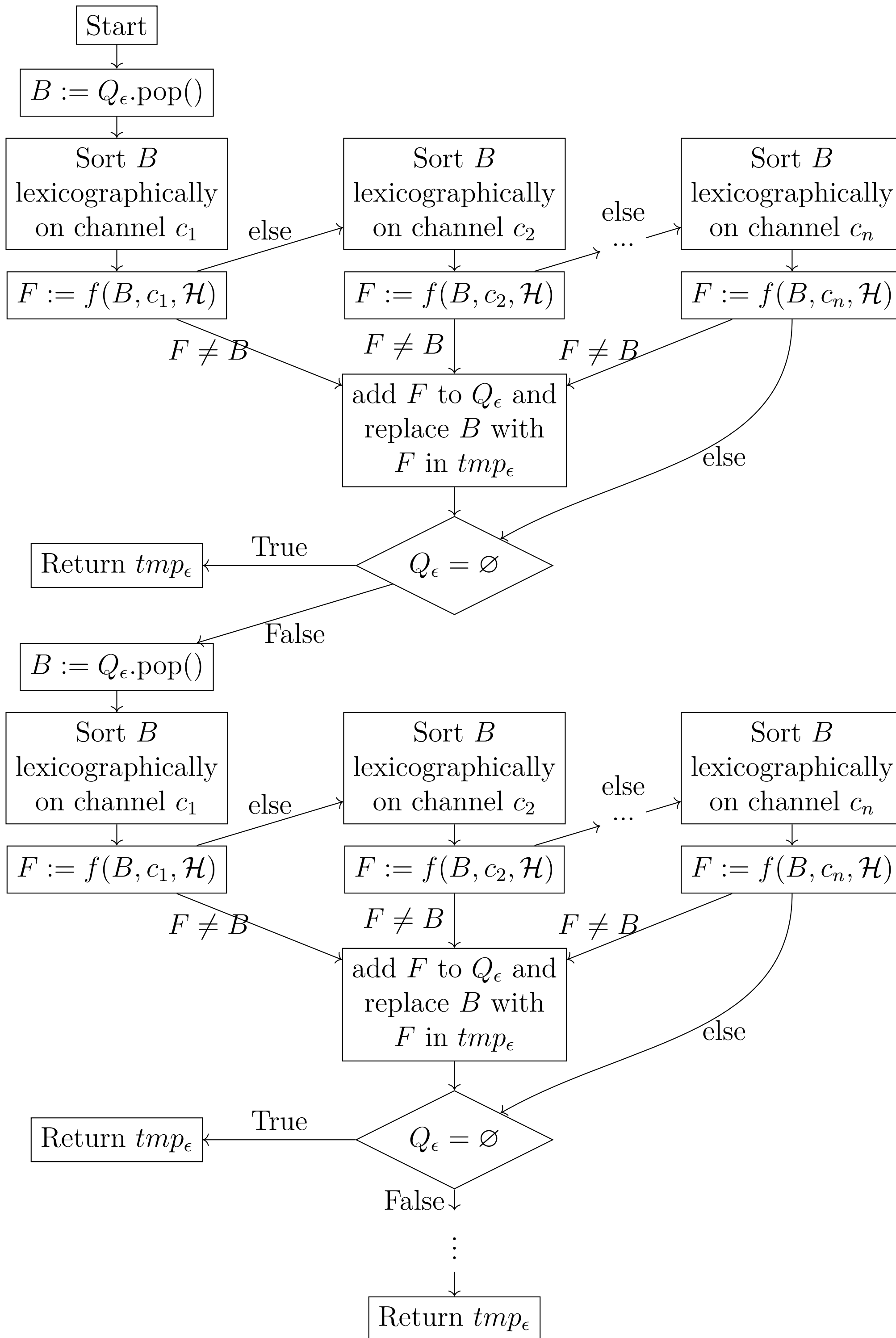

Figure 4.5: DAG of the execution of ‘Run queue’

4.3.3 Performance

As we are only interested in the best performing version, we will only present results of the parallel algorithm. The parallel version performed much better than the parallel proof of concept algorithm (using partitioning), with more states resulting in a larger difference. The exact results are shown in Table 4.2.

The big thing to note here is the results for the two largest systems. The proof of concept algorithm was not able to decompose the systems in 24 hours, but *Alg I* was able to do it in approximately 30 seconds and 30 minutes respectively. This is a big improvement, beyond expectations.

Specification	Number of agents	Number of states in the central system	Average time taken to optimise using *Alg I* (s)
LTL4simple (A.1)	4	5	0,264
AMBADecompArb3 (A.2)	4	5	0,273
AMBADPA10 (A.3)	7	13	1,242
LilyDemoV18 (A.4)	3	19	1,001
LTL3Arb (A.5)	3	27	1,576
Priority3Arb (A.6)	3	28	1,740
LTL2+3Arb (A.7)	5	84	31.597
Priority2+3Arb (A.8)	5	244	1575,917

Table 4.2: Performance of *Alg I*

Specification	Number of states in the central system	Proof of concept (s)	*Alg I* (s)
LTL4simple (A.1)	5	0,264	0,264
AMBADecompArb3 (A.2)	5	0,270	0,273
AMBADPA10 (A.3)	13	1,246	1,242
LilyDemoV18 (A.4)	19	1,510	1,001
LTL3Arb (A.5)	27	7,641	1,576
Priority3Arb (A.6)	28	4,838	1,740
LTL2+3Arb (A.7)	84	N/A	31.597
Priority2+3Arb (A.8)	244	N/A	1575,917

Table 4.3: Performance comparison between *Alg I* and proof of concept

4.4 Improved Algorithm II

The second algorithm proposed, named *Alg II* (shown in Fig. 4.6), is based on *Alg I* and employs partition refinement techniques. *Alg II* uses a different splitting technique than *Alg I* by firstly not using lexicographic sorting and secondly by performing the equivalent of Paige-Tarjan's three-way split (denoted $f^{\star}$). The reason why lexicographic sorting is not used is because it does not affect the speed of the refinement technique of the algorithm. This is because this algorithm's refinement technique does not care about the internal ordering of each block, but only the size of the block. *Alg II*s refinement of partition is inspired by Hopcroft's 'process the smaller half' technique, meaning processing the smaller block when splitting. Thus, the size of each block matters when refining.

Unlike the previous two algorithms, this algorithm also keeps track of the partition from the previous iteration. This is done so only blocks that were refined in the last iteration are iterated over, i.e., the blocks that can be split further. If there has not been a previous iteration yet the current partition will be preprocessed which consists of running each blocks in the partition naïvely through f once (equivalent to one iteration of the proof of concept algorithm). A previous iteration is required for the algorithm to function. When the previous partition and the current partition are the same, a fixed point has been reached.

When the algorithm is executed in parallel the refinement and splitting of the partition has been completed for each state, just as in *Alg I*. Then all of the threads are synchronised and the partitions are updated until a joint fixed point has been reached.

4.4.1 Implementation

Just as *Alg I*, this algorithm stores intermediate results, uses hash maps and state-indexed partitions to keep track of equivalent states. It also uses *Alg I*'s concept of performing multiple iterations until a fixed local point has been reached, but it differs when it comes to how the local fixed point is found.

Alg II takes inspiration from the Paige-Tarjan algorithm when it comes to splitting and refining blocks, and keeping track of the partition from the previous iteration. The previous partition is denoted X in Fig. 4.6.

The algorithm starts by setting X and the current partition to the initial partition of each state index. Since no iteration of the refinement has been done yet, the current partition is preprocessed. This is done to find which blocks can be refined further, by comparing the previous and current partition. The preprocessing is done by applying f to the current partition and a queue Q is set to the current partition. Note that the split function in the preprocessing stage is the same as in *Alg I*. This is because our equivalent of the three-way split needs blocks from the previous partition, but since there has not been a previous iteration yet it is not possible to use the three-way splitting technique.

```
1  input: T = ⟨S, s_0, INT, LS, L, Δ⟩ to be minimised with respect to the
2         parameter TS C = ⟨E, ε_0, INT_E, LS^E, L_E, Δ_E⟩
3  for all ε ∈ E, create H_ε := {S}, tmp_ε := ∅
4  while true
5      for each state ε ∈ E
6          Q_ε, X_ε, tmp_ε := H_ε
7          for each block D ∈ X_ε
8              for each channel c ∈ Y
9                  F := f(D, c, H)
10                 if F ≠ D
11                     Q_ε := Q_ε ∪ F
12                     replace D with F in tmp_ε
13                     break
14                 end
15             end
16         end
17         while Q_ε ≠ ∅
18             pick a block S ∈ X_ε \ tmp_ε
19             find and remove a block B ∈ Q_ε such that
20                 B ⊆ S and | B | ≤ 1/2 | S |
21             replace S in X_ε with B and S \ B
22             for each channel c ∈ Y
23                 F := f*(B, S \ B, c, H)
24                 if F ≠ B
25                     Q_ε := Q_ε ∪ F
26                     replace B with F in tmp_ε
27                     break
28                 end
29             end
30         end
31     end
32     if tmp_ε = H_ε for all ε ∈ E then
33         return H_ε1
34     end
35     for all ε ∈ E
36         H_ε := tmp_ε
37     end
38 end
```

Figure 4.6: Pseudocode of *Alg II*

After the preprocessing is done, a block S is picked from a set containing the blocks that were split during the previous iteration. Then a block B is removed from the queue where B is a block given by splitting S and is less than or equal to half the size of S. The block S in the previous partition is then replaced by the blocks B and a block containing the states of S not in B. Then block B is tried to be split using an equivalent to Paige-Tarjan's three-way split. If successful, B is replaced by the new blocks created from the split in the current partition and the new blocks are also added to the queue for further splitting. This whole process is repeated until the queue is empty which means that the previous partition is equal to the current partition, and a fixed point has been found.

Alg II was also first implemented sequentially and after it had been analysed, a parallel version was created of it. For the same reason as mentioned in Section 4.3.1, each state's partition refinement can be run in parallel.

4.4.2 Computational Complexity

To estimate the performance, we calculate the time complexity for both the sequential and parallel versions of this algorithm.

Sequential version

The sequential *Alg II*, just like *Alg I*, is an improvement of the proof of concept algorithm (using partitioning instead of relations). However, this algorithm uses the technique of keeping track of the prior partition to efficiently refine the current partition.

To be able to achieve this, the current partition needs to be preprocessed on the first iteration of the refinement (as stated in Section 4.4.1) which in the pseudocode (Fig. 4.6) occurs between the lines 7 and 15. The function f during the preprocessing behaves in the same way as in *Alg I* and has the time complexity of $O(m^2n^2)$ where the factor k is added subsequently by the outside loop on line 7. Then because the function f is in the worst case run over n elements on line 8, a factor n is added to the complexity. This results in the preprocessing part having the time complexity $O(km^2n^3)$

After the preprocessing, the algorithm continues by refining the current partition in a Paige-Tarjan manner as described earlier. This occurs between lines 16 - 27. Instead of using the function f, a function $f^\star$ is used. $f^\star$ works similarly to the three-way split used in the Paige-Tarjan algorithm, by using f to split the two input blocks. This means that $f^\star$ has a time complexity of $O(m^2n^2)$. Thus, having the time complexity of $O(m^2n^2)$ where the factor k is added subsequently by the loop outside. But since the input for $f^\star$ is always half the size or smaller than then the previous split blocks from last iteration, the input size becomes $\log n$. Because the input has size $\log n$, the complexity of $f^\star$ will be $O(m^2(\log n)^2)$ instead of $O(m^2n^2)$. In the worst case the refinement will need to be iterated n times on line 16, leading the refinement to have the complexity of $O(km^2n(\log n)^2)$. Adding the time

complexities of the preprocessing and the refinement, results in the time complexity $O((km^2n(\log n)^2 + O(km^2n^3)) = O(km^2n^3)$.

Just like in the case of *Alg I*, in the worst case scenario, only one state's equivalence is disproved per iteration. Since we only have n elements in n partitions and loops on lines 4 and 8 can be combined in adding a factor of $O(n^2)$ to the total complexity. The for-loop on line 5 is also parallelisable but in a sequential setting adds a factor $O(n)$. In total, the sequential algorithm requires $O(km^2n^5)$ time, which is equivalent to the sequential version of *Alg I*.

Parallel version

Alg II can execute each state's partition refinement in parallel by replacing the for-loop on line 5 in Fig. 4.6. The DAG seen in Fig. 4.7 shows the execution paths of the algorithm. The work of the algorithm is equal to the sequential version's time complexity, $O(km^2n^5)$. Calculating the longest strand of the DAG in Fig. 4.7 gives a span of $O(km^2n^4)$ which is equal to *Alg I*'s span. The speedup ratio of *Alg II* thus becomes n.

4.4.3 Performance

The algorithm performs well and the test results are shown in Table 4.4. The parallel version of *Alg II* performed slightly better than *Alg I* for medium sized systems but the differences are small. A comparison can be seen in Table 4.5. For the largest two systems it was much slower than *Alg I*, so much so that not a single agent was able to be decomposed in 24 hours while *Alg I* was able to do it in 30 seconds. We believe the cause of this to be the requirement of a block in the queue to be a subset of another block, $B \subseteq S$. Verifying whether a set is a subset is not instant and as the number of sets increase, this verification will slow down as there are more sets to be checked. This is not an inherent flaw to the algorithm and can most likely be mitigated using a better implementation of the algorithm. This will be discussed further in Section 5.1.2.

Start

Create data structures $\mathcal{H}$ and tmp

Create $Q_{\epsilon_1}, X_{\epsilon_1} := \mathcal{H}_{\epsilon_1}$ | Create $Q_{\epsilon_2}, X_{\epsilon_2} := \mathcal{H}_{\epsilon_2}$ | $\cdots$ | Create $Q_{\epsilon_n}, X_{\epsilon_n} := \mathcal{H}_{\epsilon_n}$

Preprocess | Preprocess | $\cdots$ | Preprocess

Refinement | Refinement | $\cdots$ | Refinement

Synchronise threads

$\forall \epsilon . tmp_\epsilon = \mathcal{H}_\epsilon$ → Return $\mathcal{H}_{\epsilon_1}$

else

$\forall \epsilon . \mathcal{H}_\epsilon := tmp_\epsilon$

Create $Q_{\epsilon_1}, X_{\epsilon_1} := \mathcal{H}_{\epsilon_1}$ | Create $Q_{\epsilon_2}, X_{\epsilon_2} := \mathcal{H}_{\epsilon_2}$ | $\cdots$ | Create $Q_{\epsilon_n}, X_{\epsilon_n} := \mathcal{H}_{\epsilon_n}$

Preprocess | Preprocess | $\cdots$ | Preprocess

Refinement | Refinement | $\cdots$ | Refinement

Synchronise threads

$\forall \epsilon . tmp_\epsilon = \mathcal{H}_\epsilon$ → Return $\mathcal{H}_{\epsilon_1}$

else

$\forall \epsilon . \mathcal{H}_\epsilon := tmp_\epsilon$

$\vdots$

Synchronise threads → Return $\mathcal{H}_{\epsilon_1}$

Figure 4.7: DAG of the execution of *Alg II*. The DAG for ‘Preprocess’ can be found in Fig. 4.8 and the DAG for ‘Refinement’ in Fig. 4.9

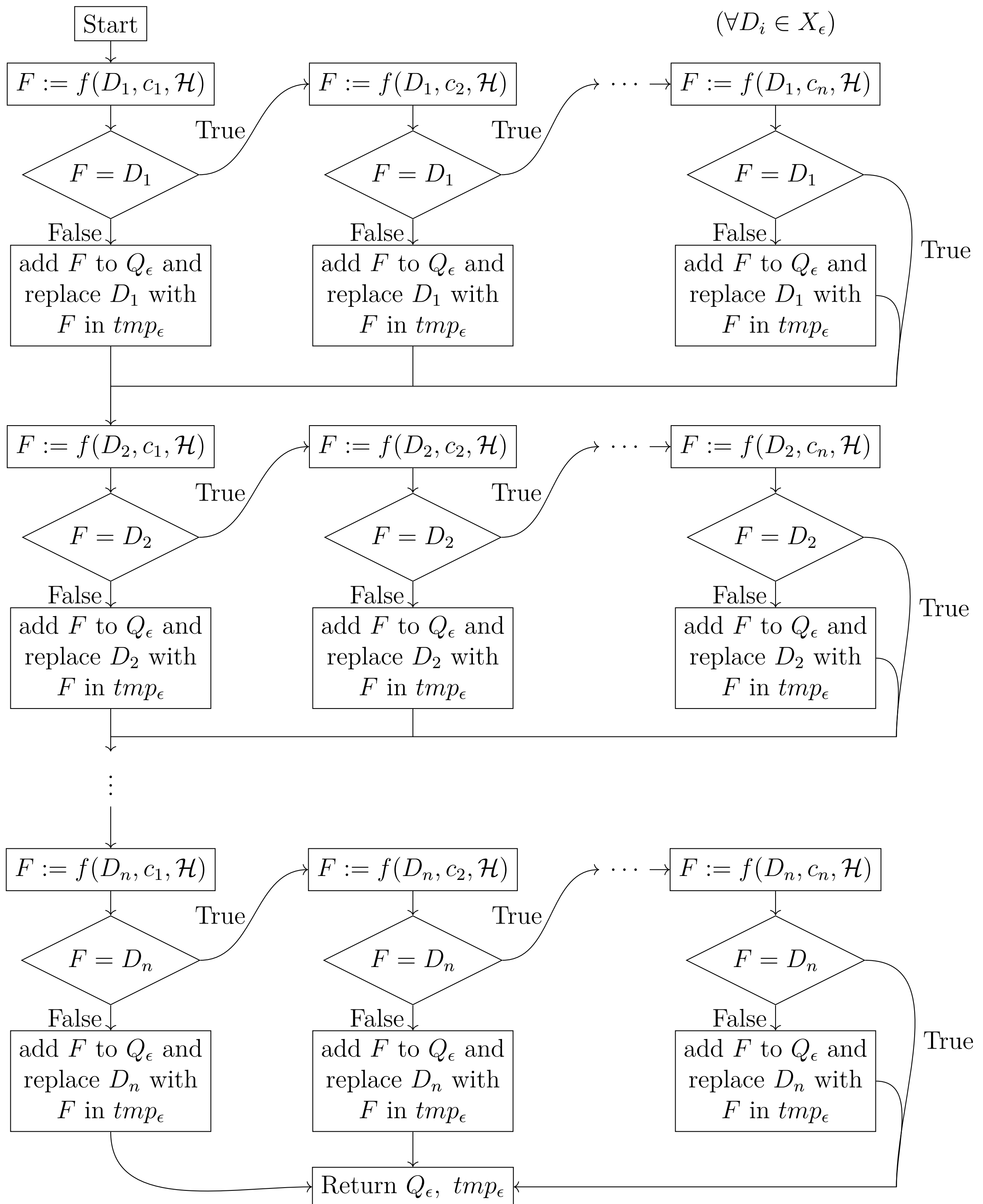

Figure 4.8: DAG of the execution of 'Preprocess'

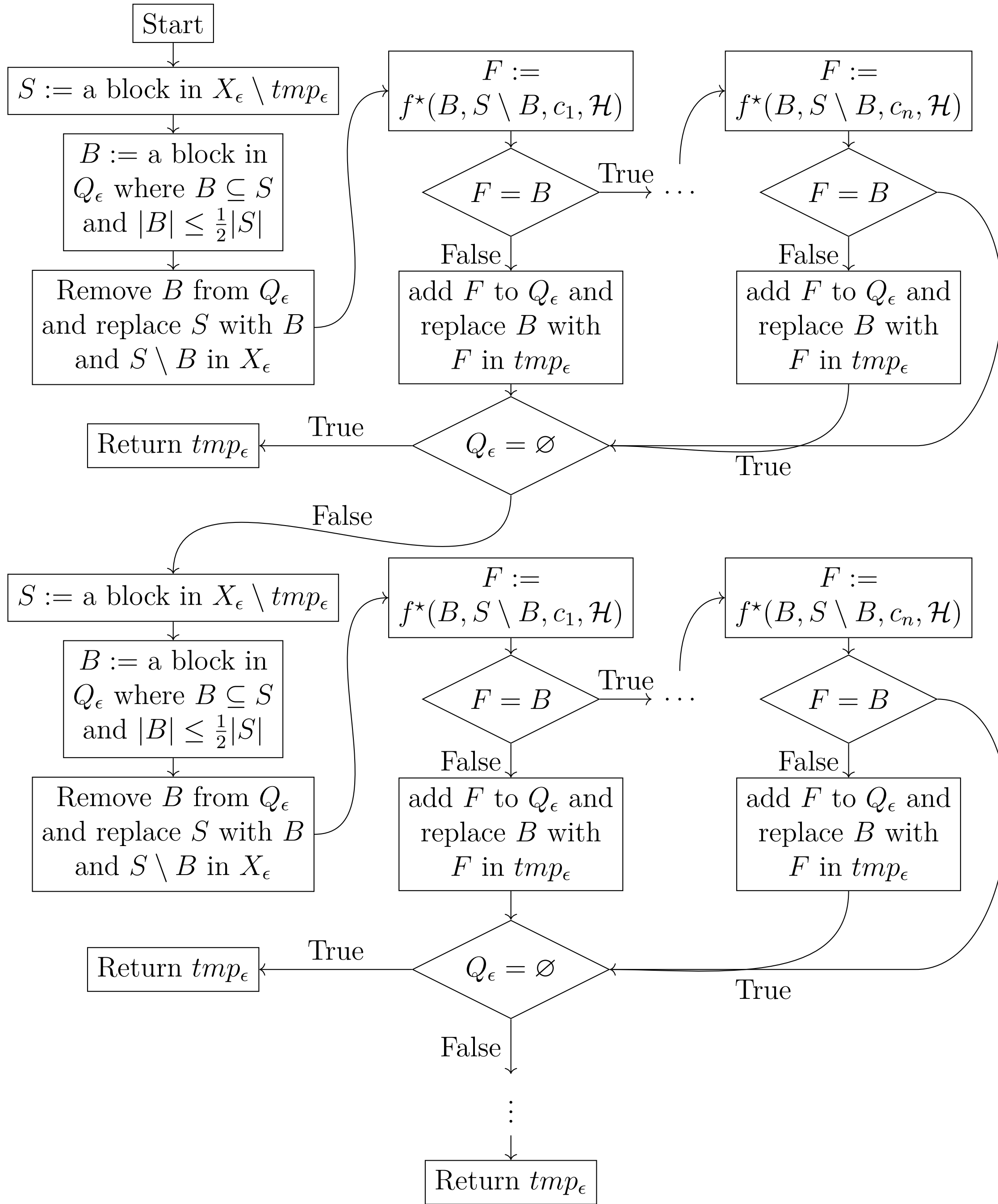

Figure 4.9: DAG of the execution of ‘Refinement’

Specification	Number of agents	Number of states in the central system	Average time taken to optimise using *Alg II* (s)
LTL4simple (A.1)	4	5	0,278
AMBADecompArb3 (A.2)	4	5	0,283
AMBADPA10 (A.3)	7	13	1,284
LilyDemoV18 (A.4)	3	19	0,813
LTL3Arb (A.5)	3	27	1,392
Priority3Arb (A.6)	3	28	1,661
LTL2+3Arb (A.7)	5	84	N/A
Priority2+3Arb (A.8)	5	244	N/A

Table 4.4: Performance of *Alg II*

Specification	Number of states in the central system	Proof of concept (s)	*Alg I* (s)	*Alg II* (s)
LTL4simple (A.1)	5	0,264	0,264	0,278
AMBADecompArb3 (A.2)	5	0,270	0,273	0,283
AMBADPA10 (A.3)	13	1,246	1,242	1,284
LilyDemoV18 (A.4)	19	1,510	1,001	0,813
LTL3Arb (A.5)	27	7,641	1,576	1,392
Priority3Arb (A.6)	28	4,838	1,740	1,661
LTL2+3Arb (A.7)	84	N/A	31.597	N/A
Priority2+3Arb (A.8)	244	N/A	1575,917	N/A

Table 4.5: Performance comparison between *Alg I*, *Alg II* and the proof of concept algorithm

5

Discussion and Conclusion

5.1 Discussion

5.1.1 Selection of Specifications

We chose to use 8 specifications of different sizes and complexities. A majority of them are representations of real problems, such as different types of arbiters. These specifications were taken from the Strix demo website [13]. The central models resulting from the specifications ranged from 5 to 244 states and the number of agents from 3 to 7. Two of them are relatively small and are easy to decompose while the two largest ones are very large and complicated. Only one of the three algorithms, *Alg I*, was able to decompose the largest systems. The other specifications result in up to 28 states and are harder to decompose than the two smallest systems, but easier than the two largest, making differences between the algorithms more visible.

Finding large, but not too large, systems turned out to be difficult. All example specifications solving realistic problems are relatively small and we have included the largest in our testing. The reason for the realistic systems being so small is that the synthesis of going from a specification to a central model is a 2-EXP problem [14]. This makes larger specifications hard to synthesise.

Our next tactic was combining existing specifications to form a larger one. A large majority of these turned out to not be realisable for distributed systems and were thus useless. The ones that were realisable ended up with a number of states ranging between 13 and 666. These specifications are not realistic and are only useful for testing the performance of the algorithms.

5.1.2 Performance

The tests were run on an AMD Ryzen 5 7520U quad-core processor with 8 GB RAM. Running the algorithms on a computer with relatively limited resources works well on the smaller and realistic systems with each execution not taking longer than a few seconds. The two largest systems were decomposable in 30 seconds and 30 minutes respectively using *Alg I*. The other two algorithms could not handle systems of this size.

The memory usage of each algorithm was not a prioritised evalutation factor but measurements were still made. No major difference in memory usage between the algorithms was seen which we are satisfied with.

Each test specification was run 10 times for each algorithm with the exception of Priority2+3Arb (A.8) which was run 5 times due to the long execution time. There were no significant outliers on any tests except Priority2+3Arb (A.8) where the execution times ranged between 22 and 32 minutes. The difference is large but we do not see this as a major problem. Because only *Alg I* could decompose that specification, it was more interesting to show that it *could* be done (and reasonably quickly). Because of this, the exact timing for this specification is not as important.

The execution times of the algorithms for each specification are shown in Table 5.1, Fig. 5.1 and Fig. 5.2. They show that the difference between the algorithms is not noticeable for smaller systems, but the performance of the proof of concept algorithm is much worse as the systems grow larger. The choice of algorithm for the smallest systems does not matter since the decomposition part only forms a small part of the total process. This part increases as the systems become larger and more complicated. Recall that the proof of concept algorithm used in the testing uses partition refinement and is not the original algorithm using relations presented in Section 4.2. We would expect the original one to perform much worse due to the inferior complexity.

Specification	Number of states in the central system	Proof of concept (s)	*Alg I* (s)	*Alg II* (s)
LTL4simple (A.1)	5	0,264	0,264	0,278
AMBADecompArb3 (A.2)	5	0,270	0,273	0,283
AMBADPA10 (A.3)	13	1,246	1,242	1,284
LilyDemoV18 (A.4)	19	1,510	1,001	0,813
LTL3Arb (A.5)	27	7,641	1,576	1,392
Priority3Arb (A.6)	28	4,838	1,740	1,661
LTL2+3Arb (A.7)	84	N/A	31.597	N/A
Priority2+3Arb (A.8)	244	N/A	1575,917	N/A

Table 5.1: Performance comparison between *Alg I*, *Alg II* and the proof of concept algorithm

The reason why *Alg II* performs better than the others on small to larger sized systems but not on enormous systems is due to a required part for the 'process the smaller half' technique to work. This part is, when splitting, we need to find a block that is a result of splitting another block, i.e., a subset, and is less than or equal to half the size of the original block. In the current implementation, this is not a quick operation. As the number of states increase, so does the time taken to find the correct block. This is not an inherent flaw of the algorithm, it is an issue stemming from our implementation. Currently, the different blocks are stored separately, meaning that many blocks have to be checked to find the correct subset. By using some good heuristics or a superior data structure where they are grouped

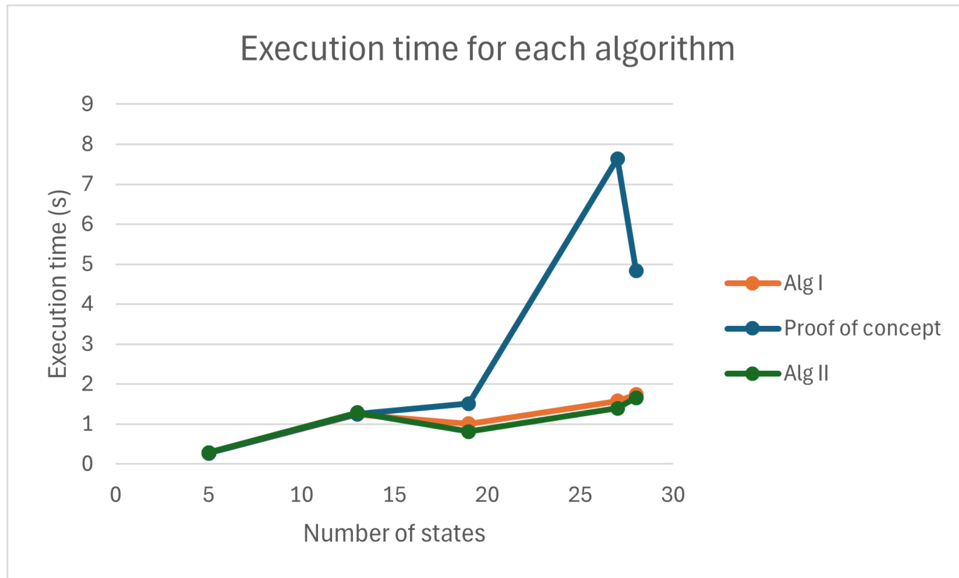

Figure 5.1: Diagram of the execution time for each algorithm for a selection of the tests

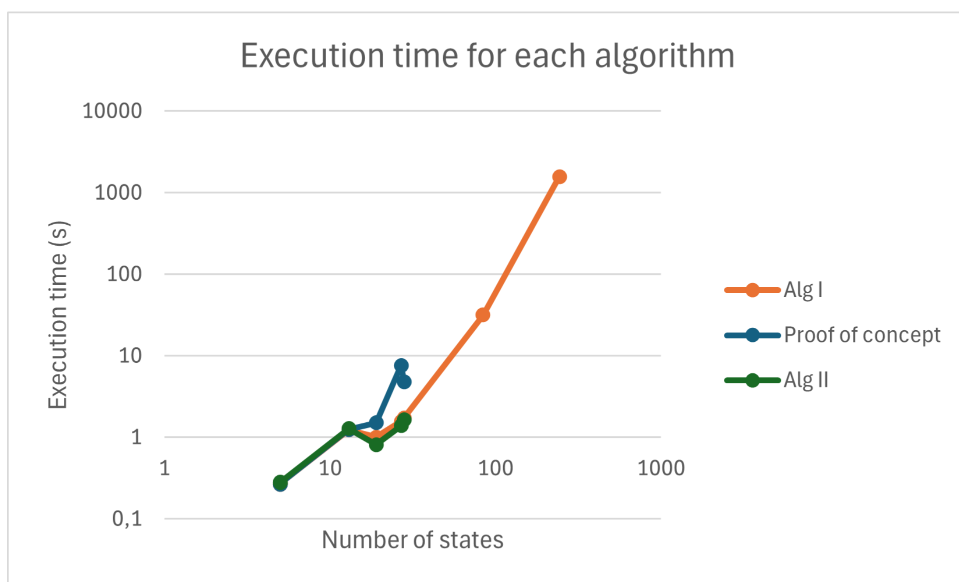

Figure 5.2: A log-log diagram of the execution time for each algorithm for all tests. There are no data points for the proof of concept algorithm and *Alg II* for the two largest systems since they cannot handle systems of that size. The two algorithms tend towards infinity

better, we believe that this can be avoided. We have not explored this further due to time constraints.

This explains the differences between *Alg I* and *Alg II*. For smaller systems, the process of finding a subset is only a minor part of the total execution, where the rest is taken care of by a more efficient refinement algorithm. This helps reducing the total execution time of the algorithm. As the number of states increase, the time taken to find the subsets overtakes the increased performance offered by the 'process the smaller half' part, thus becoming slower than *Alg I*. Due to the difficulties of finding good specifications for testing, we cannot say at what point this crossover occurs which would have been interesting to study.

Another interesting thing to study would have been comparing the execution times to the number of transitions in the TS. The number of transitions is a variable in the complexities of the algorithms, thus contributing to the performance. Unfortunately, we have no method to find the number of transitions in each transition system in a practical way. It might be the explanation of the difference in execution time between the proof of concept algorithm and *Alg I*/*Alg II* when it comes to LTL3Arb (A.5) and Priority3Arb (A.6). The difference is large despite them having almost the same number of states in their central systems.

Another explanation could be that the resulting number of states for each decomposed agent in LTL3Arb is on average larger than in Priority3Arb. When decomposing LTL3Arb, its agents end up with 17, 19 and 23 states while Priority3Arb's agents end up with 9, 16 and 23 states. It could be that since the individual agents in LTL3Arb have a larger number states than Priority3Arb, that more blocks are needed to be split, thus making the required time to decompose LTL3Arb longer than the amount of time to decompose Priority3Arb despite the latter having one more state in the central system.

The largest of the specifications, which is not included in the testing, is Comb3Arb (A.9). It was created by combining LTL3Arb (A.5) and Priority3Arb (A.6) and resulted in a very large TS with 666 states. This specification could not be handled by the hardware used in the rest of the testing so we used one of Chalmers' computer clusters called Vera [15] instead. By using different hardware for this test, it does not give a fair comparison with the other specifications which is why we present it separately. The proof of concept algorithm and *Alg II* did not manage to decompose a single agent in 24 hours. *Alg I* on the other hand managed to decompose the TS for all 6 agents in just over 11 hours (673 minutes) using 32 threads on the cluster. What this shows is that specifications resulting in TSs of this size are still decomposable in a reasonable amount of time, but we are nearing the upper limit. We are not concerned about this though as the realistic specifications we have used in testing result in TSs with fewer than 30 states.

5.1.3 Computational Complexity

Theoretical complexity does not always reflect reality. An example of this is the parity game problem described in Section 3.2 where quasi-polynomial time algorithms

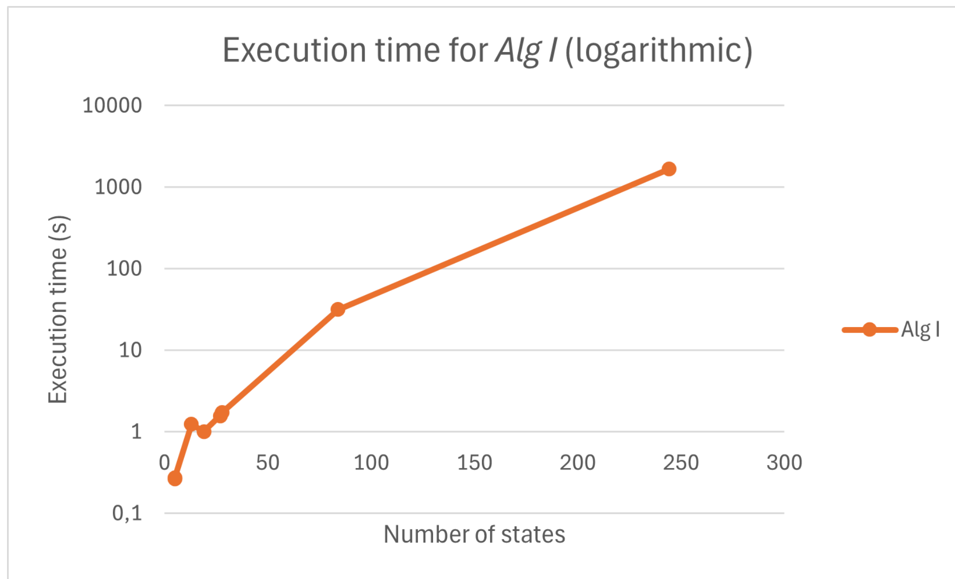

Figure 5.3: Logarithmic diagram of the execution time for *Alg I*

exist but the exponential algorithm is in fact the fastest in practice [2]. The complexity only tells us about the worst possible case, which can be very rare. We are looking for algorithms with a better average performance for practical examples. Thus, we cannot not rule out algorithms based on their complexities, they may perform better despite looking worse theoretically.

The algorithms we propose both have the same complexity, $O(km^2n^4)$ of the parallel versions. Despite this, the performance on large systems vary drastically. *Alg II* is essentially useless due to the flawed implementation while *Alg I* performs beyond expectations. When looking at the logarithmic plot for *Alg I* in Fig. 5.3, the times look exponential in the beginning but tapers off for larger systems, thus showing the polynomial property.

5.1.4 Potential Improvements and Future Work

We believe that the preprocessing for *Alg II* is not optimal and could be enhanced. If the complexity of the preprocessing part (which is currently the deciding factor for the complexity of *Alg II*) were improved, the time complexity of the algorithm as a whole would be enhanced. A performance increase would be expected but not automatically followed due to reasons explained earlier. We believe that it should be possible to reduce the complexity by either performing the preprocessing before the main loop or by removing the need for preprocessing completely. This could lead the time complexity of the algorithm possibly becoming $O(km^2n(\log n)^2)$, which would be a great improvement. These ideas have not been explored due to time constraints. We encourage others to continue this research.

We have only looked at the worst-case complexity in this thesis. Due to the low number of realisable specifications as mentioned in Section 5.1.1, analysing an ‘av-

erage' case would be difficult. Calculating a lower bound was not included in the scope of this thesis either, so we encourage others to explore this area.

As stated earlier, our implementation of *Alg II* does have a flaw when it comes to finding subsets. By utilising different data structures or heuristics, we believe this can be mitigated and thus make the algorithm perform better than *Alg I* in practice.

We do not rule out the possibility of further improvement, but a more substantial improvement of the algorithms would be hard to achieve due to the recursive nature of Definition 2.1.4. If new traditional bisimulation techniques are developed, we expect these to be usable in $\mathcal{E}$-cooperative bisimulation algorithms as well due to the relative similarities between the two problems.

5.2 Conclusion

To help create new distributed systems, we set out to make the decomposition stage of going from a centralised model to individual models for each agent more efficient in accordance with the methods used in the SynTM project. We needed to create new algorithms for this purpose. Using a new technique proposed in the SynTM project called $\mathcal{E}$-cooperative bisimulation, interaction between agents can be minimised while ensuring correctness. Existing bisimulation algorithms such as the Kanellakis-Smolka and Paige-Tarjan algorithms cannot be used for this purpose but some techniques used by them can be implemented.

We proposed two new algorithms using techniques from the above mentioned bisimulation algorithms. They both improve the performance of the decomposition process with the first algorithm being significantly better than previous algorithms. The second one should be even better if the implementation can be improved.

Thanks to our proposed algorithms, problems that were previously not possible to compute in a reasonable amount of time are now computable relatively quickly. The upper limit of what is possible to decompose reasonably quickly has been raised significantly. The synthesis problem of going from a specification to a central model is hard and can only be computed in 2-EXP time with respect to the specification. Thus, the process is still limited to relatively small problems but the decomposition part is now much more efficient.

To allow more complex problems, a hierarchical approach can be taken where the problem is split into smaller chunks that can be synthesised efficiently. The results can be combined later, thus forming a correct and efficient system. This allows larger and more complex distributed systems to be able to be modelled using the SynTM technique.

Bibliography

[1] Y. Abd Alrahman and N. Piterman, 'Correct-by-design teamwork plans for multi-agent systems,' *CoRR*, vol. abs/2301.01257, 2023. DOI: 10.48550/ARXIV.2301.01257. arXiv: 2301.01257. [Online]. Available: https://doi.org/10.48550/arXiv.2301.01257.

[2] P. Parys, 'Parity games: Zielonka's algorithm in quasi-polynomial time,' *arXiv preprint arXiv:1904.12446*, 2019.

[3] L. Aceto, A. Ingolfsdottir, J. Srba *et al.*, 'The algorithmics of bisimilarity.,' *Advanced Topics in Bisimulation and Coinduction*, vol. 52, pp. 100–172, 2012.

[4] A. V. Aho, J. E. Hopcroft and J. D. Ullman, 'The design and analysis of computer algorithms,' 1974. [Online]. Available: https://api.semanticscholar.org/CorpusID:29599075.

[5] P. C. Kanellakis and S. 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*, 1983, pp. 228–240.

[6] R. Paige and R. E. Tarjan, 'Three partition refinement algorithms,' *SIAM Journal on computing*, vol. 16, no. 6, pp. 973–989, 1987.

[7] J. Hopcroft, 'An n log n algorithm for minimizing states in a finite automaton,' in *Theory of Machines and Computations*, Z. Kohavi and A. Paz, Eds., Academic Press, 1971, pp. 189–196, ISBN: 978-0-12-417750-5. DOI: https://doi.org/10.1016/B978-0-12-417750-5.50022-1. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780124177505500221.

[8] CodeHS. 'Fencepost problem.' (), [Online]. Available: https://codehs.com/glossary/term/38 (visited on 16/04/2024).

[9] U. Vishkin, *Thinking in parallel: Some basic data-parallel algorithms and techniques*. [Online]. Available: https://users.umiacs.umd.edu/~vishkin/PUBLICATIONS/classnotes.pdf (visited on 20/05/2024).

[10] Ubuntu. 'Ubuntu manuals: Time.' (), [Online]. Available: https://manpages.ubuntu.com/manpages/jammy/en/man1/time.1.html (visited on 17/05/2024).

[11] Valgrind. 'About valgrind.' (), [Online]. Available: https://valgrind.org/info/about.html (visited on 17/05/2024).

[12] Valgrind. 'Massif: A heap profiler.' (), [Online]. Available: https://valgrind.org/docs/manual/ms-manual.html (visited on 17/05/2024).

[13] Strix. 'Strix demo.' (), [Online]. Available: https://strix.model.in.tum.de/try/ (visited on 02/06/2024).

[14] A. Pnueli and R. Rosner, 'On the synthesis of a reactive module,' in *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of*

Programming Languages, ser. POPL '89, Austin, Texas, USA: Association for Computing Machinery, 1989, pp. 179–190, ISBN: 0897912942. DOI: 10.1145/75277.75293. [Online]. Available: https://doi.org/10.1145/75277.75293.

[15] C3SE Chalmers. 'Vera.' (), [Online]. Available: https://www.c3se.chalmers.se/about/Vera/ (visited on 29/05/2024).

A

Specifications Used in Tests

Most specifications are taken from the Strix demo website [13].

A.1 LTL4simple

```
In(r0,r1,r2,r3) , Out(g0,g1,g2,g3)

Agent A1 -> In(r0) , Out(g0)
Agent A2 -> In(r1) , Out(g1)
Agent A3 -> In(r2) , Out(g2)
Agent A4 -> In(r3) , Out(g3)

Guarantee Sys
G ((!g0 & !g1 & !g2) | !g0 & !g1 & !g3 | !g0 & !g3 & !g2 |
  (!g3 & !g1 & !g2))
G (r0 -> F g0)
G (r1 -> F g1)
G (r2 -> F g2)
G (r3 -> F g3)
```

A.2 AMBADecompArb3

```
In(allready, hbusreq_0, hbusreq_1, hbusreq_2),
Out(decide, busreq, hgrant_0, hgrant_1, hgrant_2)

Agent A1 -> In(allready) , Out(decide, busreq)
Agent A2 -> In(hbusreq_0) , Out(hgrant_0)
Agent A3 -> In(hbusreq_1) , Out(hgrant_1)
Agent A4 -> In(hbusreq_2) , Out(hgrant_2)

Assume Env
G F allready
allready

Guarantee Sys
G ((!hgrant_0 & ! hgrant_1) |
  ((!hgrant_0 | ! hgrant_1) & !hgrant_2))
G (hgrant_0 | hgrant_1 | hgrant_2)
G (!allready -> (X hgrant_0 <-> hgrant_0))
G (!allready -> (X hgrant_1 <-> hgrant_1))
G (!allready -> (X hgrant_2 <-> hgrant_2))
G (hbusreq_0 -> F (!hbusreq_0 | hgrant_0))
G (hbusreq_1 -> F (!hbusreq_1 | hgrant_1))
G (hbusreq_2 -> F (!hbusreq_2 | hgrant_2))
G (hgrant_0 -> (busreq <-> hbusreq_0))
G (hgrant_1 -> (busreq <-> hbusreq_1))
G (hgrant_2 -> (busreq <-> hbusreq_2))
G (!allready -> !decide)
G (decide <-> (!(X hgrant_0 <-> hgrant_0) |
  !(X hgrant_1 <-> (hgrant_1)) | !(X hgrant_2 <-> (hgrant_2))))
G ((!hbusreq_0 & !hbusreq_1 & !hbusreq_2 & decide) -> X hgrant_0)
! decide
hgrant_0
```

A.3 AMBADPA10

```
In(a, b, c, allready, hbusreq_0, hbusreq_1, hbusreq_2),
Out(p0, p1, p2, decide, busreq, hgrant_0, hgrant_1, hgrant_2)

Agent A1 -> In(a) , Out(p0)
Agent A2 -> In(b) , Out(p1)
Agent A3 -> In(c) , Out(p2)
Agent A4 -> In(allready) , Out(decide, busreq)
Agent A5 -> In(hbusreq_0) , Out(hgrant_0)
Agent A6 -> In(hbusreq_1) , Out(hgrant_1)
Agent A7 -> In(hbusreq_2) , Out(hgrant_2)

Assume Env
G F allready
allready

Guarantee Sys
G ((p0 & !p1 & !p2) | (!p0 &  p1 & !p2) | (!p0 & !p1 & p2))
(F G a | G F b) <-> (G F p0 | (G F p2 & !G F p1))
G ((!hgrant_0 & ! hgrant_1) | ((!hgrant_0 | ! hgrant_1) & !hgrant_2))
G (hgrant_0 | hgrant_1 | hgrant_2)
G (!allready -> (X hgrant_0 <-> hgrant_0))
G (!allready -> (X hgrant_1 <-> hgrant_1))
G (!allready -> (X hgrant_2 <-> hgrant_2))
G (hbusreq_0 -> F (!hbusreq_0 | hgrant_0))
G (hbusreq_1 -> F (!hbusreq_1 | hgrant_1))
G (hbusreq_2 -> F (!hbusreq_2 | hgrant_2))
G (hgrant_0 -> (busreq <-> hbusreq_0))
G (hgrant_1 -> (busreq <-> hbusreq_1))
G (hgrant_2 -> (busreq <-> hbusreq_2))
G (!allready -> !decide)
G (decide <-> (!(X hgrant_0 <-> hgrant_0) |
  !(X hgrant_1 <-> (hgrant_1)) | !(X hgrant_2 <-> (hgrant_2))))
G ((!hbusreq_0 & !hbusreq_1 & !hbusreq_2 & decide) -> X hgrant_0)
! decide
hgrant_0
```

A.4 LilyDemoV18

```
In(i0, i1, i2) , Out(a0, a1, a2, a3)

Agent A1 -> In(i0) , Out(a0, a1)
Agent A2 -> In(i1) , Out(a2)
Agent A3 -> In(i2) , Out(a3)

Guarantee Sys
G !(a0 & a1)
G !(a0 & a2)
G !(a0 & a3)
G !(a1 & a2)
G !(a1 & a3)
G !(a2 & a3)
G F i0 -> G F a0
G F i1 -> G F a1
G F i2 -> G F a2
G F a3
```

A.5 LTL3Arb

```
In(r0,r1,r2), Out(g0,g1,g2)

Agent A1 -> In(r0) , Out(g0)
Agent A2 -> In(r1) , Out(g1)
Agent A3 -> In(r2) , Out(g2)

Guarantee Sys
G ((g0 & G !r0) -> (F !g0))
G ((g1 & G !r1) -> (F !g1))
G ((g2 & G !r2) -> (F !g2))
G ((g0 & X (!r0 & !g0)) -> X (r0 R !g0))
G ((g1 & X (!r1 & !g1)) -> X (r1 R !g1))
G ((g2 & X (!r2 & !g2)) -> X (r2 R !g2))
G ((!g0 & !g1) | ((!g0 | !g1) & !g2))
r0 R !g0
r1 R !g1
r2 R !g2
G (r0 -> F g0)
G (r1 -> F g1)
G (r2 -> F g2)
```

A.6 Priority3Arb

```
In(rm, r0, r1) , Out(gm, g0, g1)

Agent Am -> In(rm) , Out(gm)
Agent A0 -> In(r0) , Out(g0)
Agent A1 -> In(r1) , Out(g1)

Assume Env
G F !rm

Guarantee Sys
G ((!gm & !g0) | ((!gm | !g0) & !g1))
G (rm -> X ((!g0 & !g1) U gm))
G (r0 -> F g0)
G (r1 -> F g1)
G ((g0 & X (!r0 & !g0)) -> X (r0 R !g0))
G ((g1 & X (!r1 & !g1)) -> X (r1 R !g1))
G ((gm & X (!rm & !gm)) -> X (rm R !gm))
```

A.7 LTL2+3Arb

```
In(r0,r1,r2, request_0, request_1), Out(g0,g1,g2, grant_0, grant_1)

Agent A1 -> In(r0) , Out(g0)
Agent A2 -> In(r1) , Out(g1)
Agent A3 -> In(r2) , Out(g2)
Agent A4 -> In(request_0) , Out(grant_0)
Agent A5 -> In(request_1) , Out(grant_1)

Guarantee Sys
G ((g0 & G !r0) -> (F !g0))
G ((g1 & G !r1) -> (F !g1))
G ((g2 & G !r2) -> (F !g2))
G ((g0 & X (!r0 & !g0)) -> X (r0 R !g0))
G ((g1 & X (!r1 & !g1)) -> X (r1 R !g1))
G ((g2 & X (!r2 & !g2)) -> X (r2 R !g2))
G ((!g0 & !g1) | ((!g0 | !g1) & !g2))
r0 R !g0
r1 R !g1
r2 R !g2
G (r0 -> F g0)
G (r1 -> F g1)
G (r2 -> F g2)

G ((grant_0 & G !request_0) -> (F !grant_0))
G ((grant_1 & G !request_1) -> (F !grant_1))
G ((grant_0 & X (!request_0 & !grant_0)) -> X (request_0 R !grant_0))
G ((grant_1 & X (!request_1 & !grant_1)) -> X (request_1 R !grant_1))
G (!grant_0 | !grant_1)
request_0 R !grant_0
request_1 R !grant_1
G (request_0 -> F grant_0)
```

A.8 Priority2+3Arb

```
In(rm, r0, r1, request_0, request_1),
Out(gm, g0, g1, grant_0, grant_1)

Agent Am -> In(rm) , Out(gm)
Agent A0 -> In(r0) , Out(g0)
Agent A1 -> In(r1) , Out(g1)
Agent A2 -> In(request_0) , Out(grant_0)
Agent A3 -> In(request_1) , Out(grant_1)

Assume Env
G F !rm

Guarantee Sys
G ((!gm & !g0) | ((!gm | !g0) & !g1))
G (rm -> X ((!g0 & !g1) U gm))
G (r0 -> F g0)
G (r1 -> F g1)
G ((g0 & X (!r0 & !g0)) -> X (r0 R !g0))
G ((g1 & X (!r1 & !g1)) -> X (r1 R !g1))
G ((gm & X (!rm & !gm)) -> X (rm R !gm))

G ((grant_0 & G !request_0) -> (F !grant_0))
G ((grant_1 & G !request_1) -> (F !grant_1))
G ((grant_0 & X (!request_0 & !grant_0)) -> X (request_0 R !grant_0))
G ((grant_1 & X (!request_1 & !grant_1)) -> X (request_1 R !grant_1))
G (!grant_0 | !grant_1)
request_0 R !grant_0
request_1 R !grant_1
G (request_0 -> F grant_0)
G (request_1 -> F grant_1)
```

A.9 Comb3Arb

```
In(ar0, ar1, ar2, rm, r0, r1), Out(ag0, ag1, ag2, gm, g0, g1)

Agent A1 -> In(ar0) , Out(ag0)
Agent A2 -> In(ar1) , Out(ag1)
Agent A3 -> In(ar2) , Out(ag2)
Agent A4 -> In(rm)  , Out(gm)
Agent A5 -> In(r0)  , Out(g0)
Agent A6 -> In(r1)  , Out(g1)

Assume Env
G F !rm

Guarantee Sys
G ((ag0 & G !ar0) -> (F !ag0))
G ((ag1 & G !ar1) -> (F !ag1))
G ((ag2 & G !ar2) -> (F !ag2))
G ((ag0 & X (!ar0 & !ag0)) -> X (ar0 R !ag0))
G ((ag1 & X (!ar1 & !ag1)) -> X (ar1 R !ag1))
G ((ag2 & X (!ar2 & !ag2)) -> X (ar2 R !ag2))
G ((!ag0 & !ag1) | ((!ag0 | !ag1) & !ag2))
ar0 R !ag0
ar1 R !ag1
ar2 R !ag2
G (ar0 -> F ag0)
G (ar1 -> F ag1)
G (ar2 -> F ag2)
G ((!gm & !g0) | ((!gm | !g0) & !g1))
G (rm -> X ((!g0 & !g1) U gm))
G (r0 -> F g0)
G (r1 -> F g1)
G ((g0 & X (!r0 & !g0)) -> X (r0 R !g0))
G ((g1 & X (!r1 & !g1)) -> X (r1 R !g1))
G ((gm & X (!rm & !gm)) -> X (rm R !gm))
r0 -> F !ar0
```