DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

A Tool for Verification and Simulation of Population Protocols

Philip Offtermatt

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

A Tool for Verification and Simulation of Population Protocols

Ein Werkzeug zur Verifikation und Simulation von Populationsprotokollen

Author:	Philip Offtermatt
Supervisor:	Univ.-Prof. Dr. Dr. h.c. Javier Esparza
Advisor:	M. Sc. Stefan Jaax, Dr. Michael Blondin
Submission Date:	16.08.2017

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, .7.2017 Philip Offtermatt

Acknowledgments

I am deeply thankful to my advisers Michael Blondin and Stefan Jaax for their patience and dedication. This thesis would not have been such a fruitful learning experience for me without them.

Abstract

Population protocols are a model for distributed systems of tiny agents. These tiny agents are indistinguishable and one can imagine them as floating around aimlessly, with interactions happening when two agents meet. While individual agents possess only very limited memory and computational capacity, the system as a whole is capable of performing classical distributed tasks such as leader election and majority voting. In this thesis, we present a tool that is capable of simulating and verifying population protocols. In two case studies, we demonstrate how the tool can be used to implement, verify and compare existing and newly devised protocols.

Contents

Acknowledgments **iii**

Abstract **v**

1 Introduction **1**

2 Preliminaries **3**

2.1 Graphs 3

2.2 Markov Chains 3

3 Population Protocols **5**

3.1 Motivation 5

3.2 Definitions 5

3.3 Examples of Population Protocols 9

3.3.1 Threshold 9

3.3.2 Remainder 10

3.3.3 Majority 10

3.3.4 Flock of Birds 12

4 Fairness **15**

4.1 Global Fairness and Variations 15

4.2 Fairness and Probability Distributions 17

4.3 A Necessary and Sufficient Criterion for Fairness 18

5 A Tool for Population Protocols **21**

5.1 Data Structures for Populations and Protocols 21

5.2 Generating a Protocol 22

5.3 Generating a Population 23

5.4 Simulating a Protocol 23

5.5 Verifying a Protocol 25

5.6 Using a Model Checker to Analyse Convergence Properties 25

5.6.1 Arbitrary Probability Distributions 27

5.6.2 Uniform Rules Scheduling 27

5.6.3 Specifying Properties in PRISM 30

5.7 Finding Strongly Connected Components of the Reachability Graph . . . 31

6 Case Study **35**
6.1 Simulating established Protocols 35
6.1.1 Majority Problem 35
6.1.2 3-state Majority 38
6.2 A new Protocol for Flock of Birds 41
6.2.1 The *Prime-Flock-Protocol* 41
6.2.2 Checking Well-Specification 42
6.2.3 Proof of Correctness 43
6.2.4 Comparison of Convergence Properties with Existing Protocols . . 48

7 Conclusion and Future Work **51**

List of Figures **53**

Bibliography **55**

1 Introduction

Population protocols are a model for distributed networks of tiny, passively-mobile agents. It differs from other models for distributed networks in that agents are assumed to have very low memory and computational power. They consist of anonymous agents and a transition relation between those agents. At each point in time, every agent is in one of finitely many states. One can imagine the agents as wandering around blindly. Periodically, two agents bump into each other. When they do, they interact and update their states according to the transition relation. Moreover, there are two essential properties of the model:

- agents are anonymous: when two agents meet, the transition can only depend on the states of the two agents, as there are no other identifying features.
- agents are passively-mobile: agents have no way of influencing when and with which other agent they interact.

For example, population protocols can be used to simulate sensors that are carried by birds, and the population protocol model has some similarity to chemical reactions [14]. Population protocols are in fact capable of solving many classical tasks in distributed computing such as leader election [2] and majority voting [1, 4].

In 2004, the population protocol model was first introduced in a preliminary version by Angluin, Aspnes, Diamadi, Fischer and Peralta in [2]. In the years before that, a similar probabilistic model to study trust in a social network was introduced by Diamadi and Fischer in [13]. In [3], a model of urn automata is presented, which also shares some similarity with the population protocol model.

Much work was focused on characterizing predicates computable by population protocols. By now, it is known that the computable predicates are exactly the semilinear predicates [6]. These are equivalent to the predicates that can be defined in Presburger arithmetic [6, 17]: the first-order theory of natural numbers with addition and order.

While early work relied on a fairness assumption which ensures that possible interactions cannot be avoided forever, more recent work [12, 5] introduces underlying probability distributions that probabilistically determine which interactions happen between agents in each step. This allows to quantify the expected number of steps needed to solve a task or compute a function. Angluin, Aspnes and Eisenstat give measures of the expected number of steps for a special class of protocols in [5]. There has also been work on finding lower bounds for the number of steps needed for various tasks, such as for majority voting in [1].

While there has been much work on population protocols, there are few tools that are capable of simulating and verifying them. In [20], Sun, Liu, Dong and Chen give a model checker that can be used to verify correctness of protocols for fixed sizes. Clément, Delporte-Gallet, Fauconnier and Sighireanu use the model checkers SPIN and PRISM for the same purpose of verifying protocols for fixed sizes [12]. In [11] Chatzigiannakis, Michail and Spirakis present the tool bp-ver, a dedicated tool for verification of population protocols. Another tool exists in peregrine, which Blondin, Esparza, Jaax and Meyer use to verify correctness of protocols for all sizes [10], but only for a subclass of protocols.

While a few tools allow verification of protocols, none that we know of are capable of simulating protocols using some underlying probability distribution, although this can be a useful way to obtain a measure of performance of protocols. We present in this thesis a new tool that for the simulation and verification of population protocols. It differs from existing tools in that it allows storing and manipulating population protocols.

One can specify a population protocol in the tool, and simulate executions of it, starting at a given initial population of agents. The tool also provides a way of verifying correctness of a protocol for a given initial population. Additionally, computations of the protocol, using a given probability distribution, can be simulated and key statistics such as the average number of steps until convergence can be extracted. In addition, an export of protocols to the model checker PRISM is supported.

It is easy to see that the probability distribution that determines the interactions has a large impact on convergence times. We investigate the question of how probability distributions and convergence times are connected, as it can be helpful in understanding which classes of protocols have especially fast convergence times.

The goal of this thesis is to describe the tool, and demonstrate its usefulness. We use it to measure the average number of steps until convergence for protocols from the literature. In addition, we propose a new protocol, called prime-flock-protocol. We use the tool to verify correctness of the prime-flock-protocol up to a certain size. As finding a correct protocol can be non-trivial, the tool can be used to easily implement a protocol that one suspects to be correct and give some first confidence in its correctness. For completeness, we also give a formal proof for the correctness of the prime-flock-protocol. In addition, we use the tool to measure the average number of steps to convergence of protocols from the literature. This can give an idea of how efficient protocols are under given probability distributions, and we can compare other protocols to the new prime-flock-protocol.

Outline. The rest of this thesis is structured as follows. We give some preliminary definitions in Chapter 2, and then define population protocols in Chapter 3. In Chapter 4, we give different criteria for fairness and explain how Markov chains can be used to describe protocols. In addition, we investigate the connection between probability distributions and fairness. We describe the tool we developed in Chapter 5, and present two case studies for the tool in Chapter 6. In Chapter 7, we give our conclusions and discuss avenues for future research and development of the tool.

2 Preliminaries

A *multiset* M over a set S is a mapping $S \to \mathbb{N}$, where each $s \in S$ is assigned a multiplicity. Informally, a multiset is a set allowing multiple occurrences of the same element. $|M|_s = M(S)$ denotes the *multiplicity* of s in M. Denote by $|M| := \sum_{s \in S} |M|_s$ the total number of elements in M. Note that we can naturally treat a set K as a multiset by having that $\forall s : K(s) = 1$ if $s \in K$ and $K(s) = 0$ otherwise. Let O be a multiset, then $M \backslash O$ denotes the multiset M' where for all $e \in S$, $|M'|_e = max(0, |M|_e - |O|_e)$.

By $pop(S)$ we denote the set of all multisets over the set S.

2.1 Graphs

Formally, a finite graph is a pair $G = (V, E)$, where:

- V is a finite set of nodes, and
- $E \subseteq V \times V$ is a set of edges, such that $(v, w) \in E$ means that there is an edge from v to w. We also write $v \to w$ for this.

Note that for our purposes, graphs are considered to be directed. Therefore it may be the case that $(v, w) \in E$ but $(w, v) \notin E$.

We say that there is a *path* from v to w if $v = w$ or there is a sequence of nodes $x_1 x_2 ... x_n$ such that $v \to x_1 \to ... \to x_n \to w$.

A set $C \subseteq V$ is a *strongly connected component (SCC)* of G if C is a maximal set such that for all nodes $v, w \in C$ there is a path from v to w and a path from w to v. A *bottom strongly connected component (BSCC)* of G is then an SCC of G from which there are no outgoing edges. More formally, an SCC C is a BSCC if $\forall v \in C, \forall w \in V : v \to w$ implies $w \in C$. We can also lift the reachability relation to SCCs, i.e. for two SCCs C and D, $C \to D$ iff $\exists v \in C, \exists w \in D : v \to w$.

The *topological ordering* of the SCCs of a graph G is a relation $<$ over the SCCs of G such that for any two SCCs C and D with $C \neq D$: $C \to D \leftrightarrow C < D$.

2.2 Markov Chains

A *Markov chain* is a tuple $M = (S, P, \iota_{init})$ where:

- S is a set of finitely many *states*

- P: $S \times S \rightarrow [0,1]$ is the *transition probability distribution*, where $P(s,s') = \rho$ means that when in state s, the probability of moving to state s' is given as ρ, and
- $\iota_{init} : S \rightarrow [0,1]$ is the *initial distribution*, where $\iota_{init}(s) = \rho$ means that the probability of the initial state being s is ρ.

For $P(s,s')$ we also sometimes write $P(s'|s)$. Note that P and ι_{init} are probability distributions, therefore it must be the case that $\sum_{s \in S} \iota_{init} = 1$ and for all states s, it holds that $\sum_{s' \in S} P(s'|s) = 1$.

A *run* in a Markov chain is an infinite sequence of states $R = s_0 s_1 s_2 \cdots$. To denote the state at index i in the Markov chain, we use R_i. We denote by $inf(R)$ the largest set of states that occur infinitely often in R.

In the scope of this thesis, the definitions we have given so far for Markov chains will be sufficient. We refer to [9] for much more information.

.

3 Population Protocols

3.1 Motivation

Population Protocols are models that can be used to describe distributed computing. They consist of a finite number of agents, with each agent having only a limited amount of computational power and storage capacity. In the canonical example for such a system, the agents are sensors that are attached to birds. Each small sensor is only capable of storing a small amount of data and receiving and sending communication over short distances. When two birds get close to each other, the sensors can communicate, and change their states according to their current states. Even though the sensors are not capable of controlling when and with which other sensor they interact and only have minimal storage, the system as a whole is still capable of performing significant tasks under certain constraints. Among others, tasks such as leader election [2], majority voting [1, 4, 8], threshold predicates and remainder predicates [7] can be solved.

3.2 Definitions

In population protocols, computations take place in discrete-time steps. At every point in time, each agent assumes one state out of a finite set of states. When two agents meet, a transition function determines how the agents will change their states, depending on their states at the time of meeting. Another function determines what input (e.g. real-world data gathered from a sensor) corresponds to which initial state in the protocol. Lastly, a function is needed to map states of the protocol to their output value.

Formally, a *protocol* P is defined as a tuple $(Q, \Sigma, \iota, \delta, \omega, Y)$ where:

- Q is a finite set of states
- Σ is a finite input alphabet
- $\iota : \Sigma \to Q$ is a mapping from inputs to states
- $\delta \subseteq Q^2 \times Q^2$ is a transition relation, with transitions of the form (a_0, b_0, a_1, b_1) meaning that a pair of agents in states a_0 and b_0 can interact to change their states to a_1 and b_1.
- $\omega : Q \to Y$ is a mapping from states to the output range Y and
- Y is a finite set of outputs

Generally, we will not give Y explicitly, since it is sufficient to give ω, as one can simply deduce Y from the image of ω under Q.

We will consider a protocol called *4-state majority* as an example to better illustrate the concepts of population protocols. This protocol is also given in [8]. In the beginning, each agent in the system is either coloured red (R) or blue (B), and the protocol determines whether the number of red agents is at least as high as the number of blue agents. This protocol is given as follows: The input alphabet is $\Sigma = \{R, B\}$, and the states are given as $Q = \{R, B, 1, 0\}$. The input relation ι is then given as $\iota = id$, the identity relation. The output function ω maps states R and 0 to 1 and states B and 1 to 0. The protocol has four transitions:

$$\begin{gathered} R, B \to 0, 1 \\ R, 1 \to R, 0 \\ B, 0 \to B, 1 \\ 0, 1 \to 0, 0 \end{gathered}$$

Intuitively, the protocol works by having agents in states B and R interact via the transition $R, B \to 0, 1$ until there are only agents in states 0, 1 and either R or B. From there on, agents in states 0 and 1 will then be converted by the remaining agents in states R or B, until only agents in states B and 1 or R and 0 remain.

Note that as is the case in the 4-state majority protocol, transition functions are not necessarily symmetrical, i.e. a_0, b_0 might not have the same resulting states as b_0, a_0.

To highlight this asymmetry, for a transition $a_0, b_0 \to a_1, b_1$ we call a_0 the *initiator* and b_0 the *responder*.

In addition, transitions can be nondeterministic, i.e. there can be two transitions $t = a_0, b_0 \to a_1, b_1$ and $t' = a_0, b_0 \to a_2, b_2$ in the protocol. Similarly, there might not be any transition for a given pair a_0, b_0. It is then assumed that there is a *silent transition* $a_0, b_0 \to a_0, b_0$. For example, in the 4-state majority protocol, since there is no transition given for the pair R, R, we assume that there is a transition $R, R \to R, R$.

Let $t = a_0, b_0 \to a_1, b_1$ be a transition, then we denote the *pre* of t as $pre(t) = (a_0, b_0)$. Similarly, the *post* of t is given by $post(t) = (a_0, b_1)$.

To describe our system, we need to describe which state each agent is in. Since the agents in the system are anonymous, only the amount of agents in each state is important. Therefore, it is sufficient to use a multiset to describe the system at a certain point in time. We call a multiset of the states of the agents in the system a *configuration* or *population*. An example for a configuration in the 4-state majority protocol would be $\{|R, R, B, 0, 1|\}$, where there are two agents in state R, and one agent in state 0, B and 1 respectively.

We call the states in the image of ι under Σ *initial states*. By *initial configuration*, we denote a configuration that only contains initial states. We denote the set of initial configurations of a protocol P by P_{init}. Initial states of the 4-state majority protocol are only R and B, initial configurations are those that contain any amount of R's and B's, but no 1's and 0's.

A transition $a_0, b_0 \to a_1, b_1$ is called *enabled* in a configuration C if $|C|_{a_0} \geq 1$ and $|C|_{b_0} \geq 1$ if $a_0 \neq b_0$, or $|C|_{a_0} \geq 2$ otherwise.

Let C and C' be configurations, and let $t = a_0, b_0 \to a_1, b_1$ be a transition. We say that applying t to C results in C' if $(C \backslash \{|a_0, b_0|\}) \cup \{|a_1, b_1|\} = C'$. To denote that C' is reachable from C via t, we also write $C \xrightarrow{t} C'$. Note that $\xrightarrow{t}$ defines a transition relation over configurations.

We can naturally extend the relation $\xrightarrow{t}$ to include configurations reachable via all transitions as follows: $\to := \bigcup_t \xrightarrow{t}$. For two configurations C and C', we say that C' is reachable from C in one step if $C \to C'$.

By $\xrightarrow{*}$, we denote the reflexive transitive closure of $\to$. For two configurations C and C', if $C \xrightarrow{*} C'$, we say that C' is reachable from C.

A *computation*, *run* or *execution* is an infinite sequence $C_0C_1C_2...$, where for all $n \geq 0$, $C_n \to C_{n+1}$. As an example, the run

$$\{|R, B, B|\}\{|0, 1, B|\}\{|0, 0, B|\}\{|0, 1, B|\}\{|0, 0, B|\} \cdots$$

would be a run in the 4-state majority, since $\{|R, B, B|\}$ is a configuration in which transition $R, B \to 0, 1$ is enabled, which leads to $\{|0, 1, B|\}$, from which $0, 1 \to 0, 0$ is enabled, which then again leads to $\{|0, 0, B|\}$, from which $B, 0 \to B, 1$ is enabled, which leads back to $\{|0, 1, B|\}$, and so on. We denote the set of all runs of a protocol P by $Paths(P)$.

Let π be an execution. Then by $\pi(i) = C$, we denote that the i-th configuration in the execution is C.

We say a configuration C is in a *consensus* if:

$$\exists y \in Y : \forall s \in C : \omega(s) = y$$

We say that a configuration that is not in a consensus is in a *dissensus*. Let C be a configuration in a consensus. Then we denote by $O(C)$ the output value of the configuration. Formally, $O(C) = y$ if $\forall s \in C : \omega(s) = y$. Note that the output value of C is undefined when C is in a dissensus.

For the 4-state majority, we have that R and 0 give output 1, and B and 1 give output 0. Therefore, all configurations that either contain only R's and 0's or B's and 1's are in a consensus. For example, the configuration $\{|1, 1, B|\}$ is in a consensus with $O(\{|1, 1, B|\}) = 0$. On the other hand, configuration $\{|0, 1, B|\}$ is in a dissensus, as states B and 1 have an output of 0, while state 0 has an output of 1.

Since computations are infinite, instead of requiring termination, we look at *convergence* of the protocol. We say that a computation π converges to an output value y if there exists $i \geq 0$ so that for all $j \geq i$, it holds that $O(\pi(j)) = y$. We write $O(\pi) = y$ for this. This means transitions will still occur, and the states of agents can still change, but their output values must remain the same. We also say that the computation has reached a *lasting consensus*.

Let P be a protocol. A *stable configuration* is a configuration C such that:

$$\exists y \in Y : O(C) = y \wedge \forall \pi \in Paths(P) : \forall i \geq 0 : \pi(i) = C \rightarrow \forall j \geq i : O(\pi(j)) = y$$

Informally, when an execution reaches a stable configuration with output value y, then at that point, the execution has also converged to y. As an example, the configuration $\{|1, 1, B|\}$ is a stable configuration in the 4-state majority. The only enabled transitions are transitions involving only agents in states 1 and B, and none of these change the states of the involved agents. Therefore, the output will also not change, and the consensus is lasting.

To define correctness of a protocol, a notion of fairness is needed. Without fairness, there might be an execution where only two agents interact, while the rest of the agents does not take part. This means that no meaningful computation that depends on all agents can take place. For example, consider the execution

$$\{|R, B, B|\}\{|0, 1, B|\}\{|0, 1, B|\} \cdots$$

, that initially uses the transition $R, B \rightarrow 1, 0$ to obtain $\{|0, 1, B|\}$, and then only uses silent transitions (for example, pairs the agents B and 0) although the transitions $B, 1 \rightarrow B, 0$ and $0, 1 \rightarrow 0, 0$ are enabled in every step.

We call an execution π *fair* if for all configurations C and C', if C appears infinitely often in π and $C \rightarrow C'$, then C' appears infinitely often in π. Informally, if a configuration can be reached infinitely often, then it should also be reached infinitely often. This would mean the aforementioned example of

$$\pi = \{|R, B, B|\}\{|0, 1, B|\}\{|0, 1, B|\} \cdots$$

would not be a fair execution, since from $\{|0, 1, B|\}$, which occurs infinitely often in π, $\{|0, 0, B|\}$ is reachable, but does not occur infinitely often in π.

This gives rise to the definition of *well-specification*. Informally, well-specification means the protocol will converge to the same output value in all fair executions on the same input. More formally, a protocol is well-specified if for all initial configurations C, there exists $y \in Y$ such that for all executions π, if $\pi(0) = C$, then $O(\pi) = y$.

Closely related to well-specification is the notion of the *computed function* of a protocol. Let i be the extension of the input function ι to multisets, where ι is applied to each element in the multiset. We say that a protocol computes a function $f : pop(\Sigma) \rightarrow Y$ if for every multiset of inputs $M \in pop(\Sigma)$, all fair executions π such that $\pi(0) = i(M)$, the protocol reaches a lasting consensus with value $f(M)$.

A *terminal configuration* is a configuration from which only silent transitions are enabled, i.e. once a terminal configuration is reached, the system will stay in that configuration. For example considering again 4-state majority, $\{|1, 1, B|\}$ is a terminal configuration, but $\{|0, 1, B|\}$ is not, since in the second case, the non-silent transitions $B, 0 \rightarrow B, 1$ and $0, 1 \rightarrow 0, 0$ are enabled.

We call an execution that reaches a terminal configuration a *silent execution*, since after reaching the terminal configuration, only silent transitions can occur.

Similarly, a *silent protocol* is one such that all fair executions starting from any initial configuration are silent.

3.3 Examples of Population Protocols

To illustrate the concept of population protocols further, the following section is dedicated to introducing several classes of problems than can be solved using population protocols.

As mentioned above, the predicates computable by population protocols are precisely those definable in first-order Presburger arithmetic. The two following protocols are especially important for this, as these are the basic building blocks from which all other predicates computable by population protocols can be derived from them. This can be done using the fact that population protocols are closed under conjunction and negation. For more information on this, see [6].

3.3.1 Threshold

One class of problems that can be solved using population protocols are *threshold predicates* of the form $a_1x_1 + a_2x_2 + ... + a_dx_d < c$, with parameters $a_1, ..., a_d, c \in \mathbb{Z}$, and $x_1, ..., x_d \in \mathbb{N}$ being the input variables.

One protocol to compute threshold predicates is described by Angluin, Aspnes, Diamadi, Fischer and Peralta in [2].

To describe the protocol, first define the state space $Q = \{0,1\} \times \{0,1\} \times \{u \in \mathbb{Z} : -s \leq u \leq s\}$, where $s = max(|c| + 1, max_i|a_i|)$. Each state has 3 components, with the first component indicating whether the state is a "leader", the second component indicating the output of the state, and the third component indicating the value of the state. The input function $\iota(x_i)$ gives x_i agents with initial state $(1, 0, a_i)$. Define

$q(u, u') = max(-s, min(s, u + u'))$

$r(u, u') = u + u' - q(u, u')$

$b(u, u') = 1$ if $q(u, u') < c$, else 0

Then the transitions of the protocol are given as:

$$(l, x, u), (l', x, u') \to (1, b(u, u'), q(u, u')), (0, b(u, u'), r(u, u'))$$

with $u, u' \in [-s, s]$, $(l, l') \in \{(0, 1), (1, 1), (1, 0)\}, x \in Q$

Intuitively, when two agents meet and at least one of them is a leader, exactly one of the two agents will be a leader after the transition takes place. Therefore, whenever two leaders meet, the number of leader decreases by one. In addition, the leader will collect the values of both agents, up to the upper and lower limits of s and $-s$ respectively. Then both agents adopt the output of the leader. Between two non-leaders, only silent transitions occur.

In an initial configuration, every agent is a leader. Transitions involving two leaders will reduce the number of leaders, until there is only one leader left. That leader will then have collected enough information to have an accurate output in the second component, and will then convert all agents to that output. For a more detailed description of the protocol, including a correctness proof, see [2].

3.3.2 Remainder

Remainder predicates are of the form $a_1x_1 + a_2x_2 + ... + a_dx_d \equiv c$ (mod m). A protocol to compute remainder predicates is given in [2], and adopted here.

The protocol is similar to the protocol for threshold predicates. We define $b = 1$ if $u + u' = c$ (mod m), else 0. The transitions are given by

$$(l, *, u), (l', *, u') \rightarrow (1, b, (u + u') \bmod m), (0, b, 0)$$

with $u, u' \in [m, -m], (l, l') \in \{(0, 1), (1, 0), (1, 1)\}$.

The input function is defined as $\iota(x_i) = (1, 0, x_i \cdot a_i \bmod m)$. This is a slight variation to the protocol given in [2], where there are initially x_i many agents in state $(1, 0, a_i)$.

As with threshold, the first component is a leader bit, the second component stores the current output of the agent, and the last component holds the value of the agent.

The intuition is similar to the protocol for threshold, where one agent collects the values of both agents, whereas the other agent is assigned 0, and the number of leaders will reduce, until there is only one leader left that also holds the collective value (mod m) and can therefore decide whether the predicate is true.

The output function is given as $\omega((*, b, *)) = b$ and maps agents to their output bits. We again refer to [2] for a more detailed description.

3.3.3 Majority

While Threshold and Remainder protocols can be used as the building blocks to obtain a protocol for every predicate definable in Presburger arithmetic, this generally does not yield in the fastest or most efficient protocol. For this reason, there are many dedicated protocols for special cases. One such special case is the majority predicate. The aforementioned 4-state majority protocol is one protocol able to compute the predicate.

A formal definition of the predicate is given as follows: Define the input Σ as $\{R, B\}$ and the output range Y as $\{0, 1\}$. The function we want to compute is then given as $f : pop(\Sigma) \rightarrow Y$, where $f(X) = 1$ if $|X|_R \geq |X|_B$ else 0, i.e. the function returns 1 if initially, there are at least as many R's as B's in the system, otherwise it returns 0.

Apart from the 4-state majority protocol, there are more protocols that compute the majority predicate.

3-state Majority

A protocol that has especially fast convergence, but is only approximate (in that there are fair runs that converge to the wrong value) is given in form of the *3-state majority* protocol, as introduced by Angluin, Aspnes and Eisenstat in [4].

As always for the majority protocol, we assume $\Sigma = \{R, B\}$. We then have $\iota(R) = x, \iota(B) = y$.

The transitions of the protocol are best given as a matrix. Reading line i, column j with entry (k,l) as $i, j \to k, l$, the matrix looks like this (adopted from [4]):

$$\begin{array}{cccc} & x & b & y \\ x & (x,x) & (x,x) & (x,b) \\ b & (b,x) & (b,b) & (b,y) \\ y & (y,b) & (y,y) & (y,y) \end{array}$$

The basic idea is that there are undecided agents in the blank states b, and decided agents in states x and y. When x or y initiate an interaction with b, they convert that agent to their respective state, whereas when they initiate an interaction with an agent in the opposing state, they reduce that agent to state b.

The output function ω then maps x to 1 and y and b to 0. The choice for b does not actually matter, since if there are only agents in state b and either x or y, then the agents in state b will eventually be converted to x or y respectively. In addition, the case of a full configuration of agents in state b is not possible, since the last agent in state x or y will never be converted to b.

The protocol is only approximately correct. Consider the case where all but one agents begin in state x, with the one spare agent beginning in state y. Then consider the fair execution where transitions only happen with y as the initiator, which then converts all other states to y, first from x to b, and then from b to y. This execution converges to the wrong output. As seen from this example, the protocol is only approximate, in that not all fair executions reach a lasting consensus with the correct value. However, assuming a probabilistic scheduler that chooses pairs of agents to interact based on some probability distribution (for example, picking uniformly at random among agents), the protocol has a high probability to converge to the right output for a given input, as long as the majority is sufficiently large [4].

Average and Conquer

Another protocol that can provide an exact solution to the majority problem is the *Average and Conquer* (AVC) protocol. It was proposed by Alistarh, Gelashvili and Vojonović in [1].

The AVC protocol depends on two parameters, an odd integer $m > 0$ and an integer $d > 0$. There are multiple subsets of states for the protocol:

Strong States: $\{-m, -m+2, ... -3, 3, ..., m-2, m\}$

Intermediate States: $\{-1_1, -1_2, ..., -1_d, 1_1, 1_2, ...1_d\}$

Weak States: $\{-0, +0\}$

Each state has a sign (± 1), and a weight, (0 or an odd integer up to m). The intuition behind the states is that the sign determines which output the state currently has, whereas the weight determines the level of confidence in the output.

We will not go into full detail on the transitions (for a more in-depth explanation of the protocol, see [1]), but the basic principle is that when two states meet either

- a strong state and a non-weak state meet, in which case the states average their values (and if the average is even, round the weight of one agent up and that of the other down);
- a weak state meets a non-weak state, in which case the weights remain the same, but the weak state changes its sign to match that of the non-weak state;
- Two intermediate states meet, in which case, if one of the states is either -1_d or 1_d, both states are downgraded to 0, otherwise both states are downgraded from $+1_i$ to $+1_{i+1}$ (or respectively from -1_i to -1_{i+1}).

The output function depends only on the sign, where $\omega(s) = 1$ if $sign(s) = +1$, $\omega(s) = 0$ otherwise. The input function ι maps R to $+m$ and B to $-m$.

Compared with the 4-state majority protocol, the AVC protocol needs more states, but has the advantage of faster convergence.

3.3.4 Flock of Birds

A special case of threshold predicates is known as the *flock-of-birds* predicate. The name comes from the original scenario the protocol was proposed for. Consider a flock of birds, where each bird either has normal or elevated temperature, and fix some natural number N. The question is: are there more than N birds with elevated temperature?

For all of the following protocols, we have $\Sigma = pop(\{0,1\})$, with 0 indicating a bird with normal temperature, and 1 a bird with elevated temperature. In addition, the output range is given as $Y = \{0,1\}$. The predicate can then be given as $f(X) = 1$ if $|X|_1 \geq N$, else 0.

$\mathbf{flock_i}$

This protocol was introduced in [11] by Chatzigiannakis, Michail and Spirakis. Consider for the states $Q = \{0, 1, ..., N\}$. The input function ι is given by $\iota(0) = 0, \iota(1) = 1$, and the output function $\omega(s) = 1$ if $s = N$ otherwise 0.

The transitions are defined by

$$\begin{aligned} k, j &\to k + j, 0 && \text{if } k + j < N \\ k, j &\to N, N && \text{else} \end{aligned}$$

for $k, j \in Q$.

Intuitively, whenever two agents interact, the values of the agent's states are collected into one of the agents, whereas the other is reduced to state 0. When one agent reaches state N, it will then convert all other agents to that state.

Note that the amount of states is linear in N.

Threshold(N)

The Threshold(N) protocol is very similar to the $flock_i$ protocol, but uses a different transition function. It was proposed by Clément, Delporte-Gallet, Fauconnier, and Sighireanu in [12].

ι, ω and Q are the same as in the $flock_i$ protocol, but the transition relation δ differs. The transitions are as follows:

$$\begin{aligned} q, q &\to q, q+1 && \text{for } 1 \leq q < N \\ N, q &\to N, N && \text{for } 0 \leq q < N \end{aligned}$$

Intuitively, when two agents with the same state meet, one of them gets promoted to the next state, while the other stays in the current state. For an agent to reach state N, there must be at least one agent for every state greater than 0 and less than N, i.e. N or more agents that started in state 1 (since agents in state 0 only have meaningful interaction with agents in state N).

Note that again, the number of states is linear in the size of N.

4 Fairness

4.1 Global Fairness and Variations

As mentioned in Section 3.2, in order to ensure meaningful computations take place in a protocol, we need a notion of fairness. Without such a notion it could be the case that only two agents interact, while the rest does not take part in the computation. Recall the example we gave in Section 3.2 for such an execution, where only silent transitions are taken, even though non-silent transitions are enabled. In such a case, it cannot be expected that the protocol gives the correct output. We define *global fairness* (as briefly described in Chapter 3) as follows:

- (I): Let $P = (Q, \Sigma, \iota, \delta, \omega, Y)$ be a population protocol. For all configurations C, $C' \in Pop(Q)$, all executions π: If C occurs infinitely often in π and $C \to C'$, then C' occurs infinitely often in π.

While this definition is sufficient by itself, there are several subtle variations of global fairness. For one, consider the variant where we formulate fairness in the following way:

- (II): Let $P = (Q, \Sigma, \iota, \delta, \omega, Y)$ be a population protocol. For all configurations C, $C' \in Pop(Q)$, all executions π: If C occurs infinitely often in π and $C \xrightarrow{*} C'$, then C' occurs infinitely often in π.

Although the notions look similar, they differ: In (I), we consider only configurations reachable in one step ($\to$) whereas in (II), we consider all configurations reachable in any number of steps ($\xrightarrow{*}$). These differing notions exist since it can be easier to reason using (II), but (I) is the most basic definition. We will make use of this later, e.g. in Section 6.2.

Proposition 1. *For every execution E: E satisfies fairness as defined in (I) if any only if E satisfies fairness as defined in (II).*

Proof. We first show that (II) implies (I), i.e. when the condition in (II) is satisfied, the condition from (I) is also satisfied. Let E be an execution that is fair as given in (II). Then it is trivial that E satisfies fairness as given in (I). This is the case since by definition, if $C \to C'$ holds, then $C \xrightarrow{*} C'$ holds.

Now let E be an execution that satisfies fairness as given in (I). Let C, C' be configurations so that $C \xrightarrow{*} C'$, and the length of the shortest path from C to C' is n. In addition, C appears infinitely often in E. Proof by induction over n: Start with $n = 0$. Then $C = C'$.

Then both notions of fairness are trivially satisfied. Now assume that for some fixed n, it holds for all configurations C and C', if C appears infinitely often in E, and $C \xrightarrow{*} C'$, and the length of the shortest path between C and C' has length n, then C' appears infinitely often in E. Now let C' be reachable from C via a shortest path with $n+1$ steps. Consider now the subpath (of the path from C to C') with length n, so that the subpath starts at C and ends at C_n, and therefore $C_n \rightarrow C'$. Then by the induction hypothesis, as C occurs infinitely often in E, C_n must appear infinitely often in E. Since $C_n \rightarrow C'$, under (I), C_n occurring infinitely often implies C' occurring infinitely often, so C occurring infinitely often implies C' occurring infinitely often. Therefore, C' must occur infinitely often in E. Therefore, the two notions of fairness are equivalent. $\square$

There is a third notion of fairness that again slightly differs, but is equivalent:

- (III): For every configuration C', if π contains infinitely many configurations $C_1, C_2, ...$ in the execution such that $C_1 \xrightarrow{*} C', C_2 \xrightarrow{*} C', ...$, then C' occurs infinitely often in π.

It can be shown that fairness as described in (III) is equivalent to (I) by using the fact that in population protocols, the population never changes sizes, and therefore, there are only finitely many different configurations from which any configuration C' is reachable. From this, it follows that at least one of them must appear infinitely often by the pigeonhole principle.

Another notion of fairness is given as follows:

- (IV) For all transitions t, for all executions $C_0C_1...$: If there is an infinite set of natural numbers N such that in $\forall n \in N$: in C_n, t is enabled, then there exists an infinite set of natural numbers U such that $\forall u \in U : C_u \xrightarrow{t} C_{u+1}$.

Intuitively, if a transition is enabled infinitely often, it should occur infinitely often.

We call this *local fairness*. Local fairness is not equivalent to global fairness as given in (I). The following example illustrates this.

Consider a protocol with the transitions
$a, a \rightarrow b, b$
$b, b \rightarrow a, a$

Then, starting from configuration $\{a, a, a, a\}$, the execution alternating between $\{a, a, b, b\}$ and $\{a, a, a, a\}$ uses both transitions infinitely often, since both are enabled infinitely often. This satisfies local fairness. However, this execution is not globally fair, since the configuration $\{b, b, b, b\}$ is reachable from $\{a, a, a, a\}$, and $\{a, a, a, a\}$ appears infinitely often. When we talk about fairness in the context of population protocols, we will always be talking about global fairness.

In the following, we want to give a criterion that can be used to determine whether a given run is fair. For this, we need to define the reachability graph of a protocol.

For a protocol P, the reachability graph with respect to some configuration C is given by $P_C = (V, E)$ where:

- $V = \{C' | C \xrightarrow{*} C'\}$.
- $E = \{(s, s') | s, s' \in V \wedge s \rightarrow s'\}$.

It is easy to see that each infinite run of the protocol starting at C corresponds to an infinite path over P_C, starting at C, as each node corresponds to one configuration. This leads to a criterion for deciding whether a given run is fair.

Lemma 1. *A run $\mathcal{R}$ starting from some configuration C in a population protocol P is fair iff $\mathcal{R}$ visits all nodes of some BSCC Δ of P_C infinitely often: $inf(\mathcal{R}) = \Delta \in bscc_{P_C}$*

Proof. $\Rightarrow$: Consider a run $\mathcal{R}$ such that $inf(\mathcal{R}) \neq \Delta$ for all BSCCs Δ. Observe first that $inf(\mathcal{R})$ cannot contain nodes from two BSCCs Θ, Ξ with $\Theta \neq \Xi$, since from Θ, by definition there are no outgoing edges to Ξ, and vice versa. Similarly, once an SCC Θ is left by the run, the run cannot reach it again, as there are no cycles between SCCs, which contradicts nodes from multiple SCCs appearing infinitely often. Using that, $inf(\mathcal{R})$ must be a proper subset of some BSCC, or a subset of some SCC that is not a BSCC. We show that there is a node v reachable from some node $u \in inf(\mathcal{R})$ that is not in $inf(\mathcal{R})$. If $inf(\mathcal{R})$ is a proper subset of some SCC Θ, we have that $v \in \Theta \backslash inf(\mathcal{R})$. Such a v exists, as $\Theta \backslash inf(\mathcal{R}) \neq \emptyset$. If $inf(\mathcal{R})$ is a subset of an SCC Θ that is not a BSCC, by definition, from nodes in Θ, a node v outside of Θ must be reachable - otherwise, Θ would be a BSCC. Since $v \notin \Theta$, but $inf(\mathcal{R}) \subseteq \Theta$, it follows that $v \notin inf(\mathcal{R})$. Therefore, we know that $u \in inf(\mathcal{R})$, $v \notin inf(\mathcal{R})$, but $u \rightarrow v$. This contradicts fairness.

$\Leftarrow$: Now consider a run $\mathcal{R}$ such that there exists a BSCC $\Delta : inf(\mathcal{R}) = \Delta$. By definition of BSCCs, from nodes in Δ, no nodes outside of Δ are reachable. Therefore, the only nodes reachable from $inf(R)$ are in Δ. Since $\Delta = inf(\mathcal{R})$, fairness is ensured. $\square$

We can see from Lemma 1 that the BSCCs of the reachability graph of a protocol are closely related to fairness.

4.2 Fairness and Probability Distributions

In many practical applications, we cannot guarantee the fairness condition directly. Instead, there is an underlying probability distribution that determines which transitions occur. We are interested in investigating the connection between probability distributions and fairness, to see what criteria ensure that a distribution is fair. Our goal is to find a class of probability distributions that ensures that executions following such a distribution are fair with probability 1.

Formally, by *probability distribution* we mean a function λ, where $\lambda(t|C)$ is the probability of picking transition t when the system is in configuration C. Since probabilities should sum up to one, we require that $\sum_t \lambda(t|C) = 1$ for all configurations C. In addition, $\lambda(t|C) \in [0, 1]$ for all t, C.

We call a probability distribution λ *admissible* if it is of the following form:

$$\lambda(t|C) = \begin{cases} \alpha_{t,C} & \text{if } t \text{ is enabled in } C \\ 0 & \text{otherwise} \end{cases}$$

In words, an admissible probability distribution ensures that in a configuration C, transitions that are not enabled in C have probability 0 to occur. This is a rather obvious requirement to have, since only enabled transitions can occur. Note that the reverse does not have to hold: There can be enabled transitions that have probability 0 of occurring.

In the following, we are only considering admissible probability distributions.

Note that this requirement is not strong enough to ensure fair runs with probability 1. An example is given as a protocol with the following transitions:

$$a, a \rightarrow b, b$$
$$b, b \rightarrow a, a$$
$$a, a \rightarrow c, c$$

Consider now the probability distribution such that $\lambda((a, a, b, b)|\{a, a\}) = 1$ and $\lambda((b, b, a, a)|\{b, b\}) = 1$. Assume for all other configurations a uniform distribution over all enabled transitions. Then, when $\{a, a\}$ is the initial configuration, with probability 1, the execution starting at $\{a, a\}$ will alternate between $\{a, a\}$ and $\{b, b\}$, with $\{c, c\}$ never occurring in the execution, even though the configuration $\{a, a\}$ occurs infinitely often, and $\{c, c\}$ is reachable from it. This contradicts fairness.

On the other hand, requiring each enabled transition to have a strictly positive probability is too strong, since there are probability distributions that lead to fair executions with probability 1 and that assign some enabled transitions probability 0.

Consider for example a protocol with the transitions

$$a, a \rightarrow b, b$$
$$a, a \rightarrow c, c$$

with a being the only initial state. Let $\lambda((a, a, b, b)|C) = 1$ and $\lambda((a, a, c, c)|C) = 0$ for all configurations C that contain at least two agents with state a. Then starting from any initial configuration, only the transition $a, a \rightarrow b, b$ will be used until there is at most one agent in state a, with the rest being in state b. Then that configuration is also the only configuration appearing infinitely often, and it is also terminal (since only silent transitions are enabled). Therefore, fairness is satisfied, since any configuration reachable infinitely often is reached infinitely often.

4.3 A Necessary and Sufficient Criterion for Fairness

In the following, we will show a criterion for probability distributions that is necessary and sufficient to ensure that executions following that distribution are fair with probability 1.

We extend our definition of the reachability graph of a protocol to take probability distributions into account. Markov chains as described in Section 2.2 give an angle of attack for this.

For a given protocol $P = (Q, \Sigma, \iota, \delta, \omega, Y)$, a probability distribution λ and an initial configuration C, we define the Markov Chain induced by P with respect to λ and C as $M_{P,\lambda,C} = (S, P, \iota_{init})$, where:

- $S = \{C' : C \xrightarrow{*} C'\}$,
- $P(s, s')$ is the probability of picking a transition that changes the system to configuration s' when in configuration s, i.e. $P(s, s') = \sum_{t \in \delta \wedge s \xrightarrow{t} s'} \lambda(t|s)$,
- $\iota_{init}(s) = \begin{cases} 1, & \text{if } s = C \\ 0, & \text{if } s \neq C \end{cases}$

Let $s \in S$ be a state. Then by $Paths_M(s)$, we denote the set of infinite paths in the Markov chain that start at s.

Denote by $bscc_{M_{P,\lambda,C}}$ and $bscc_{P_C}$ the BSCCs of the Markov chain and the reachability graph of the protocol respectively.

It can be shown that the following holds:

Theorem 1. *For every Markov chain $M = (S, P, \iota_{init})$, for every state $s \in S$: $Pr(\pi \in Paths_M(s) | inf(\pi) \in bscc_M) = 1$*

Intuitively, this means that for each state in the Markov chain, paths starting at that state have probability 1 to reach precisely all nodes of some bottom strongly connected component infinitely often. A proof is given in [9].

Let P be a population protocol and let C be a configuration. Further, let λ be a probability distribution and let $M = M_{P,\lambda,C}$ be the Markov chain induced by P with respect to λ and C. Denote by $F_{P,C}$ be the set of fair executions in P that start at C. Let $F_{P,\lambda,C} = Paths_M(C) \cap F_{P,C}$ be the set of fair runs in M. From Lemma 1, we know that

$$F_{P,C} = \{\pi \in Paths(C) | \exists S \in bscc_P : \forall C' \in S : C' \text{occurs infinitely often in } \pi\}$$

Using results given in [9], this is sufficient to show that the set $F_{\lambda,C,P}$ is measurable.

Then we define the following: a probability function λ gives rise to fair executions of P if and only if for all configurations C: $Pr(\pi \in Paths(C) | \pi \in F_{P,\lambda,C}) = 1$.

We can use this fact, together with Theorem 1 and Lemma 1 to prove the following:

Proposition 2. *Given a protocol P and a probability distribution λ, λ gives rise to fair executions of P if and only if:*

$$\forall C \in P_{init} : bscc_{M_{P,\lambda,C}} \subseteq bscc_{P_C}$$

Proof. Let P be a protocol, let C be a configuration and let λ be a probability distribution. We write M for $M_{P,\lambda,C}$ for sake of brevity.

$\Rightarrow$: Assume λ gives rise to fair executions of P. By Theorem 1, it holds that $Pr(\pi \in Paths_M(C) | inf(\pi) \in bscc_M) = 1$. Next, note that for every BSCC $\Delta \in bscc_M$: $Pr(\pi \in Paths_M(C) | inf(\pi) = \Delta) > 0$. This is the case since every BSCC in M must be reachable from the initial state C by definition of M. Additionally, Lemma 1 gives us that a run π is fair iff there is some BSCC $\Delta \in bscc_{P_C}$ such that $inf(\pi) = \Delta$. Assume for contradiction that there is some Θ such that $\Theta \in bscc_M$ but $\Theta \notin bscc_{P_C}$. Then we have that $Pr(\pi \in Paths_M(C) | inf(\pi) = \Theta) > 0$. By Lemma 1, a run π is fair iff $inf(\pi) \in bscc_{P_C}$. As $\Theta \notin bscc_{P_C}$ holds, it follows that $Pr(\pi \in Paths(C) | \pi \notin F_{P,\lambda,C}) \geq Pr(\pi \in Paths_M(C) | inf(\pi) = \Theta) > 0$. This contradicts that λ gives rise to fair executions.

$\Leftarrow$: Assume that it holds that for all configurations $C \in P_{init} : bscc_M \subseteq bscc_{P_C}$. Then from Theorem 1 we have that $Pr(\pi \in Paths(C) | inf(\pi) \in bscc_M) = 1$. From $bscc_M \subseteq bscc_{P_C}$ it follows that $Pr(\pi \in Paths(C) | inf(\pi) \in bscc_{P_C}) = 1$. Using Lemma 1, we can then follow that $1 = Pr(\pi \in Paths(C) | inf(\pi) \in bscc_{P_C}) = Pr(\pi \in Paths(C) | inf(\pi) \in F_{P,\lambda,C})$. □

From Proposition 2, it can easily be seen that a probability distribution that assigns a positive probability to each enabled transition gives rise to fair runs with probability 1, as the BSCCs of the Markov chain are then equal to the BSCCs of the reachability graph of the protocol.

Two distributions have been studied in the literature:[1]

Uniform rules scheduling is the probability function where in configuration C, if there are n transitions enabled, we assign each of those transitions probability $1/n$, i.e. we choose uniformly at random among enabled transitions.

Uniform pairs scheduling is the probability function where we choose uniformly at random among agents. More formally, assume we are in configuration C. Then, a pair (a, b) has probability

$$\frac{|C|_a}{|C|} \cdot \frac{|C \backslash \{a\}|_b}{|C \backslash \{a\}|}$$

Due to nondeterminism, there might be multiple possible transitions for (a, b). Once we have chosen a pair, we then choose uniformly at random over the possible transitions for that pair. More formally, the full probability for a transition $t = a, b \to a', b'$ in a configuration C is given as

$$P(t|C) = \frac{|C|_a}{|C|} \cdot \frac{|C \backslash \{a\}|_b}{|C \backslash \{a\}|} \cdot \frac{1}{|T|}$$

where T is the set of all possible transitions for (a, b), i.e. $T = \{t | pre(t) = (a, b)\}$.

[1]In [12], schedulers implementing these distributions are called *random pairing scheduler* and *random ruling scheduler* respectively.

5 A Tool for Population Protocols

Even though population protocols have been researched for a decade, there are very few tools for simulating and verifying them. Some of the tools we are aware are the following:

- In [11], Chatzigiannakis, Michail and Spirakis give the tool bp-ver, which is capable of verifying protocols for fixed sizes using graph exploration algorithms.
- In [12], Clément, Delporte-Gallet, Fauconnier and Sighireanu use the model checkers SPIN and PRISM to verify protocols for fixed sizes, and compare run times for both model checkers.
- In [20], Sun, Liu, Dong and Chen introduce the model checker PAT, which can be used for verifying protocols for fixed sizes as it supports global fairness.
- In [10], Blondin, Esparza, Jaax and Meyer give the tool peregrine, which is capable of verifying a restricted class of protocols for all infinitely many input sizes.

We have developed a tool that allows specifying protocols and populations as well as their manipulation, verification and simulation. It was written using the Python programming language [19] and the PyCharm development environment [15]. It can be accessed on `https://gitlab.lrz.de/ga96jib/tool_for_population_protocols`.

5.1 Data Structures for Populations and Protocols

There are two underlying components in the tool: protocols and populations.

Populations store the current configuration. They are represented as a dictionary with the states keys, and their respective multiplicity as entries. For example, the configuration $\{|R, R, R, B, B, 0, 1|\}$ would be encoded by the following dictionary:

```
{'1': 1, '0': 1, 'B': 2, 'R': 3}
```

Note that there might be more states in the protocol, for example, there could be a state X that is a state of the protocol, but the configuration does not contain it. It would not appear in the dictionary, as we only have entries for states that are contained at least once in the configuration.

Populations also come in two variations, *mutable* populations, and *frozen* populations. The two versions only differ in a few methods: While items can still be added to and removed from a mutable population, a frozen population cannot be modified once it is created. The frozen version is needed when one wants to use a population in a set or as a key of a dictionary, since Python does not allow that for mutable objects.

Protocols store information about the population protocol that hold independently of the current configuration, such as the transition relation, the output function, and the set of initial states. The states are not represented explicitly, as they can be derived from the transitions. This is also the reason why configurations do not have entries for states that are not present in the configuration; the full state space is not stored explicitly. The transitions are stored in a dictionary in the following way: Let a_0 and b_0 be a pair of agents, and let the transitions where a_0 is the initiator and b_0 is the responder be given as

$$\begin{aligned} a_0, b_0 &\to a_1, b_1 \\ a_0, b_0 &\to a_2, b_2 \\ &\vdots \\ a_0, b_0 &\to a_n, b_n \end{aligned}$$

Assuming there are no other transitions in the protocol, the dictionary for the transition relation is then as follows:

```
{('a0', 'b0'): [('a1', 'b1'), ('a2', 'b2'), ..., ('an', 'bn')]}
```

Observe that non-determinism is in this case handled by storing a list instead of only a single pair. If there is no explicit transition with a_0, b_0 as pre, we assume there is a silent transition. This is achieved by accessing elements of the dictionary using the following method:

```
def __getitem__(self, item):
    return self.transitions.get(item, [item])
```

The *get* method of dictionaries returns the entry for *item* when there is one, otherwise it defaults to the second argument, i.e. *[item]*, a list containing only the *item* itself. Therefore, when there would not be a transition for a given pair, there is a silent transition by default.

In the following, we are going to consider the 4-state majority protocol as an example.

5.2 Generating a Protocol

To generate a protocol, one can supply a collection of transitions. A transition $a_0, b_0 \to a_1, b_1$ is encoded by the tuple (a_0, b_0, a_1, b_1). In addition, initial states and an output function can be given, respectively as a set and a function over the states of the protocol. Recall that transitions need not be deterministic, i.e. it is possible to have two transitions (a_0, b_0, a_1, b_1) and (a_0, b_0, a'_1, b'_1).

Recall that the transitions for 4-state majority were defined as follows:

$$R, B \to 1, 0$$
$$R, 0 \to R, 1$$
$$B, 1 \to B, 0$$
$$1, 0 \to 1, 1$$

As an example, the function used to generate this protocol in the tool is as follows:

```
def get_4state_majority_protocol():
    transitions = [("R", "B", "0", "1"), ("R", "1", "R", "0"),
                   ("B", "0", "B", "1"), ("0", "1", "0", "0")]
    initial_states = ["a", "b"]
    output_function = lambda x: True if x == "a" or x == "0" else False
    return Protocol(transitions, initial_states, output_function)
```

For example, the transition $R, B \to 0, 1$ corresponds to the tuple $(''R'', ''B'', ''0'', ''1'')$. As mentioned before, the initial states are simply given as a set, which in this case contains only R and B.

The output function simply gives True for R and B, and False otherwise, which corresponds to the output function of the 4-state majority, where we had $\omega(R) = 1$, $\omega(0) = 1$, $\omega(B) = 0$ and $\omega(1) = 0$.

5.3 Generating a Population

A population can be generated by supplying a collection containing the elements in the configuration, e.g. a list.

The following code-snippet shows an example of code used to generate a population for the 4-state majority:

```
population = p.FrozenPopulation([R, R, R, B, B, B])
```

This corresponds to the initial configuration $\{|R, R, R, B, B, B|\}$. This instantiates a frozen version of the population, which cannot be modified after creation. If we wanted to create a mutable population with the same elements, we simply write Population instead of FrozenPopulation:

```
population = p.Population([R, R, R, B, B, B])
```

5.4 Simulating a Protocol

There are different functions to simulate runs on a protocol.

```
def average_convergence_steps(self, initial_population, num_iterations=5,
    ↪ stable_configurations=None, probability_function=None):
        ...
        for i in range(num_iterations):
            count = 0
            current_population = initial_population

            # simulate protocol until a stable configuration is reached
            while not (current_population in stable_configurations):
                # pick transition according to probability function, then
                    ↪ update current population using that transition
                current_population = self.successor_transition(
                    ↪ probability_function(current_population),
                    ↪ current_population)
                count += 1
```

Listing 5.1: Average convergence steps method of the *Protocol* class, shortened for presentation here.

Average number of steps to convergence (Listing 5.1). This function measures how many steps are on average needed for runs in the protocol to converge. We say that a run has converged once it reaches a stable configuration. The number of steps needed to convergence is then the number of steps until a stable configuration is reached.

In the following, we will give a brief explanation of the code of the function.

Observe first the parameters.

- *self* is needed since it is a method of a class (for specifics on this, see [19]),
- *initial_population* is the population from which executions should start,
- *num_iterations* is how many runs should be simulated,
- *stable_configurations* are the stable configurations reachable from the specified initial population, and
- *probability_function* is the probability function to be used.

Each run starts at the specified initial population, and then goes on until it has reached a stable configuration for the first time, at which point the run is completed and the steps the run needed to converge are stored. At the end, the average over the runs as well as the standard deviation are computed using the stored values. In each step the probability function chooses a transition to be taken based on the current population, and the *successor_transition* method then returns the resulting population.

For details on how the stable configurations can be computed, see Section 5.7.

Failure ratio. It is straightforward to adapt the method for computing the average steps until convergence for the computation of the failure ratio. Instead of measuring the number of steps for each run, we count how many runs reach a stable configuration with the correct output value, and how many runs reach an incorrect one. For this, it is additionally needed that the user supplies which output should be seen as correct for the protocol with regards to the given initial configuration. Again, a method for this can be found in the *protocol* class of the tool.

5.5 Verifying a Protocol

Additionally to simulation, the tool can be used for verification. The tool offers ways to check a protocol for well-specification for all inputs of a given size. Well-specification is a property of the BSCCs of the reachability graph, since fair runs will eventually end up in some BSCC and visit all nodes there infinitely often, as we have shown in Section 4.1. Therefore, we can simply check if all BSCCs of the reachability graphs for all inputs of the given size only contain populations with the same consensus value.

The tool also allows to check whether the protocol computes a given function for a given size of inputs. By abuse of notation, we say a protocol is-well-specified for a size s if for all initial configurations C with $|C| = s$, there exists $y \in Y$ such that for all executions π, if $\pi(0) = C$, then $O(\pi) = y$. Informally, this means we relax well-specification to well-specification for initial populations of some given size.

For this, the correct output for a given input population will be computed using the function and compared to the consensus values of the BSCCs of the reachability graph starting at the population. The protocol then computes the function for the given size if for all input populations of that size, all BSCCs contain only configurations in a consensus with the correct value, i.e. the result of applying the function to the input population.

Lastly, the tool also has a function to check whether a given protocol is silent up to a given size. For this, the BSCCs are computed for all initial populations up to the given size. If all BSCCs reachable from an initial population contain a single node, then the protocol is silent for that population.

We will present the procedure used to compute BSCCs of reachability graphs in Section 5.7.

5.6 Using a Model Checker to Analyse Convergence Properties

Simulating a protocol to find the expected steps until convergence or the failure ratio is only an approximation of the real value that gets better with larger sample size. For exact results, we used the PRISM probabilistic model checker [18]. PRISM allows deterministic Markov chains as input models, so we used the fact that a protocol, a scheduler and an initial configuration induce a Markov chain (see Section 4.3) to obtain a representation that we could use for PRISM.

We introduce a method to export a protocol to the *PRISM language*, and use the *PRISM property specification language* to give properties that represent well-specification. PRISM also allows to check quantitative properties, which we use to compute the expected number of steps to convergence.

Roughly speaking, PRISM models consist of a finite number of variables, each with their respective bounds, and commands. Commands consist of a guard that determines when the command should be usable, and one or more updates, each with a probability, that describe changes to variables. Consider the following example:

```
[] x=0 -> 0.8:(x'=0) + 0.2:(x'=1);
```

This command means that the command describes the behaviour of the system when $x = 0$. If that is the case, there is a probability of 0.8 for x to remain at 0 and a probability of 0.2 for x to be changed to 1. The "[]" at the beginning can be ignored for our purposes. See [18] for a more detailed explanation of the PRISM language.

In addition, so-called rewards can be assigned to states, depending on conditions. For example, one could assign a reward of 1 for each state that has $x = 0$, and reward 0 to the rest. For us, rewards are going to be steps, so that we can look at the cumulative rewards to obtain the number of steps needed, i.e. the expected number of steps until convergence.

Our export function to PRISM comes in two variations, one for arbitrary distributions, and one for uniform distributions.

Both approaches have in common that each state of the protocol is encoded as a variable that can assume values between 0 and the size of the initial configuration. These variables encode the current multiplicity of agents with the respective state in the current configuration. Since population sizes do not change, this suffices to be able to represent all configurations reachable from the initial configuration. Initially, all variables therefore have a value that equals their multiplicity in the initial configuration.

The variables and their initial values for the 4-state majority with an initial population of $\{|R, R, B, B|\}$ are depicted in Listing 5.2.

```
R : [0..4] init 2;
B : [0..4] init 2;
"0" : [0..4] init 0;
"1" : [0..4] init 0;
```

Listing 5.2: Snippet of the exported Markov chain for the 4-state majority protocol, starting from $\{|R, R, B, B|\}$.

There is also a slight variation to this variable initialization. Imagine one wants to export Markov chains for all initial populations of a given size. Instead of generating a model for each, we use another capability of PRISM to achieve this in an easier manner, as one can specify ranges for initial values of variables. We simply specify that the sum of the multiplicities of all initial states must equal the desired population size, while the multiplicities of non-initial states are initially 0. An example of this is given in Listing 5.3 for the 4-state majority protocol and all initial populations of size 4.

```
R : [0..4];
B : [0..4];
"0" : [0..4];
"1" : [0..4];
...
...
...
init R+B = 4 & "0" = 0 & "1" = 0 endinit
```

Listing 5.3: Snippet of the exported Markov chain for the 4-state majority protocol for all initial populations of size 4.

5.6.1 Arbitrary Probability Distributions

In the approach for arbitrary distributions, the PRISM model contains one command for each configuration C' that is reachable from C. The guard of the command ensures that the command is only usable when the system is in the respective configuration, i.e. each variable has a value equal to its multiplicity in C'. The updates are then the transitions leaving C' in the Markov chain $M_{P,\lambda,C}$, with their respective probabilities. These updates change the values of the variables according to the transition. Note that in P, there might be multiple transitions that lead to the same resulting configuration, in which case the probabilities of those transitions need to be summed into a total probability, as defined in Section 4.3.

Each state is assigned a reward of 1, since each command represents *one* step of the protocol.

As an example, we are going to examine the commands and states of the exported Markov chain for the 4-state majority protocol, starting from the initial configuration $C_0 = \{|R, R, B, B|\}$, as seen in Listing 5.4.

Note that this is only a snippet, and the whole model has 8 commands, one for each configuration reachable from C_0. This approach needs exactly one command per reachable configuration, and each command will have as many updates as there are transitions with positive probability.

5.6.2 Uniform Rules Scheduling

Our approach for using arbitrary probability distributions requires the whole reachability graph to be built, and generates one command per configuration. This means that the exported model can get large even for relatively small populations. For this reason, we implemented a special case for uniform rules scheduling. We tested and implemented two different approaches for this.

```
[] R = 2 & B = 2 & "0" = 0 & "1" = 0 ->
    1.0: (R'=R-1) & (B'=B-1) & ("0"'="0"+1) & ("1"'="1"+1);
[] R = 1 & B = 1 & "0" = 1 & "1" = 1 ->
    0.25: (R'=R-1) & (B'=B-1) & ("0"'="0"+1) & ("1"'="1"+1) +
    0.5: ("1"'="1"-1) & ("0"'="0"+1) +
    0.25: ("0"'="0"-1) & ("1"'="1"+1)
[] R = 0 & B = 0 & "0" = 2 & "1" = 2 ->
    1.0: ("0"'="0"+1) & ("1"'="1"-1);
```

Listing 5.4: Snippet of the variables of the exported Markov chain for the 4-state majority protocol, with the initial configuration of $\{|R, R, B, B|\}$.

Nondeterministic encoding. This approach makes use of the fact that PRISM, when confronted with overlapping guards (i.e. multiple commands are enabled in a state), resolves this nondeterminism by choosing uniformly at random among the enabled commands. Therefore, we specify for each transition one command. The guard of that command evaluates to true if and only if the transition is enabled, and there is a single update with probability 1 that describes how the transition changes the configuration.

In Listing 5.5, we illustrate the exported model of the 4-state majority protocol with the initial configuration $\{|R, R, B, B|\}$. Note that the four commands in the snippet are already all the commands that are needed.

Note that the guards contain two parts: one ensures that the transition is enabled, e.g. in the first command, that there is at least one agent in state R and one agent in state B. The other part ensures that the variables are not incremented above their bounds, i.e. the size of the initial configuration. This requirement is needed even though no configuration can violate it, since otherwise, PRISM will report the guards as too weak. This is due to PRISM checking the guards for states that are not reachable from the configuration, but within the bound of the variables.

Again, each state corresponds to one step, so we set the reward of each state to 1.

```
[] R >= 1 & B >= 1 & "0" <= 3 & "1" <= 3 ->
    (R'=R-1) & (B'=B-1) & ("0"'="0"+1) & ("1"'="1"+1);
[] R >= 1 & "1" >= 1 & "0" <= 3 ->
    ("1"'="1"-1) & ("0"'="0"+1);
[] B >= 1 & "0" >= 1 & "1" <= 3 ->
    ("0"'="0"-1) & ("1"'="1"+1);
[] "0" >= 1 & "1" >= 1 & "0" <= 3 ->
    ("0"'="0"+1) & ("1"'="1"-1);
```

Listing 5.5: Snippet of the exported Markov chain for the 4-state majority protocol, starting from $\{|R, R, B, B|\}$ and using uniform rules scheduling with the approach for arbitrary distributions.

One downside to this approach is that it exploits an implicit behaviour of PRISM that may be changed in future versions. An upside is that the size of the model is reduced drastically, requiring one command per transition instead of one per reachable configuration, and it does not require computing the reachability graph.

Deterministic encoding. We present an alternative approach that allows us to export a protocol, using uniform rules scheduling as probability distribution, without generating the reachability graph. For this approach, we make use of two additional counters, n and i. For this, we assume that we have numbered the transitions in some way, so that they are given as $t_0, t_1, t_2, ...$. The approach works by considering the transitions in succession. The counter n stores which transition is currently in consideration. Initially, $n = 0$, since we are considering the first transition. We then check whether transition t_n is enabled. If it is not, we increase n by 1 and move on to consider the next transition. If t_n is enabled, we choose transition t_n with probability $1/T$, where T is the number of currently enabled transitions. On the other hand, with probability $1 - (t/T) = (T-1)/T$, we do not take the transition (even though it is enabled) and go on to consider the next transition. Since we check transitions in succession we need to take into account conditional probabilities, i.e. if we were to simply always assign probability $1/T$ to choosing a transition that is enabled, we would have that the first enabled transition has probability $1/T$ to be taken, whereas we have a probability of $1/T$ to move to the next transition. Due to conditional probabilities, we would then have a conditional probability of $((T-1)/T) \cdot (1/T)$ to take the second enabled transition, i.e. the probability of not taking the first enabled transition times the probability of taking the second enabled transition. This would result in the cumulative probabilities not equalling a uniform distribution. To adjust for this, we used another counter i. Since using only n, we have no information about how many enabled transitions were considered (but not taken), we use i to store exactly that information.

At first, i is 0. When we check an enabled transition, the probability to take that transition is then $1/T - i$. When we do not take an enabled transition, we increment i. In general, the probability for the k-th enabled transition to be taken is:

$$\left(\prod_{i=0}^{k-2} \frac{(T-i)-1}{T-i} \right) \cdot \frac{1}{T-(k-1)} = \frac{1}{T}$$

This describes the cumulative probability not to take any of the first $k-1$ transitions times the probability to then take the k-th transition. By this we have that our distribution does in fact model a uniform distribution over enabled transitions, since the probability for all enabled transitions is $1/T$. When a transition is finally taken, we apply the transition to update the current configuration. After that, we start again at $n = 0, i = 0$. For a visualization of the concept, see Figure 5.1.

In this model, not every state corresponds to a step in the protocol. To address this issue, we assign a reward of 1 to all states with $n = 0$, and a reward of 0 to all other

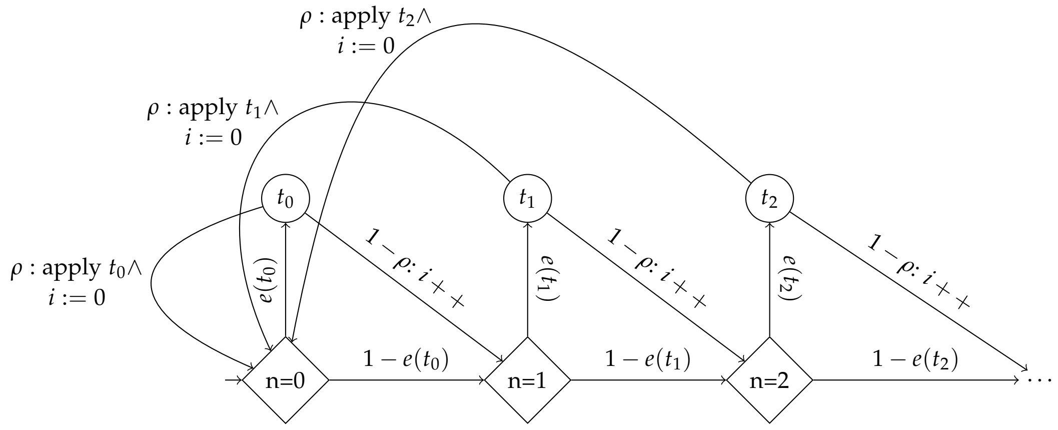

Figure 5.1: Illustration of the approach for deterministic encoding of uniform rules scheduling.
$e(t_x) = 1$ if t_x is enabled, else 0;
$\rho = (i+1)/T$, where $T = \sum_x e(t_x)$, i.e. T is the number of currently enabled transitions.

states. This is because we set n to 0 when we apply a transition, so it is sufficient to count the number of times where $n = 0$ to get an accurate number for the steps taken.

Problems with different encodings. A problem we encountered was that of numerical errors. We compared our approach for arbitrary distribution, choosing uniform rules scheduling as the distribution, with the nondeterministic and deterministic encodings for uniform rules scheduling. While arbitrary distribution and nondeterministic encoding for uniform rules scheduling gave the exact same results when calculating the average number of steps until convergence, the deterministic encoding for uniform rules scheduling gave slightly differing numbers. The difference was rather small, on the scale of 10^{-7} for populations of size 20. After some investigation, we suspect the numerical instability of the calculations PRISM uses to be the cause of this. While the difference was negligible in our case, this is still an area of concern. However, setting the stopping threshold for the numerical approximation in PRISM lower, which leads to more accurate computations, seems to solve this problem.

5.6.3 Specifying Properties in PRISM

We are mainly interested in two general classes of properties that we can specify in PRISM: *probability-based properties* and *reward-based properties*. PRISM's *property specification language* (see [18]) allows us to specify both.

Probability-based properties are properties that are expressed as a probability, such as the probability to reach a configuration in a consensus with a certain value. In the scope of this thesis, the probability-based properties we are interested in are well-specification and failure probabilities. well-specification can be specified as the following formula:

```
P=? [ (F G "output_true")|(F G "output_false") ]
```

where all configurations that are in a consensus with value 1 (resp. 0) satisfy "output_true" (resp. "output_false"). Informally, the formula gives the probability of: In a run on the protocol, is there a point from which on all configurations satisfy "output_true" (resp. "output_false"). This property therefore gives the probability of reaching a lasting consensus with value 1 or value 0. For well-specified protocols, we expect this probability to be 1 (assuming 0 and 1 are the only outputs).

Reward-based properties are properties that are expressed as a cumulative reward, e.g. the expected number of steps to reach a consensus. Reward-based properties differ from probability-based properties: PRISM does not allow the G operator in the former, nor does it allow a negated F operator, while it is usable in the latter. It is difficult to specify a lasting consensus in LTL without these operators, as there is no way to ensure a consensus that has been found is actually lasting.

We overcame that problem in our implementation by precomputing all stable configurations, and creating new labels that are satisfied when the current configuration is stable. This is identical to how we identified stable configurations when we measure average steps to convergence, presented in Section 5.4. Even though this approach makes computing reward-based properties possible, it requires computing the reachability graph and identifying the strongly connected components.

An example for a reward-based property is given as follows:

```
R=? [ (F "stable_true")|(F "stable_false") ]
```

where stable configurations in consensus with value 1 satisfy "stable_true", and stable configurations in consensus with value 0 satisfy "stable_false". This gives the expected number of steps until a stable configuration is reached.

5.7 Finding Strongly Connected Components of the Reachability Graph

When we want to know whether a protocol is silent or well-specified for a certain size, we need to identify the bottom strongly connected components of the reachability graph of the protocol. For this, we used Tarjan's algorithm for strongly connected components [21] to identify them, but added a slight modification that allows to obtain the bottom strongly connected components without having to check every strongly connected component for outgoing edges. The algorithm is given in Listing 5.6, with our modification in orange. The algorithm follows Tarjan's algorithm, but has an additional variable for each node that indicates whether or not the node is marked.

In the following, we give an explanation of why the modification identifies the bottom strongly connected components.

When Line 21 is called, `v` has a successor `w` that is indexed, but not on the stack. Let Δ be the SCC of `w`. Then Δ must already be completed. This follows from the fact that nodes get indexed and put on the stack when they are first visited, and the only way to remove a node in Δ from the stack is when Δ is completed, and all nodes of Δ are removed from the stack. Therefore, from the SCC that contains `v` a node in Δ, reachable. Therefore, the SCC of `v` cannot be a BSCC.

Lines 33-36 are called when an SCC is completed and mark the node on top of the stack. Since a node on the stack is reachable from the nodes under it in the stack, that also means that from the SCC of `w`, the SCC of `v` is reachable, i.e. the former SCC cannot be a BSCC.

On the other hand, the algorithm in Listing 5.6 examines every edge at least once. Let Δ be an SCC, from which another SCC Θ is reachable. Then we know that Θ will be completed before Δ, since Tarjan's algorithm completes the SCCs in topological order. This means that either from Δ, the algorithm explores Θ and puts it on the stack, which leads to Lines 33-36 marking some node in Δ, or Θ is explored and completed from some SCC other than SCC Δ, in which case, Line 21 will mark a node in Δ when the edge from Δ to Θ is explored.

In addition to the code shown here, in the actual implementation, the algorithm calls a "stopping criterion" each time an SCC is found, which allows us to potentially stop the algorithm early. For example, when checking whether a protocol is silent, we can stop after finding a single BSCC containing more than one configuration, instead of waiting for the algorithm to return the set of SCCs.

```
input: a graph (V, E)
output: a set of strongly connected components

Stack S = ()

def strongconnect(v):
    v.index = index
    v.lowlink = index
    v.marked = false
    index += 1

    S.push(v)

    for (v, w) in E do:
        if w.index is undefined:
            strongconnect(w)
            v.lowlink = min(v.lowlink, w.lowlink)
        else if S.contains(w):
            v.lowlink = min(v.lowlink, w.index)
        else:
            v.marked = true

    if v.lowlink = v.index:
        start new SCC c
        c.bottom = true
        do:
            w = S.pop()
            c.add(w)
            if w.marked:
                c.bottom = false
        while(w != v)
        add c to the set of SCCs
        if S is not empty:
            w = S.pop()
            w.marked = true
            S.push(w)

index = 0
for all v in V:
    if v.index is undefined:
        strongconnect(v)
```

Listing 5.6: Tarjan's algorithm for strongly connected components. Modifications are given in orange.

6 Case Study

We conducted two case studies for our tool.

In the first case study, we used the tool to simulate and verify several protocols given in Section 3.3. The research question we investigated was:

How do different probability distributions influence time to convergence for different protocols?

In the scope of this question, we implemented the 4-state, 3-state and AVC majority protocols, as well as a modified 4-state protocol, where we remove the "tiebreaker" transition $0, 1 \to 0, 0$. We used these implementations to measure the average number of steps until convergence for these protocols, using different probability distributions. The results show that different probability distributions vastly influence convergence times for the same inputs. It can then be helpful to investigate how distributions may be adapted to give faster convergence, or provide a better failure ratio.

Secondly, we propose a new protocol for the flock of birds predicate (see Section 3.3.4), with the objective to use less states than existing protocols. We use the tool to check well-specification of the new protocol, and additionally give a formal proof. Lastly, will compare convergence times of the new protocol and existing protocols for the flock-of-birds predicate.

All tests were run on the same machine with an Intel Core i5-4210U CPU, as well as 8 GB of RAM, with a timeout set to 1 hour.

6.1 Simulating established Protocols

6.1.1 Majority Problem

We defined the majority problem in Section 3.3.3, and gave several protocols for it: 4-state, 3-state and AVC majority. In first tests, we could see that convergence for the 4-state protocol can be very slow for cases where $R < B$, but where the majority is very small, e.g. there are 10 agents in state R and 11 agents in state B. This is the case since for the protocol to converge, all agents in state R must initiate an interaction with an agent in state B, so that there are no more agents in state R. Therefore, there will at some point only be one agent in state B left, while the rest of the agents are in states 0 and 1. Due to the "tiebreaker" transition $0, 1 \to 0, 0$, the protocol has a bias towards converting agents to state 0 instead of state 1, and that bias gets stronger when there are more agents in states 0 and 1, and less agents in state B. This is due to the fact that progress converting agents to state 1 achieved by the transition $B, 0 \to B, 1$ is likely to be reverted by the transition $0, 1 \to 0, 0$.

To get experimental data on the impact of this tiebreaker, we add a second version of the 4-state majority, that does not have the tiebreaker transition. Then, the transitions are given as follows:

$$R, B \to 0, 1$$
$$R, 1 \to R, 0$$
$$B, 0 \to B, 1$$

All other properties remain unchanged. Note that this protocol is still guaranteed to converge for all initial configurations C where $|C|_R \neq |C|_B$. For the forthcoming graphs, we used the tool to export the protocols to PRISM, and then used PRISM to compute the expected number of steps until a stable configuration is reached.

Parameters. For the AVC protocol, we used $m = 3, d = 1$. The value for d was chosen as 1 since first exploratory tests led to the conclusion that increasing d does not lead to better convergence behaviour[1], but lead to larger numbers of states, and therefore time and space requirements. Higher values for m on the other hand did give lower convergence times, but due to memory constraints, higher values led to PRISM running out of memory during model checking. We decided on $m = 3$, as this is enough to lead to improvements in convergence behaviour over both variations of the 4-state majority protocol. For our measurements, we computed the convergence times for all initial configurations of size 21.

Procedure. This was done by generating the protocol, then exporting it to PRISM for all initial configurations, and lastly invoking PRISM to verify the following property:

```
R=? [ (F "stable_true")|(F "stable_false") ]
```

This property gives the average steps until a stable configuration of output True or False is reached. We used a time limit of 1 hour for separate initial population. We ran the measurements twice, once for uniform rules scheduling, and once for uniform pairs scheduling.

Results. The results of the measurements for uniform pairs scheduling can be seen in Figure 6.1, while the results for uniform rules scheduling are shown in Figure 6.2. The graphs are to be read as follows: Each group of bars is for one initial population of size 21. The bottom axis describes how many agents started with input R, with the rest starting with input B. The height of the bars represents the expected number of steps to reach a stable configuration.

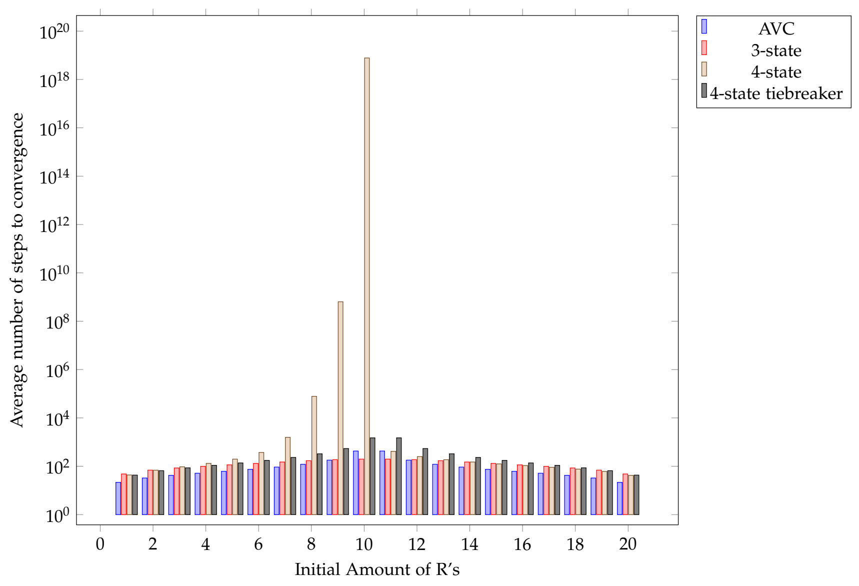

Figure 6.1: Comparison of average number of steps to convergence of protocols for the majority problem, using uniform pairs scheduling.

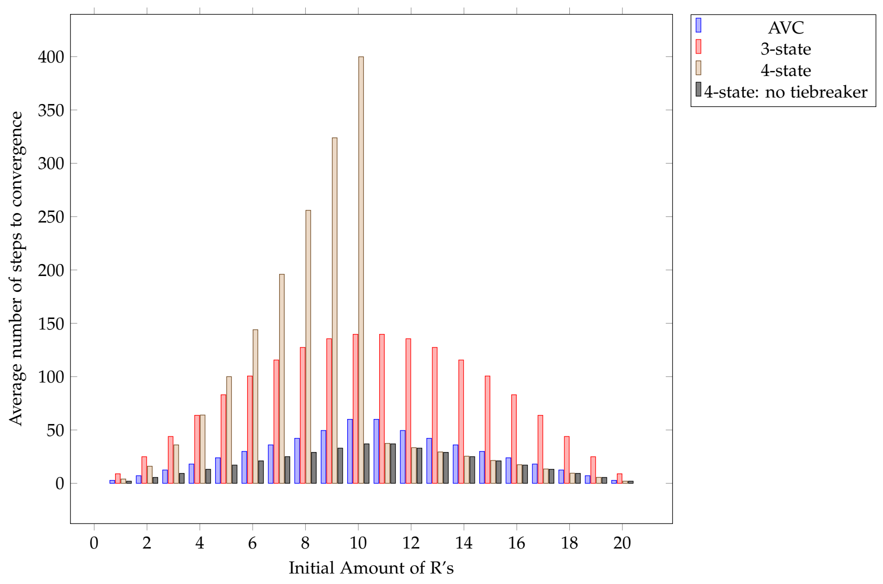

Figure 6.2: Comparison of average number of steps to convergence of protocols for the majority problem, using uniform rules scheduling.

Observe that regardless of the probability distribution, as per our expectations, the 4-state majority protocol with tiebreaker takes significantly more time to reach a stable configuration when there is a slight majority in favour of B in the initial configuration. In the case of uniform pairs scheduling, the time to convergence in the worst case (i.e. 10 R's and 11 B's) is large enough that for all practical concerns, it does not converge. On the other hand, uniform rules scheduling gives better convergence times for nearly all protocols and cases, and leads to even the 4-state majority with tiebreaker being able to converge in a number of steps only around 4 times as large as the number for the other protocols. It seems to be that this is the case due to uniform pairs scheduling leading to many silent transitions, while silent transitions will never occur in uniform rules scheduling. Note also that for uniform pairs scheduling, the AVC protocol lead to the best results, while for uniform rules scheduling, the 4-state majority protocol without tiebreaker has the best convergence time. In addition, it can be observed that with uniform pairs scheduling, the 4-state majority with tiebreaker converges faster than the 4-state majority protocol without a tiebreaker in the cases where there is a majority in favour of R.

Conclusions. Experiments indicate that probability distributions, especially those that assign probability 0 to silent transitions, can have significant impact on convergence time. This is the case even when considering only the rather simple distributions of uniform pairs scheduling and uniform rules scheduling. The results raise the question of finding an efficient way to handle ties in an exact protocol.[2]

6.1.2 3-state Majority

In the following, we want to focus on the 3-state majority protocol. It is unique among the protocols that we considered, since it is only approximately correct. As there is only a small number of states, and therefore, reachable configurations, we could handle larger population sizes than for the other protocols. Instead of using PRISM, in this study, we used the tool to simulate the protocol with different probability distributions.

Parameters. For the measurement, we considered initial configurations of size 50. We set the number of simulations per initial population to 10^4, and computed the average out of these runs.

Procedure. The measurement was done by generating the protocol within the tool, and using the methods for computing the average number of steps to convergence as well as failure ratio. These methods simulate the protocol until a stable configuration is

[1]This was also observed in [1].

[2]For parallel steps, i.e. where as many agents as possible interact in each step, Alistarh et al. [1] gave a lower bound for worst case convergence steps of an exact protocol in $\Omega(\log n)$, with n being the number of agents.

reached, and are described in more detail in Section 5.4. We did this once for uniform pairs scheduling, and once for uniform rules scheduling.

Results. The results for the average number of steps until a stable configuration is reached can be seen in Figure 6.3, while the failure ratios can be seen in Figure 6.4. Note that the colors of the bars indicate the probability function that they belong to, as per the legends of the figures. Again, the x-axis describes the population distribution, where an x-value of n describes that n agents have input R, while $50 - n$ agents have input B. The height of the bars represents the average number of steps and the failure ratio respectively.

Observe that as there is no bias in the 3-state protocol, the graphs are symmetrical around $x = 25$. As is expected without bias, the failure ratio is 0.5 in that case, as it is equally likely to reach a consensus of either 0 or 1. Considering failure ratio, it is clear that uniform pairs scheduling is strictly superior to uniform rules scheduling, as the failure ratio of the latter is higher for all initial configurations. To understand why, recall the states of the 3-state majority problem, i.e. x, y and b. Consider now the case where there is one agent in the blank state b, one agent in state y, and the remaining 48 agents in state x. Recall that when an agent in a non-blank state (either x or y) initiates an interaction with an agent in state b, it converts the agent in the blank state to its respective state. Using uniform rules scheduling, the transitions $x, b \to x, x$ and $y, b \to y, y$ are equally likely to occur, whereas with uniform pairs scheduling, it is vastly more likely that $x, b \to x, x$ occurs. This means that using uniform rules scheduling, even with fairly significant majorities in favour of either R or B, there is still a non-negligible chance to converge to the wrong result. On the other hand, for uniform pairs scheduling, the probability approaches 0 faster as the majority gets more significant. For the average number of steps to convergence, we observe that the convergence behaviour for uniform rules scheduling is worse than for uniform pairs scheduling when the amounts of R's and B's are close to each other. When the majority is very distinct in favour of either R or B, uniform pairs scheduling takes more steps than uniform rules scheduling. The latter fact can be explained by considering the transitions that need to happen when there are very few states of the colour with less states, i.e. the minority. With uniform pairs scheduling, it is very likely that instead of a transition that makes progress taking place, a silent transition between two states of the majority colour happens. For uniform rules scheduling, these transitions have probability 0 to occur. On the other hand, when there is only a slight majority, uniform rules scheduling is likely to schedule transitions that reverse progress that was already made towards a stable configuration. Uniform pairs scheduling is less likely to schedule these transitions as more and more agents are converted to the majority.

Conclusions. While for 4-state majority with or without a tiebreaker, it seemed that uniform rules scheduling was mostly superior, the reverse is true for the 3-state majority protocol. It might warrant further investigation into which criteria make a protocol have

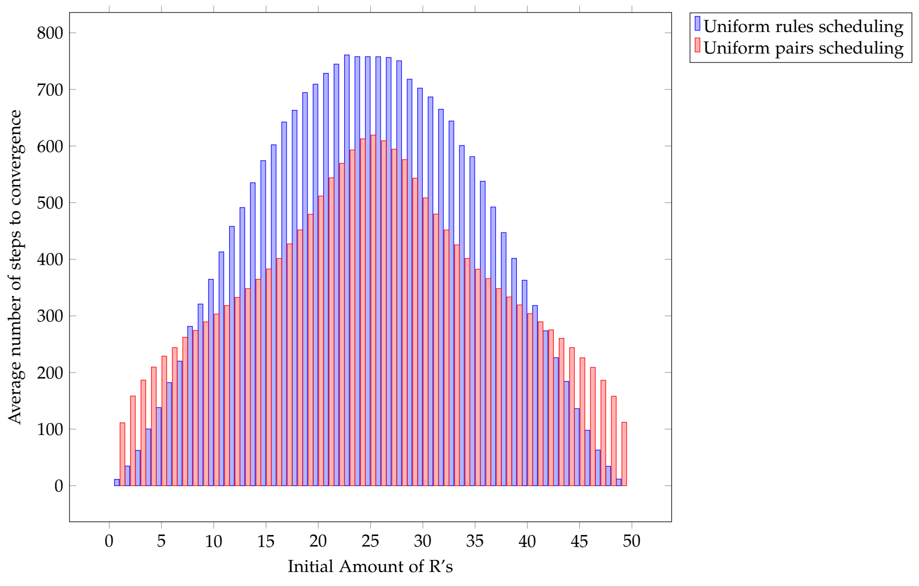

Figure 6.3: Average number of steps until convergence for the 3-state majority protocol with different probability distributions.

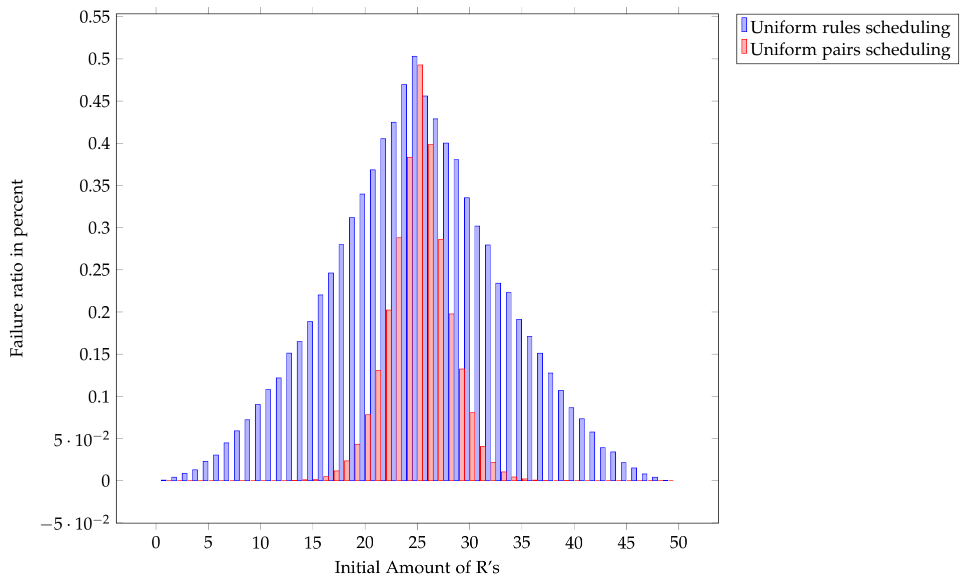

Figure 6.4: Failure ratio for the 3-state majority protocol with different probability distributions.

more favourable convergence properties with certain probability distributions. Possible criteria that lead to better convergence times with uniform pairs scheduling could be the ratio of silent transitions to non-silent transitions, or the presence of rare states that need to interact multiple times for convergence. By rare states, we mean states that are only present in the population with a very small multiplicity, e.g. when there is a single agent in state B in the 4-state majority. If such agents need to take part in a sequence of interactions, this slows down convergence greatly. The worst case seems to be when a rare state needs to take part in a number of transitions, but at the same time, progress the rare state made through interactions can be reversed by some other transitions.

6.2 A new Protocol for Flock of Birds

In the following, we will show how to use the tool to check and simulate a newly devised protocol.

In Section 3.3.4, we introduced the flock of birds predicate, and gave two existing protocols that compute it. Recall that for a given constant $N \geq 0$, the predicate evaluates to true when more than N input values are 1. One downside of the existing protocols is that they need at least N states. We devised a new protocol for the flock of birds problem, which we will give in the following. We will then use the tool and PRISM to check well-specification of our protocol up to a given size, followed by a formal proof of correctness for the protocol. Lastly, we are going to compare the protocol to the existing protocols, with respect to expected time to convergence.

6.2.1 The *Prime-Flock-Protocol*

We describe a new protocol that can compute the flock of birds predicate.

Consider the prime factorization of $N = p_1 \cdots p_n$ s.t. prime factors are in non-decreasing order, i.e. $p_1 \leq \cdots \leq p_n$. The states of the protocol are given as $Q = \{i_k \mid 1 \leq k \leq n, 1 \leq i < p_k\} \cup \{1_{n+1}, 0\}$. The input function is given as $\iota(1) = 1_1, \iota(0) = 0$, and the output function as $\omega(s) = 1$ if $s = 1_{n+1}$ else $\omega(s) = 0$.

For transitions, consider the following relation:

$$\begin{aligned} i_k, j_k &\to (i+j)_k, 0 && \text{if } i+j < p_k, \\ i_k, j_k &\to 1_{k+1}, (i+j-p_k)_k && \text{otherwise,} \end{aligned}$$

where $k \leq n$, $i, j < p_k$.

There are also transitions $1_{n+1}, x \to 1_{n+1}, 1_{n+1}$ for all $x \in Q$, i.e. when 1_{n+1} initiates a transition, no matter the responder, both agents are put into state 1_{n+1}

Intuitively described, agents have two counters, one that determines which level the agent is on (with each level being associated with one prime factor, expect for level $n+1$, which is associated with the full number N), and the other counter determining the progress towards the next level. When two agents on the same level meet, their progress counters are summed into one of the agents. If that is enough to get a progress counter of at least the prime factor associated with the current level, the agent is put into the next level, while the other agent keeps the surplus progress counter, which is the sum of the counters minus the prime factor of the current level. Otherwise, one agent stays on the current level (and now has the added progress counters of both agents), while the other agent is put into state 0.

One can also think of states as representing numbers. For that, let the value of an agent in state i_j be given as $val(i_j) = p_1 \cdot p_2 \cdots p_{j-1} \cdot i$ and let $val(0) = 0$.

Let the level of an agent in state i_j be $lvl(i_j) = j$ and let $lvl(0) = 0$.

As an example, consider the protocol for $N = 45$. The prime factorization in non-decreasing order is given as $45 = 3 \cdot 3 \cdot 5$. The set of states is therefore

$$\{1_1, 2_1, 1_2, 2_2, 1_3, 2_3, 3_3, 4_3, 1_4, 0\}$$

We have, for example, $val(1_2) = p_1 \cdot 1 = 3$, $val(2_3) = p_1 \cdot p_2 \cdot 2 = 3 \cdot 3 \cdot 2 = 18$ and $val(1_4) = p_1 \cdot p_2 \cdot p_3 \cdot 1 = 3 \cdot 3 \cdot 5 \cdot 1 = 45 = N$.

Finally, note that the protocol has a number of states in $O(\sum_{i=1}^{n} p_i)$. The best case is when N is a power of 2, whereas the worst case occurs when N is a prime number, as $N + 1$ states are needed.

6.2.2 Checking Well-Specification

As mentioned in Section 5.5, it is possible to use the tool to check whether a given protocol computes a function, for a given size. We implemented the prime-flock-protocol, and then checked whether the protocol computes the following predicate on the initial configuration C:

$$|C|_{1_1} \geq N$$

i.e. are there at least N agents starting in state 1_1. Observe that due to agents with input 1 starting in state 1_1, this is equivalent to asking whether the sum of inputs is at least N. This means the protocol is correct if it computes the predicate given above for all sizes.

For the prime-flock protocol, we checked correctness for initial configurations of sizes 5 to 15, with N ranging between 0 and the population size. We got that the protocol is well-specified in these cases, and that it computes the predicate $|C|_{1_1} \geq N$. The limitation of this approach is that as there are infinitely many sizes and therefore initial configurations, hence we can never fully prove correctness of the protocol using the approach implemented in the tool. It is nevertheless useful to get some first confidence in the correctness of the protocol, i.e. that it is correct for at least some sizes and values of N.

In addition to using the tool to check for correctness, we used Peregrine [16], a tool that is capable of verifying correctness of silent population protocols for all of the infinitely many inputs. For more information on how Peregrine works, see [10].

6.2.3 Proof of Correctness

Previously, we have used the tool to get a first check that for some given sizes, the prime-flock-protocol is well-specified, and computes the correct function. In the following, we give a formal proof of correctness.

Let the value of a configuration C be the sum of the values of all agents in that configuration, i.e. $val(C) = \sum\limits_{s \in C} val(s) \cdot |C|_s$.

We now want to prove that the protocol computes the function $f(X) = 1$ if $|X|_1 \geq N$, else 0. First, observe that the only state with output 1 is 1_{n+1}. Therefore, for all configurations where no agent is in state 1_{n+1}, we know that the configuration is in a consensus with value 0.

Our goal is to prove the following:

Proposition 3. *For every fair execution $C_0C_1C_2\dots$, there is some i such that for all $j \geq i$, $|C_j|_{1_{n+1}} = |C_j|$ if and only if $val(C_0) \geq N$.*

Every fair execution will converge to output 1 if and only if the value of the initial configuration of that execution is larger or equal to N.

Note that due to ι mapping input 1 to a state with value 1 (in the form of 1_1) and input 0 to a state with value 0, proving Proposition 3 is equivalent to showing that the execution will converge to output 1 if and only if the predicate f is satisfied. We will see that it is then easy to show that the execution will converge to output 0 if and only if the predicate f is not satisfied.

To prove this, we first prove the following:

Lemma 2. *For all configurations C and C': $|C|_{1_{n+1}} = 0$ and $C \to C'$ implies $val(C) = val(C')$*

Proof. Let C, C' be configurations such that $|C|_{1_{n+1}} = 0$ and $C \to C'$. We then show that $val(C) = val(C')$.

Note that if $C \xrightarrow{T} C'$ via a silent transition T, i.e. $C' = C$, this is trivially satisfied, as $val(C) = val(C)$ holds. Therefore, focus on $C \xrightarrow{T} C'$ for a non-silent transition T. The protocol has three classes of non-silent transitions:

(I) transitions of the form $1_{n+1}, * \to 1_{n+1}, 1_{n+1}$. Note that since $|C|_{1_{n+1}} = 0$, transitions of this form cannot occur.

(II) transitions of the form $i_k, j_k \to (i+j)_k, 0$ that occur if agents on level $k \leq n$ with $i + j < p_k$ interact.

We have:

$$\begin{aligned} val(i_k) + val(j_k) &= i \cdot p_1 \cdots p_{k-1} + j \cdot p_1 \cdots p_{k-1} \\ &= (i+j) \cdot p_1 \cdots p_{k-1} \\ &= (i+j) \cdot p_1 \cdots p_{k-1} + 0 \\ &= val((i+j)_k) + val(0). \end{aligned}$$

Therefore, we have shown that if $C \xrightarrow{i_k, j_k \to (i+j)_k, 0} C'$, then $val(C) = val(C')$.

(III) transitions of the form $i_k, j_k \to 1_{k+1}, (i+j-p_k)_k$ that occur if agents on level $k \leq n$ with $i+j \geq p_k$ interact.

We have:

$$\begin{aligned} val(1_{k+1}) + val((i+j-p_k)_k) &\\ &= 1 \cdot p_1 \cdots p_k + (i+j-p_k) \cdot p_1 \cdots p_{k-1} \\ &= 1 \cdot p_1 \cdots p_k + (i+j) \cdot p_1 \cdots p_{k-1} - p_k \cdot p_1 \cdots p_{k-1} \\ &= (i+j) \cdot p_1 \cdots p_{k-1} \\ &= i \cdot p_1 \cdots p_{k-1} + j \cdot p_1 \cdots p_{k-1} \\ &= val(i_k) + val(j_k). \end{aligned}$$

Therefore, if $C \to C'$ and $|C|_{1_{n+1}} = 0$, then $val(C) = val(C')$. □

Next, we show the following:

Corollary 1. *For every execution $C_0C_1C_2\ldots$, if there exists some t such that $|C_t|_{1_{n+1}} > 0$, then $val(C_0) \geq N$.*

Proof. Let $C_0C_1C_2\ldots$ be an execution such that $|C_t|_{1_{n+1}} > 0$ for some t. Without loss of generality, assume t is minimal. Note that if $t = 0$, then it follows that $|C_0|_{1_{n+1}} > 0$. This implies that $val(C_0) \geq N$, since $val(1_{n+1}) = N$. Therefore, assume $t \geq 1$.

First, note that $val(C_t) \geq N$ since $|C_i|_{1_{n+1}} \geq 1$ and $val(1_{n+1}) = p_1 \cdot p_2 \cdots p_n = N$.

Recall that for all executions, for all k, it holds that $C_k \to C_{k+1}$. Additionally, since t is minimal it holds that $|C_j|_{1_{n+1}} = 0$ for all $j < t$. Then by Lemma 2, we have shown that for all j with $0 \leq j < t$, it holds that $val(C_j) = val(C_{j+1})$. It follows that $val(C_0) = val(C_1) = \cdots = val(C_{t-1}) = val(C_t) \geq N$. □

Note that since all states but state 1_{n+1} have output 0, if $val(C_0) < N$, we know the execution has already converged to output 0, since there cannot be a configuration C_i with $C_0 \xrightarrow{*} C_i$ with $|C_i|_{1_{n+1}} > 0$.

Now it remains to show that in all fair executions $C_0C_1C_2\ldots$, if $val(C_0) \geq N$, then there exists some i such that for all $j \geq i$, $|C_j|_{1_{n+1}} = |C_j|$ holds.

Denote by $val_j(C)$ the total value of all agents on level j in C, i.e. $val_j(C) = \sum_{0<i<p_j} val(i_j) \cdot |C|_{i_j}$. Similarly, we can denote the total value of multiple levels by

$$val_{j_0, j_1, \ldots, j_x}(C) = \sum_{i=0}^{x} val_{j_i}(C)$$

.

Let $C \to_j C'$ be the transition relation restricted to level j, i.e. $\{(C, C') | \exists t : C \xrightarrow{t} C' \wedge \exists i, i' : pre(t) = (i_j, i'_j)\}$.

We show the following:

Lemma 3. *For every fair execution $C_0C_1\dots$, if there is some i such that $|C_i|_{1_{n+1}} > 0$, then there exists some $j > i$ such that for all $k \geq j$, it holds that $|C_k|_{1_{n+1}} = |C_k|$.*

Intuitively, in all fair executions where there is one agent in state 1_{n+1}, the execution will from some point onwards only consist of configurations that contain only agents in state 1_{n+1}, and will therefore have converged to output 1.

Proof. Choose the smallest i such that the condition $|C_i|_{1_{n+1}} > 0$ is satisfied. Note that there is no transition of the protocol that changes the state of an agent in state 1_{n+1}, i.e. for all C, C' with $C \to C'$, it must hold that $|C|_{1_{n+1}} \leq |C'|_{1_{n+1}}$. Note also that for all configurations C with $1 \leq |C|_{1_{n+1}} \leq |C|$ (i.e. there is at least one agent in state 1_{n+1}, but also at least one agent in some other state s), there is a configuration C' with $|C'|_{1_{n+1}} = |C|_{1_{n+1}} + 1$ such that $C \xrightarrow{1_{n+1},s \to 1_{n+1},1_{n+1}} C'$ for some $s \neq 1_{n+1}$, simply by choosing for s one of the states present in the configuration other than 1_{n+1}. By doing this for each state different from 1_{n+1}, i.e. iteratively reducing the number if agents not in state 1_{n+1}, we can also reach a configuration F containing only agents in state 1_{n+1}, i.e. $|F|_{1_{n+1}} = |F|$. Since F is reachable from all configurations C with $1 \leq |C|_{1_{n+1}} \leq |C|$, together with the fact that for all C, C', $C \xrightarrow{*} C'$ implies that $|C|_{1_{n+1}} < |C'|_{1_{n+1}}$, we have that F is reachable infinitely often from any execution that satisfies the condition $\exists i : |C_i|_{1_{n+1}} > 0$. Fairness then implies that F must occur infinitely often in the execution. We conclude the proof by noting that for all configurations F' such that $F \to F'$ it must hold that $F = F'$, as both must consist only of agents in state 1_{n+1}. □

Due to this, we know that it is sufficient to show that if in a fair execution, a configuration with at least one agent in state 1_{n+1} is reached to show that in the execution, a consensus with value 1 is reached.

Next, we prove another lemma:

Lemma 4. *For all configurations C, C' and all levels k and j, with $j < k$: If $C \xrightarrow{*}_k C'$, then $val_j(C) = val_j(C')$.*

Informally, this means transitions between agents on some level k do not influence agents on levels $< k$.

Proof. Observe that $\to_k$ consists only of transitions between agents on level k. There are two classes of transitions in $\to_k$:

(I) transitions of the form $i_k, j_k \to (i+j)_k, 0$. Observe that $lvl(i_k) = k$, $lvl(j_k) = k$, $lvl((i+j)_k) = k$, $lvl(0) = 0$ and $val(0) = 0$, i.e. when $C \xrightarrow{i_k,j_k \to (i+j)_k,0} C'$, then for all $j < k$, $val_j(C) = val_j(C')$.

(II) transitions of the form $i_k, j_k \to 1_{k+1}, (i+j-p_k)_k$. Observe that $lvl(i_k) = k, lvl(j_k) = k, lvl(1_{k+1}) = k+1$ and $lvl((i+j-p_k)_{k+1}) = k$. Therefore, it can be that $val_{k+1}(C) \neq val_{k+1}(C')$ and $val_k(C) \neq val_k(C')$, but for all $j < k$, $val_j(C) = val_j(C')$.

□

We need another lemma, with which we will show that transitions will always lead to a configuration with equal or greater value:

Lemma 5. *For all transitions $t \in \delta$, for all configurations C, C', if $C \xrightarrow{t} C'$, then $val(C) \leq val(C')$.*

Proof. Note that in Lemma 2 we have shown that $val(C) = val(C')$ for transitions of the forms $i_k, j_k \to (i+j)_k, 0$ and $i_k, j_k \to 1_{k+1}, (i+j-p_k)_k$. This satisfies $val(C) \leq val(C')$, so we reuse this result here. Note that even though we made assumptions in Lemma 2 that do not need to hold here (i.e. that $|C|_{1_{n+1}} = 0$), we did not use these assumptions for the cases we want to reuse here, so the argumentation remains valid.

It remains to show this for transitions of the form $1_{n+1}, * \to 1_{n+1}, 1_{n+1}$, i.e. we want to show that for all states s and for all configurations $C, C' : C \xrightarrow{1_{n+1}, s \to 1_{n+1}, 1_{n+1}} C'$ implies $val(C) \leq val(C')$. Note that for all states s, $val(s) \leq val(1_{n+1})$. Then:

$$val(1_{n+1}) + val(s) \leq val(1_{n+1}) + val(1_{n+1}).$$ □

We will, in the next steps, prove that in all fair executions, we can restrict the values on levels smaller than $n+1$ to a certain threshold:

Lemma 6. *For all configurations C and for all levels k so that $k \leq n$, there is some configuration C' with $C \xrightarrow{*}_k C'$ and $val_k(C') < p_k \cdot p_{k-1} \cdots p_1$.*

Proof. For $val_k(C) < p_k \cdots p_1$, this is trivially satisfied. Assume therefore $val_k(C) \geq p_k \cdot p_{k-1} \cdots p_1$. We again consider the transitions in $\xrightarrow{*}_k$, and see that there are two classes:

(I) Transitions of the form $i_k, j_k \to 1_{k+1}, (i+j-p_k)_k$. Observe that this transition only occurs when $i + j > p_k$. Then we know that for all C, C' if $C \xrightarrow{i_k, j_k \to 1_{k+1}, (i+j-p_k)_k} C'$, then

$$\begin{aligned}
&val_k(C') = \\
&val_k(C) - (val(i_k) + val(j_k)) + val(i+j-p_k)_k = \\
&val_k(C) - (i \cdot p_{k-1} \cdots p_1 + j \cdot p_{k-1} \cdots p_1) + (i+j-p_k) \cdot p_{k-1} \cdots p_1 = \\
&val_k(C) - (i+j) \cdot p_{k-1} \cdots p_1 + (i+j) \cdot p_{k-1} \cdots p_1 - p_k \cdot p_{k-1} \cdots p_1 = \\
&val_k(C) - p_k \cdot p_{k-1} \cdots p_1 < val_k(C)
\end{aligned}$$

Therefore, by this transition, $val_k(C') < val_k(C)$.

(II) Transitions of the form $i_k, j_k \to (i+j)_k, 0$. Then for all C, C' when $C \xrightarrow{i_k, j_k \to (i+j)_k, 0} C'$, then $val_k(C) = val_k(C')$. Therefore, the value on level k remains the same, but $\sum_{0<i<p_k} |C'|_{i_k} = \sum_{0<i<p_k} |C|_{i_k} - 1$, so in C', there is one less agent on level k than in C.

Now, we can construct a sequence of transition sequences $t_0, t_1, t_2, \ldots, t_\ell \in \overset{*}{\rightarrow}_k$ such that $C \xrightarrow{t_0} C_0 \xrightarrow{t_1} C_1, \ldots, C_{\ell-1} \xrightarrow{t_\ell} C'$, with $val_k(C') < p_k$ as follows:

If there are no agents in states $i_k, j_k \in C$ such that $i + j \geq p_k$, but $val_k(C) \geq p_k \cdots p_1$, use a transition of the form $i_k, j_k \rightarrow (i+j)_k, 0$. As explained above, this leads to one less agent on level k.

If there are agents in states $i_k, j_k \in C$ such that $i + j \geq p_k$, use transition $i_k, j_k \rightarrow 1_{k+1}, (i+j-p_k)_k$, with the next configuration then being C' with $C \xrightarrow{i_k, j_k \rightarrow 1_{k+1}, (i+j-p_k)_k} C'$. As shown above, $val_k(C') < val_k(C)$.

Combining these two cases, observe that as long as $val_k(C) \geq p_k \cdots p_1$, we can first apply transition $i_k, j_k \rightarrow (i+j)_k, 0$ to reduce the number of agents on level k, but retain the same value on that level. We can repeat this until there are two agents in states i_k, j_k with $i + j \geq p_k$, and then use transition $i_k, j_k \rightarrow 1_{k+1}, (i+j-p_k)_k$. By Lemma 4, these transitions do not change the value of levels smaller than k. This procedure can be repeated until $val_k(C) < p_k \cdots p_1$. □

Combined, the lemmas we have proven give the following:

Corollary 2. *For every configuration C, if $val(C) \geq N$, then there is a configuration C' such that $C \overset{*}{\rightarrow} C'$ and $|C'|_{1_{n+1}} > 0$.*

Proof. Let C be a configuration so that $val(C) \geq N$.

By Lemma 6, we know that there is a configuration C_1 so that $C \overset{*}{\rightarrow}_1 C_1$, where $val_1(C_1) < p_1$. We then apply Lemma 6 to C_1, and know that we can obtain a configuration C_2 so that $C_1 \overset{*}{\rightarrow}_2 C_2$, and $val_2(C_2) < p_2$. By Lemma 4, we know that we can find such a configuration C_2 that $val_1(C_2) = val_1(C_1) < p_1$. We repeatedly apply Lemmas 6 and 4, by which we know that there are configurations $C_1, C_2, \ldots, C_n = C'$ where for configuration C', it holds that for all levels k with $k \leq n+1$, and $C \overset{*}{\rightarrow}_1 C_1 \overset{*}{\rightarrow}_2 C_2 \overset{*}{\rightarrow}_3 \ldots \overset{*}{\rightarrow}_n C'$.

Therefore, we know that for configuration C', it holds that for all levels $k < n+1$, $val_k(C') < p_k \cdots p_1$. Observe that for all transitions $t \in \delta$, it holds that for all C, C', if $C \xrightarrow{t} C'$ then $val(C) \leq val(C')$, as per Lemma 5. Therefore, $val(C') \geq val(C) \geq N$. Observe also that agents on level k have value as multiples of $p_1 \cdots p_{k-1}$ by definition. Therefore, we know that since for all levels k, $val_k(C') < p_k \cdots p_1$, this is equivalent to having that $val_k(C') \leq (p_k - 1) \cdot p_{k-1} \cdots p_1$. From this, it follows that:

$$\begin{aligned}
&val_{0,1,\ldots,n}(C') = 0 + val_1(C') + \cdots + val_n(C') \leq \\
&(p_1 - 1) + (p_2 - 1) \cdot p_1 + \cdots + (p_n - 1) \cdot p_{n-1} \cdots p_1 = \\
&p_1 - 1 + p_2 \cdot p_1 - p_1 + \cdots + p_n \cdot p_{n-1} \cdots p_1 - p_{n-1} \cdots p_1 = \\
&p_n \cdot p_{n-1} \cdots p_1 - 1
\end{aligned}$$

Therefore $val_{0,1,\ldots,n}(C') \leq p_n \cdots p_1 - 1 = N - 1$. Since $val(C') \geq N$ and $val_{0,1,\ldots,n}(C') + val_{n+1}(C') = val(C')$, we know that $val_{n+1}(C') \geq 1$. As the only state on level $n+1$ is 1_{n+1}, it follows that $|C'|_{1_{n+1}} \geq 1$. □

Proposition 4. *For every fair execution $C_0C_1\dots$, if $val(C_0) \geq N$, there is some i such that for all $j > i$, $|C_j|_{1_{n+1}} > 0$.*

Proof. Let $E = C_0C_1\dots$ be a fair execution. By Corollary 2, we know that from C_0, a configuration C' is reachable satisfying $val_k(C') < p_k$ for all levels k. Assume for contradiction that in E, there is no such configuration C', i.e. there is infinitely many j such that C_j satisfying that for some level k, $val_k(C_j) > p_k \cdots p_1$. Observe that there is only finitely many configurations that satisfy this criterion, as configurations have a fixed size and there are only finitely many states for agents. Therefore, there must be at least one configuration that must occur infinitely often in the E, by the pigeonhole principle. Denote by K one such configuration that occurs infinitely often K. Observe that from Corollary 2, we can follow that $\exists C' : K \xrightarrow{*} C'$ and $|C'|_{1_{n+1}} \geq 1$. Let C' be such a configuration. Assume that C' does not occur in E. This contradicts fairness, since K occurs infinitely often in E, and $K \xrightarrow{*} C'$, but C' does not occur infinitely often in E. Therefore, C' must occur in E. Then, by Lemma 3, we have proven that for every fair execution $C_0C_1\dots C_{l-1}C'C_{l+1}\dots$, there is some $o > l$ such that for all $m \geq o$, it holds that $|C_m|_{1_{n+1}} = |C_m|$. □

Theorem 2. *The prime-flock-protocol is correct.*

Proof. By Corollary 1 and Lemma 4, we have shown that for all fair executions $C_0C_1\dots$, the execution converges to output 1 if and only if $val(C_0) \geq N$. Corollary 1 on its own is sufficient to show that for every fair execution $C_0C_1\dots$, the execution converges to output 0 if and only if $val(C_0) < N$. We conclude our proof by noting that due to the choice of the input function ι, for every multiset $X \in pop(\Sigma)$ and the corresponding initial configuration C_0 it holds that $val(C_0) = |X|_1$. □

6.2.4 Comparison of Convergence Properties with Existing Protocols

To get an idea of how the convergence time of the prime-flock-protocol behaves compared with the convergence time of the existing protocols for the flock of birds predicate, we used our tool to measure average steps until a stable configuration is reached, similar to the measurements of Section 6.1. The protocols compared with the prime-flock-protocol can be found in Section 3.3.4.

Parameters. For this test, we measured average steps until a stable configuration is reached for population sizes 5 up to 15. For each size, we use the population where all agents have input 1, since all protocols treat agents with input 0 in a very similar way, but they slow down measurements. Additionally, all protocols start in a stable configuration when the input values do not sum up to at least N, therefore, for population size S, we chose $N = S - 1$.

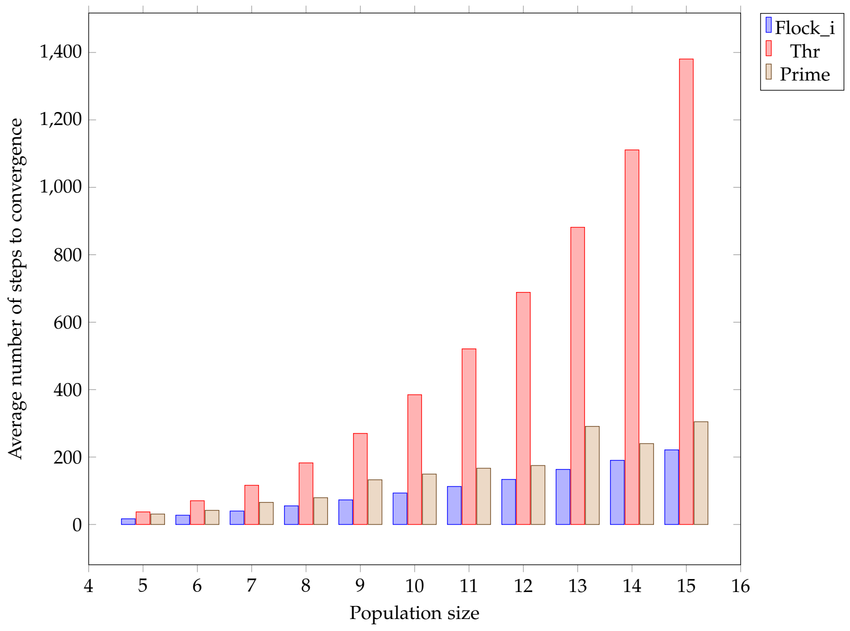

Figure 6.5: Comparison of protocols for flock of birds, using uniform rules scheduling.

Procedure. We used the tool to generate the protocols, then simulate them until a stable configuration is reached. More details on the simulation process can be found in Section 5.4. We did separate measurements for both uniform rules scheduling and uniform pairs scheduling. Lastly, we simulated 1000 runs per population, computed the average number of steps until convergence, and compute the standard deviation. Since the population sizes are rather small, 1000 runs per population are enough to get a good approximation of the actual expected steps.

Results. The measured values under uniform rules scheduling can be seen in Figure 6.5, whereas the results for uniform pairs scheduling can be seen in Figure 6.6. Note that as for many of the majority protocols, uniform rules scheduling gives faster convergence. Moreover for uniform rules scheduling, the prime-flock-protocol is the protocol with the best convergence times, while for uniform pairs scheduling, this is the $flock_i$ protocol. We suspect that this is the case as for uniform rules scheduling, it leads to better convergence behaviour if every enabled transition makes progress. The prime-flock-protocol typically has few transitions enabled, since only agents on the same level interact, whereas for the $flock_i$ protocol, all non-zero agents can interact in each step, which might mean transitions are chosen that do not make progress.

For uniform pairs scheduling, all protocols exhibit larger convergence times, due to agents in state 0 not taking part in any non-silent transitions until there is an agent in state N, and the agents in state 0 then get converted to the correct output.

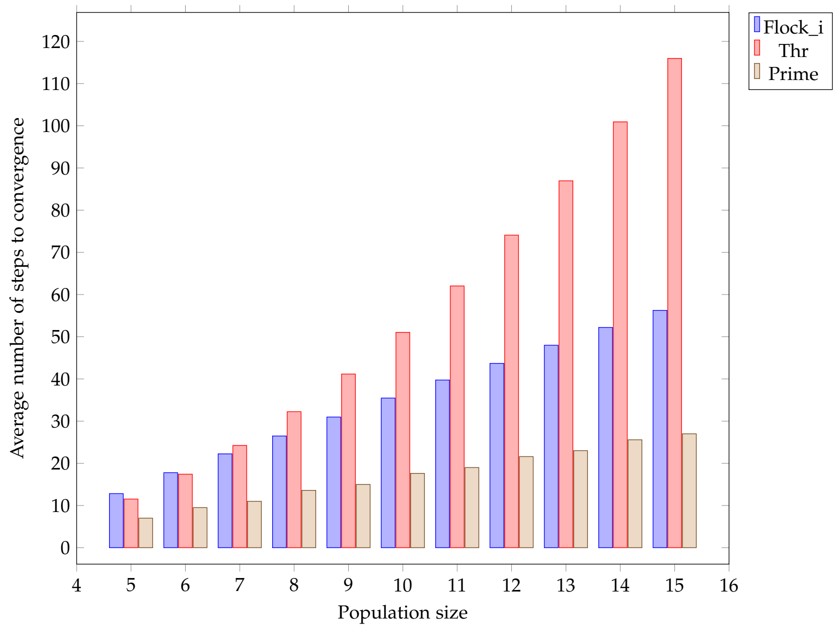

Figure 6.6: Comparison of protocols for flock of birds, using uniform pairs scheduling.

7 Conclusion and Future Work

While some tools for the verification of population protocols exist, in this thesis, we presented a new tool that is not only capable of verifying properties such as well-specification or average number of steps to convergence, but can also simulate runs of a protocol. In addition, we gave two case studies for the tool. In the first study, we measured two properties: average number of steps to convergence and failure ratio of different protocols. As the study objects, we used different protocols computing the majority predicate. In the second study, we considered protocols for the flock-of-birds predicate. A downside of existing protocols is that they have an amount of states linear in the threshold of the predicate. We proposed the *prime-flock protocol*, a protocol that computes the flock-of-birds predicate and uses less states than existing protocols. For a given threshold N with prime factorization $N = p_1 \cdot ... \cdot p_n$, existing protocols need a number of states in $O(N)$, while the prime-flock-protocol need a number of states in $O(\sum_{i=1}^{n} p_i)$. Further, we used our tool to gain confidence in the correctness of the prime-flock-protocol by verifying its correctness for several sizes. Lastly, we gave a formal proof of correctness as well as measurements for the average number of steps to convergence of the different protocols that compute the flock-of-birds predicate.

Some open research questions and opportunities for further development of the tool remain:

Finding a lower bound of states for the flock-of-birds predicate. While we have given a protocol that generally needs less states than existing protocols, the lower bound of states for protocols computing this predicate remains an open question.

Exporting protocols using arbitrary distributions in the tool without generating the reachability graph. For exporting a protocol to PRISM to verify it, there exist multiple approaches in the tool. We have two approaches for uniform rules scheduling that do not rely on the reachability graph. On the other hand, the tool needs to generate the whole reachability graph for a given population when one wants to use arbitrary probability distributions. There might be a way to extend the deterministic encoding for uniform rules scheduling so that it can simulate arbitrary distribution, but it remains an open task to find such a way and implement it.

Implementing support for parallel steps. Currently, the tool does not have a way to simulate runs where not only a single pair of agents interacts in a step, but where as many pairs as possible interact at once. As it is common in the literature to specify the average number of steps to convergence as the number of parallel steps, this could be a task for future development.

Investigate the connection between probability distributions and convergence behaviour. We presented some experimental results on the convergence times of several protocols with different probability distributions. It remains open to find classes of population protocols that behave especially favourably with some distributions.

List of Figures

5.1 Illustration of the approach for deterministic encoding of uniform rules scheduling . . . 30

6.1 Comparison of average number of steps to convergence of protocols for the majority problem, using uniform pairs scheduling. . . . 37
6.2 Comparison of average number of steps to convergence of protocols for the majority problem, using uniform rules scheduling. . . . 37
6.3 Average number of steps until convergence for the 3-state majority protocol with different probability distributions. . . . 40
6.4 Failure ratio for the 3-state majority protocol with different probability distributions. . . . 40
6.5 Comparison of protocols for flock of birds, using uniform rules scheduling. 49
6.6 Comparison of protocols for flock of birds, using uniform pairs scheduling. 50

Bibliography

[1] D. Alistarh, R. Gelashvili, and M. Vojnović. “Fast and exact majority in population protocols.” In: *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*. ACM. 2015, pp. 47–56.

[2] D. Angluin, J. Aspnes, Z. Diamadi, M. J. Fischer, and R. Peralta. “Computation in networks of passively mobile finite-state sensors.” In: *Distributed computing* 18.4 (2006), pp. 235–253.

[3] D. Angluin, J. Aspnes, Z. Diamadi, M. J. Fischer, and R. Peralta. *Urn automata*. Tech. rep. YALE UNIV NEW HAVEN CT DEPT OF COMPUTER SCIENCE, 2003.

[4] D. Angluin, J. Aspnes, and D. Eisenstat. “A simple population protocol for fast robust approximate majority.” In: *Distributed Computing* 21.2 (2008), pp. 87–102.

[5] D. Angluin, J. Aspnes, and D. Eisenstat. “Fast computation by population protocols with a leader.” In: *Distributed Computing* 21.3 (2008), pp. 183–199.

[6] D. Angluin, J. Aspnes, and D. Eisenstat. “Stably computable predicates are semilinear.” In: *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*. ACM. 2006, pp. 292–299.

[7] D. Angluin, J. Aspnes, M. J. Fischer, and H. Jiang. “Self-stabilizing population protocols.” In: *International Conference On Principles Of Distributed Systems*. Springer. 2005, pp. 103–117.

[8] J. Aspnes and E. Ruppert. “An introduction to population protocols.” In: *Middleware for Network Eccentric and Mobile Applications*. Springer, 2009, pp. 97–120.

[9] C. Baier, J.-P. Katoen, and K. G. Larsen. *Principles of model checking*. MIT press, 2008.

[10] M. Blondin, J. Esparza, S. Jaax, and P. J. Meyer. “Towards Efficient Verification of Population Protocols.” In: *Proc. 36th ACM Symposium on Principles of Distributed Computing (PODC)*. 2017, to appear.

[11] I. Chatzigiannakis, O. Michail, and P. G. Spirakis. “Algorithmic verification of population protocols.” In: *Symposium on Self-Stabilizing Systems*. Springer. 2010, pp. 221–235.

[12] J. Clément, C. Delporte-Gallet, H. Fauconnier, and M. Sighireanu. “Guidelines for the verification of population protocols.” In: *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*. IEEE. 2011, pp. 215–224.

[13] Z. Diamadi and M. J. Fischer. “A simple game for the study of trust in distributed systems.” In: *Wuhan University Journal of Natural Sciences* 6.1 (2001), pp. 72–82.

[14] D. T. Gillespie. "Exact stochastic simulation of coupled chemical reactions." In: *The journal of physical chemistry* 81.25 (1977), pp. 2340–2361.

[15] JetBrains. *Webpage of the PyCharm Integrated Development Environment*. 2017. URL: `https://www.jetbrains.com/pycharm/`.

[16] P. Meyer and S. Jaax. *Peregrine*. 2017. URL: `https://gitlab.lrz.de/i7/peregrine`.

[17] M. Presburger. "Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchen die Addition als einzige Operation hervortritt." In: *Comptes-Rendus du ler Congrès des Mathématiciens des Pays Slaves* (1929), pp. 92–101.

[18] *PRISM Manual*. May 2016. URL: `http://www.prismmodelchecker.org/manual/`.

[19] Python Software Foundation. *Webpage of the Python Programming Language*. 2017. URL: `https://www.python.org/`.

[20] J. Sun, Y. Liu, J. S. Dong, and C. Chen. "Integrating specification and programs for system modeling and verification." In: *Theoretical Aspects of Software Engineering, 2009. TASE 2009. Third IEEE International Symposium on*. IEEE. 2009, pp. 127–135.

[21] R. Tarjan. "Depth-first search and linear graph algorithms." In: *SIAM journal on computing* 1.2 (1972), pp. 146–160.