Performance Evaluation of Self-stabilizing Algorithms by Probabilistic Model Checking

by

Narges Fallahi

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2014

Abstract

A self-stabilizing protocol is one that starting from any arbitrary initial state recovers to legitimate states in a finite number of steps, and once it stabilizes to a set of legitimate states, it remains there unless it is perturbed by transient faults. The traditional methods existing for performance evaluation of a self-stabilizing algorithm usually work based on the analysis of worst case computational complexity. Another method that has been commonly used in evaluating these algorithms is simulation, which assumes the system starts from an initial state. Here, it is argued that the traditional methods have shortcomings and do not give enough insight about the behavior of the system. Moreover, they do not provide a decent method of comparison. We propose a novel method for evaluation of self-stabilizing algorithms. This method works based on probabilistic model checking and computation of the expected number of recovery steps. We execute some experiments on the case studies, and the results indicate that we can gain insight about the faults and their structure in the protocol.

Next, we explain the difficulty of designing a self-stabilizing algorithm for a system and show how it is impossible to do so for some classes of protocols. This resulted in some relaxation in the definition of self-stabilization. One of the relaxations made in the definition of self-stabilization is weak-stabilization. A weak-stabilizing protocol ensures the existence of a recovery path from an arbitrary initial configuration. Thus, some paths may contain connected components or cycles. Since a weak-stabilizing algorithm may get stuck in connected components forever, we cannot evaluate weak-stabilizing protocols by traditional and existing methods. We calculate the expected number of recovery steps for evaluating weak-stabilization. However, since it does not give us enough intuition about the structure of faults, we apply a graph-theoretic formula for estimating the weak-stabilizing algorithm's performance. This formula is based on the number of cycles and their reachability.

Based on the observations we made by performance evaluation of these protocols, we suggest algorithms called *state encoding* for modifying the performance of the algorithms. State encoding works based on changing the bit mapping of the states of the system. The aim is to make the states with faster recovery steps more probable to occur. There are three algorithms, one of which works based on betweenness centrality which is a measure of centrality of a node within a graph. The other one works based on feedback arc set which is a set of arcs whose removal makes a graph acyclic. The third algorithm works based on the length of the shortest recovery path for the states.

The other problem investigated here is the problem of state space explosion in model checking. Similar to traditional methods of model checking, probabilistic model checking also suffers from the problem of state space explosion, i.e., the number of states grows exponentially in terms of the number of components in the distributed system. Abstraction methods, which are described briefly here, are designed to combat this problem. We argue that they are not efficient enough, and there is still the lack of a sufficient abstraction method that works for systems with an arbitrary number of processes. We also propose a new approach for evaluation of an abstraction function. Then, based on the intuition gained, a new abstraction algorithm is proposed that is exclusively designed for verification of reachability properties. After executing experiments on a case study, we compare the result of our algorithm with the results obtained by existing methods. The results support our claim that our method is more efficient and precise.

Acknowledgements

I would like to thank my dear supervisor, professor Naomi Nishimura, for her endless and precious helps and dedications, that without her, writing this thesis was not possible. Then, I would like to thank professor Richard Trefler, for his precious comments on my work, and being so supportive. I also would like to thank Professor Joanne Atlee for reading my thesis and her priceless comments on it. I would also like to thank my father, Hossein Fallahi, who made this path possible in my life for a bright future for me. Being far away from the family is not easy, and I am so grateful to have a quite nice one. I would like to thank my mother, Fatemeh Moshiri, for her being so supportive and dedicated. She has always encouraged me to study, and regardless of how hard it was for her, she let me to come and study far apart. I would also like to thank my little sister, Mina Fallahi, for always making me laugh, and make me motivated to continue. At the end, I would like to thank my dear husband, Saman Nazari Nejad, for his unconditional love and inspiration, being awake some nights with me to get my work done, and helping me as much as he could.

Dedication

I would like to dedicate this work to my husband, my parents, and my sister to show my appreciation for their unconditional love and sacrifice.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction and Related Work

In this chapter, we first introduce the concept of *self-stabilization* and some self-stabilizing algorithms. Then, we explain some existing methods for evaluating the performance of self-stabilizing algorithms, and then express our own way for doing so. Next, we introduce some relaxations on the definition of self-stabilization. One of them on which we focus in this thesis is weak-stabilization. We introduce the concept and describe our way of evaluating the performance of a weak-stabilizing algorithm. Then, we discuss the concept of *state encoding* as a way of improving the performance of self-stabilization and weak-stabilization. Finally, we describe the *state explosion problem*, and explain some existing methods for dealing with this problem. Then, we explain our own method for tackling this problem.

1.1 Self-stabilization

Self-stabilization [1, 2, 3] is a versatile technique for achieving fault recovery. In 1993, Arora and Gouda [4] presented a formal definition of a fault tolerant system. Fault tolerance ensures that when the system is hit by faults and driven to some arbitrary state [5], it is guaranteed to recover to a set of legitimate states within a finite number of execution steps. Once the system reaches such set, it remains in this set thereafter in the absence of new faults. Within the past decades, the initial concept has matured and expanded into various problems [6, 7, 8, 9], and networks [10], [11], [12], [13].

Dijkstra [1] introduced three uniform self-stabilizing algorithms for the *token ring problem*, which as defined in Definition 33, is a ring of processes in which a token is passed between the processes such that eventually only one process has the token. The mutual exclusion problem first introduced by Dijkstra [14] in 1965, as defined in Definition 35, given a distributed system consisting of a set of processes and a property known as critical section, the mutual exclusion problem is defined as to reach a set where there is only one process which holds the critical section. His work did not receive much attention until in 1983, Leslie Lamport [15] mentioned this paper as the most brilliant work of Dijkstra and as a milestone for fault recovery problems. Kessels in 1988 [16] proved the correctness of the three-state algorithm of Dijkstra. This algorithm appears in Algorithm 4. In 1989, Burns and Pachl [17] proposed a self-stabilizing algorithm for the mutual exclusion problem on a ring graph with prime size. The ring graph as defined in Definition 3, is a graph in which nodes are organized as a ring.

In 1994 Flatebo and Datta [18] introduced the concept of a *randomized central daemon* which randomly chooses a possible path to get executed, and proposed a two-state self-stabilizing algorithm for the mutual exclusion problem where a two-state self-stabilizing algorithm is one in which each process has two possible states. Then, in 1995 Beauquier and Debas [19] proposed a modification of the three-state self-stabilizing algorithm of Dijkstra such that the new algorithm has less computational complexity than Dijkstra's. In this algorithm, two new lines are added to Dijkstra's algorithm.

The other problem for which several self-stabilizing algorithms have been suggested is the leader election problem. This problem, as defined in Definition 34, requires that when the algorithm terminates, one process is distinguished as the leader and every process in the system knows whether it is the leader or not [20] [21]. Before Dolev et al. in 1997 [20], some non-self-stabilizing algorithms for the leader election problem had been proposed [22, 23].

Another problem that is also considered as a case study in this thesis is the propagation of information with feedback (PIF) problem introduced by Chang [24] and Segal [25]. In this problem, as defined in Definition 32, a tree structure is given. Initially, all the processes are idle. Starting from the root, a message request propagates through neighbors of each node to a leaf; then the leaf sends a response message and this response goes up until it reaches the root. Then, all the nodes become idle again. After that, several self-stabilizing PIF algorithms were proposed for this problem [26] [27] [28]. Then, in 2001 Cournier et al. [29] proposed a deterministic self-stabilizing PIF protocol that works for an arbitrary tree structure.

1.2 Evaluation of Self-stabilization

As interest in designing self-stabilizing algorithms has increased, the number of self-stabilizing algorithms has also increased. However, there was a lack of an adequate way to compare and analyze the performance of self-stabilizing algorithms. The conventional approaches employed for performance evaluation of a self-stabilizing algorithm have been mainly based on how fast it recovers to a set of legitimate states. The conventional metric to evaluate the performance is asymptotic computational complexity in terms of the number of rounds [20] (i.e., the shortest computation in which each process executes at least one step) or waiting time (i.e., the number of global actions that have to be executed). Thus, one can express the performance of a self-stabilizing algorithm by computing the upper bound as the worst case number of, e.g., rounds. For instance, if the topology of a distributed system is a tree, then the performance could be a polynomial in h rounds, where h is the height of the tree.

The main flaw in this approach is that it assumes that all the faults in the system happen with the same probability and so, the probability distribution over the state space is uniform. However, in practice, usually some faults are more probable than others. On the other hand, we have practical ways of evaluation, such as simulation [9], which uses experimental methods to measure the efficiency of a given self-stabilizing algorithm. Moreover, we can use simulation as a measure of comparison for self-stabilizing algorithms.

Based on arguing that the existing methods do not provide us enough insight about the structure of the faults and behavior of the system, Fallahi and Bonakdarpour [30] proposed a new method for evaluation of self-stabilizing algorithms that works based on probabilistic model checking [31]. In this method, it is sufficient to compute the *expected number of recovery steps* for all the possible states outside the set of legitimate states. It can also be used as a means of comparison for self-stabilizing algorithms.

In this thesis, we argue that using asymptotic computational complexity for evaluating the performance is too abstract and does not accurately reflect the realities of deploying a distributed self-stabilizing algorithm in practice. More specifically, asymptotic computational complexity does not investigate factors that can be potentially crucial in practice (constants and smaller polynomials may matter but, most importantly, the worst case complexity may be irrelevant in practice).

On the other hand, practical performance evaluation of self-stabilizing algorithms typically uses simulation [32, 33, 12, 9], but most approaches consider running the algorithm from an initial random state, which is known to introduce significant bias [9] and also raises the question of case coverage, since only a very small portion of possible states are encountered in a typical simulation run.

In general, the conventional performance evaluation metric is computing the asymptotic computational complexity in terms of round, cycle, and recovery steps, where

- A *round* [20] is the shortest computation fragment in which each process executes at least one step.
- A *cycle* is the shortest computation fragment in which each process executes at least one complete iteration of its repeatedly executed list of commands.
- *Number of recovery steps* is the number of global actions that must be executed to move from an arbitrary state to a legitimate state.

Shortcomings of the conventional method of performance evaluation of self-stabilizing algorithms are given as following:

- The constants removed in computing the asymptotic computational complexity can be large enough to affect the performance of the algorithm.
- The smaller polynomials removed in computing the asymptotic computational complexity can be large enough to have a significant impact on the performance of the algorithm.
- The worst case complexity may rarely occur. In other words, the asymptotic computational method assumes a uniform probability distribution over the state space. However, in practice, some faults may be more likely to happen.

We believe these abstractions result in the need for a novel method for evaluating the performance of self-stabilizing algorithm which is more precise. With this motivation, in this thesis we propose a new metric that characterizes the expected number of recovery steps of a self-stabilizing algorithm independently of the underlying network topology and communication technology (e.g., shared memory vs. message passing). The new metric is the expected number of recovery steps. This expected value is computed based on the sum of probabilities for n-step reachability of legitimate states for all possible values of n; i.e., the number of execution steps to reach a legitimate state from each arbitrary state. Our technique to compute the probability of n-step reachability and subsequently the expected number of recovery steps is based upon probabilistic model checking [31] which is defined as verification of a system by a probabilistic model checker. A probabilistic model checker takes the model of a (deterministic or non-deterministic) system and a set of probabilistic temporal properties as input and automatically proves whether the model satisfies the properties. In the context of our problem, a probabilistic temporal property is of the form "whether the probability of reachability of legitimate states from an arbitrary state within n execution steps is greater than a certain value". Equivalently, one can compute the probability of n-reachability of legitimate states from an arbitrary state, which modern probabilistic model checkers can do. Obviously, computing this probability implies the computation of the expected number of recovery steps. The expected number of recovery steps computed using this technique represents the overall performance of an algorithm, and can be used as a basis for comparing the performance of different algorithms. Another advantage of our technique is that it can elegantly take into account other parameters vital to evaluation of the performance of a self-stabilizing algorithm by simply incorporating them when building the model of an algorithm. Examples of these parameters include the impact of the underlying scheduler and the likelihood of occurrence of faults. While the former can significantly change the way an algorithm behaves during recovery, the latter determines the number of states and the probability of reaching states from which recovery is fast or slow. This number and probability can significantly impact the expected number of recovery steps. The suggested technique is described in Chapter 3.

To back our claims, we conducted thorough experiments using the PRISM model checker [34] and three well-known self-stabilizing algorithms for solving distributed propagation of information with feedback (PIF) in rooted trees in Chapter 3. Moreover, as another case study we considered two self-stabilizing algorithms for a token ring problem. Our experiments clearly show that an algorithm that has the best asymptotic performance does not perform the best under all possible scenarios. Thus, to implement and deploy a self-stabilizing algorithm in practice one has to assess the environment and the likelihood of its faults to make the best choice of algorithm. In other words, our results show that the worst case analysis by itself is not a good way for deployment of self-stabilizing algorithms in real-world situations.

We believe that our new metric and automated technique provide valuable insights into behavior and performance of self-stabilizing algorithms crucial for practical system deployment.

1.3 Relaxations on Definition of Self-stabilization

Designing a self-stabilizing algorithm has been always desirable for researchers. However, for some classes of problems, such as leader election with anonymous processes [35] (meaning the processes are indistinguishable in that they execute the same set of actions and their variables have same domain), it is impossible to have a self-stabilizing protocol. Thus, since then, some relaxations on the definition of self-stabilization have been proposed. One of them is probabilistic self-stabilizing protocols in which the recovery of the system to legal behavior is guaranteed with probability 1 [36]. Herman proposed a probabilistic self-stabilizing mutual exclusion protocol for the topology of a ring with an odd number of identical processes [36]. The impossibility of designing a self-stabilizing mutual exclusion protocol for identical processes was proved by Angluin [37].

Another relaxation proposed on the definition of self-stabilization is k-stabilization [38]. In this method of stabilization, some faults are assumed to be impossible to occur, and the recovery is ensured to happen within k local state changes for each process in the system. Beauquier et al. [38] suggested a k-stabilizing algorithm for the token ring problem.

The other type of self-stabilizing protocols is snap-stabilizing. The notion of snap-stabilization was introduced in 1999 [39] as a snap-stabilizing protocol ensures that the system always exhibits legal behavior. In other words, a snap-stabilizing algorithm is a self-stabilizing algorithm that stabilizes in zero steps.

Weak-stabilization is the most recent relaxation that has been introduced in the definition of self-stabilization [40]. A weak-stabilizing protocol ensures the existence of a recovery path from any arbitrary initial configuration. Thus, there may exist some paths that are reachable from a configuration and do not recover. They can be cycles or connected components in the graph representing the protocol. There was little attention paid to weak-stabilization because of this relaxation until Gouda [40] argued that if we could design a weak-stabilizing algorithm for a problem, it can be significantly faster than self-stabilization. In 2008, Devismes et al. [41], suggested a weak-stabilizing leader election protocol for anonymous processes.

1.4 Evaluation of Weak-stabilization

We argue that the existence of cycles by itself should not be a strong reason to reject weak-stabilizing algorithms as practical solutions to deal with fault recovery in distributed systems for the following reasons:

- In a weak-stabilizing algorithm, there always exists the chance of convergence. Hence, a weak-stabilizing algorithm in the presence of a fair scheduler (which guarantees that a continuously enabled action will eventually be chosen) is guaranteed to reach a legitimate state within a finite number of execution steps as the probability of leaving the cycle converges to 1 [41].

- Unlike self-stabilization, the scheduler is considered friendly. In fact, in order to prove that a system is not self-stabilizing, it is sufficient to find an execution that satisfies the property given by the scheduler but does not converge to legitimate states. Moreover, in practice, it is usually the case that the scheduler is probabilistic, and finally chooses an enabled action. In many commonly considered systems, actions happen with some probability. For example, in data networks, a packet reaches its destination with some probability. This implies that a cycle of actions that involves data re-transmission eventually ends. In fact, it can be shown that in the presence of the probabilistic scheduler, for systems with finite possible states, any weak-stabilizing algorithm is a probabilistic self-stabilizing algorithm [41] (i.e., the probability of reaching a correct behavior in weak-stabilization always converges to 1).

- Recovery in a weak-stabilizing algorithm heavily depends on the structure of the algorithm and, in particular, reachability and length of cycles outside the legitimate states. In other words, existence of cycle(s) in an algorithm by itself is not a good measure to judge the performance of a weak-stabilizing algorithm.

To explain the latter reason in more detail, consider the structures in Figure 1.1. We assume uniform probability for all the states in Figure 1.1. In Figure 1.1(b), the recovery path that starts from state s_2 reaches a cycle, but the one that starts from state s_1 does not act in the same way. Such an algorithm is expected to perform better than the one shown in Figure 1.1(c), since in the latter both recovery paths reach a cycle. Likewise, the algorithm in Figure 1.1(c) is expected to perform better than the one shown in Figure 1.1(d), as in the latter the path that starts from state s_2 reaches two cycles during recovery. Also, the algorithm in Figure 1.1(d) is expected to perform better than the one shown in Figure 1.1(e), since the latter reaches a cycle whose length is longer. On the other hand, it is difficult to compare the performance of the structures shown in Figures 1.1(d) and 1.1(f), since the structure in Figure 1.1(f) has a longer cycle, but the path that starts from state s_1 may avoid a cycle by branching to a different recovery sub-path. Another example is the case shown in Figure 1.1(a) where faults do not perturb the system to a state from which the cycle can be reached. In such a scenario the existence of the cycle becomes irrelevant.

The above analysis clearly shows that one cannot reject weak-stabilization based only on the argument of "existence of cycles during recovery". That is, the structure of a weak-stabilizing algorithm and the number and length of cycles that recovery paths visit play an important role in the performance of the algorithm. With this motivation and in the absence of performance metrics for weak-stabilization, in this thesis, we focus on developing

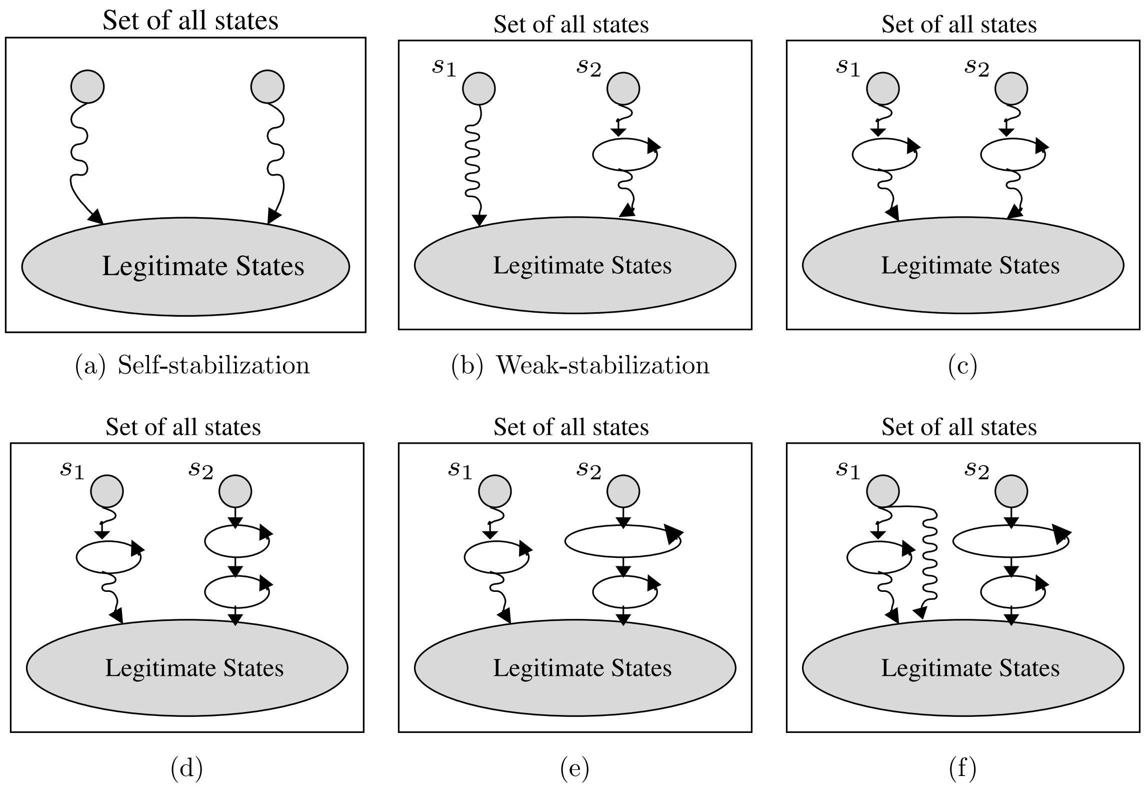


Figure 1.1: Self-stabilization and different algorithm structures for weak-stabilization.

such a metric. Our metric is based on the method introduced by Fallahi and Bonakdarpour for computing the expected number of recovery steps [30]. Unlike conventional asymptotic computational analysis (in terms of, for instance, the number of rounds [20]), the expected number of recovery steps can be computed in weak-stabilizing systems that have cycles, as the probability of leaving a cycle always converges to 1. In order to present the distributed system as a Markov chain, we consider a probabilistic scheduler, and a finite state space, i.e., weak-stabilization is as strong as probabilistic stabilization [41], and the probability of reaching correct behavior converges to 1.

Since the expected number of recovery steps does not reveal much about the structure of a weak-stabilizing algorithm, we also propose a graph-theoretic metric that characterizes the performance of a weak-stabilizing algorithm by identifying its strongly connected components and incorporating their size, density, centrality, and reachability. Our analysis and experiments on a weak-stabilizing leader election algorithm [41] in Chapter 3 clearly show that the performance of a weak-stabilizing algorithm heavily depends on its structure as well as the location and length of cycles.

In the remainder of the thesis, we may use WS to denote weak-stabilization, or weak-stabilizing.

1.5 State Encoding

Based on the insights we have gained by performance evaluation of self-stabilizing algorithms, we show that the performance can be of significant help in implementation, as we derive a general technique (called *state encoding*) that automatically (sometimes considerably) improves the performance of the implementation without changing the behavior of the protocol. Then we introduce three algorithms that automatically generate a state encoding scheme for a given self-stabilizing algorithm that can significantly reduce the expected number of recovery steps of the algorithm without changing its functional behavior. The state encoding algorithm works by changing the bit mapping function for the configuration of the system. The purpose behind this is to make the occurrence of the states from which the recovery is faster (the expected number of recovery steps is less than the overall expected value) more probable. Moreover, it reduces the probability of slower states occurring. In the context of weak-stabilization, we can also make the states that can reach cycles less probable to happen without changing the general behavior of the system. We suggest three algorithms for encoding which are fully described in Section 3.3.

The first algorithm works based on a feedback arc set for the directed graph, which is a set of arcs whose removal makes the graph acyclic. We assume we have the Markov chain of the model as the input to this algorithm, and we present it as a directed graph. Then, we obtain those arcs in the graph whose removal makes the graph acyclic. In the next step, we will try to reduce the probability of the sources of these arcs which are the endpoints from which the arcs start. The logic behind this algorithm is, if the sources of the cycles do not happen, the system never goes into a cycle.

The second algorithm makes the states with higher betweenness centrality (which is defined for a node in a graph, and counts how many times a node connects any other pair of nodes in the graph) less probable than other nodes. These two algorithms work exclusively for weak-stabilizing algorithms.

The third algorithm is a more general one and works for all types of self-stabilizing algorithms. This algorithm works based on the length of recovery paths for each state. Then, it tries to make those states whose length of recovery path is less than average more probable to occur, and consequently, make the rest of the states less probable to happen.

1.6 State Space Explosion Problem and Abstraction

Similar to traditional model checking methods, the probabilistic model checking suffers from the state space explosion problem: the number of the states in the system grows exponentially in terms of the number of components existing in the system. To tackle this problem, several solutions have been proposed. One of the solutions is bisimulation minimization [42] which suggests a model with the lowest possible number of states where this model has the same behavior and model checking results as the model before minimization. The states in the bisimilar model are gained through partitioning the state space of the model such that the states that exhibit similar behavior collapse in the same state in the bisimilar model. Another solution proposed for combating the state explosion problem is symmetry reduction [43] [44], in which the reduction of the problem to another problem that is solved with less memory usage is proposed. Then, verification is done on the reduced problem.

Other and more recent solutions contain abstraction methods [45]. In the abstraction methods, an abstraction function is defined that maps the states in the original model D to the abstract states in the abstract model $\hat{D}$. Several abstraction functions have been proposed, such as counter abstraction [46] [47]. In counter abstraction, the input is a distributed system D consisting of identical processes $1, 2, \ldots, n$ with domain $D = \{1, 2, \ldots, l\}$. The abstract system is defined with processes $k_1, k_2, \ldots, k_l$ with domain $D = \{1, 2, \ldots, n\}$ such that k_i equals to the number of processes that are at state i where $1 \leq i \leq l$. We give a fully detailed description of counter abstraction in Definition 31.

Another method is ordering abstraction [48], which sorts the states in ascending or descending order of the local states of processes. Predicate abstraction [49] is another method of abstraction that results in two-valued and three-valued abstraction models. In the two-valued logic model, the affirmative properties are conservative in the abstract model, meaning if the abstract model says true for a property verification, the property holds in the original model as well. However, when it is no, it is undecidable in the original model. The three-valued logic model is conservative for both affirmative and negative properties; however, when it is unknown, we cannot decide about the property in the original model. Even though there are several methods proposed for abstraction, there is still the lack of a decent abstraction method that works for an arbitrary number of states.

Here, we propose a novel abstraction function for verification of reachability properties where we aim to preserve the expected number of recovery steps as a measure of performance of the algorithm as we make the abstract system. Our method is described in detail in Chapter 3.

In general, the purpose of abstraction is to coalesce a set of states and put them in abstract states. Thus, the abstraction method works based on proposing a partition over the set of states in the system such that the abstract system is conservative in terms of verification of desired properties. It means the abstract system should provide a decent estimation for verification of quantitative properties.

Next, we propose a method for measuring the performance of an abstraction method. This method works based on two important factors that an abstraction method should have:

- How bisimilar is the abstract model to the system?
- How much memory does the abstraction method save us?

We investigate the answers to these questions in Chapter 3. We make use of these measurements as a tool for comparing abstraction methods. As a result, based on the insights we gained by evaluating the existing abstraction methods, we propose a new method of abstraction that is exclusively designed for verification of the expected number of recovery steps property in a probabilistic system. In our method, we collapse states with the same value of shortest path to a legal state in the same layer, which is defined as a set of states with the same number of shortest recovery path steps. Then, within each layer, we find the processes whose states happen with same probabilities, and after checking whether the same is confirmed by at least half of the layers we finalize the abstraction and put those processes in the same abstract variable. Thus, the variables in the abstract system build a partition over a set of components of the concrete system. Then, we execute some experiments on case studies and compare the results of our method with counter abstraction. This algorithm will be fully described in Section 3.4.

Organization The rest of the thesis is organized as follows. In Chapter 2, we explain the preliminary concepts such as graph-theory terms, the formal definition of a distributed system, probabilistic model checking, self- and weak-stabilization, abstraction, and the problems which are studied in this thesis. Our methods, case studies, implementation, and experimental results, consisting of the performance evaluation method for self-stabilizing algorithms based on the expected number of recovery steps, structure analysis of weak-stabilizing systems by graph-theoretical methods, and the proposed encoding and abstraction methods for improving the performance of the system, are described in detail in Chapter 3. Finally, we make concluding remarks in Chapter 4. Visual representation of some case studies are included in the appendix.

Chapter 2

Background

In this chapter, we explain technical definitions of the terms we use later in the thesis. This includes six sections: graph theory, distributed systems, probabilistic model checking, self-stabilization, state space abstraction, and self-stabilizing problems.

2.1 Graph Theory

Definition 1 (Undirected Graph) *A* undirected graph *G is denoted by $G = (V, E)$, consisting of a set V of vertices and a set E disjoint from V of edges. If two vertices u, v are endpoints of an edge $e = (p, q) \in E$, we call p, q* adjacent *or* neighbors. *An edge with identical end points is called a* loop *or* self-loop *[50]. A graph is* finite *if the sets of its vertices and its edges are finite.*

A graph with only one vertex is called *trivial*; otherwise it is *nontrivial*. A graph is called *simple* if it has no loop and no two edges with the same endpoints. A *digraph D* is denoted by $(V(D), E(D))$, where $V(D)$ is set of vertices and $E(D)$ is the set of arcs such that each arc is defined as an ordered pair of vertices of D. For instance $e = (u, v) \in E(D)$ is an arc that goes from u to v. We call u the *source* and v the *sink* of arc e.

We note that from now on, we exclusively talk about the directed graphs, and thus, all the definitions are considered for a directed graph unless we mention that it is undirected.

Definition 2 (Subgraph) *A graph H is called a* subgraph *of G, denoted by $H \subseteq G$, if and only if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. In this case, we call G a* supergraph *of H.*

The *degree* of a vertex v in an undirected graph is denoted by deg_v and is defined as the number of vertices that are incident to v. The notations δ_G and σ_G denote the minimum and the maximum degrees of the vertices of graph G, respectively.

Based on the topology the vertices in a graph constitute, different types of graph are defined. One of them is called the *ring graph*.

Definition 3 (Ring Graph) *The* ring graph *with n vertices has a vertex set corresponding to the integers 0 through $n-1$, and edges $(i, i+1)$, computed modulo n.*

Another type of graph is called *tree*.

Definition 4 (Tree) *An acyclic (without cycle) connected undirected graph is called a* tree*. The* height *of a node in tree is the length of the longest path to a leaf from that node. The height of the root is known as the* height *of the tree.*

A *walk* in directed graph G is a finite non-null sequence of vertices $W = v_1v_2v_3 \dots v_k$ such that for $1 \leq i < k$, v_i is adjacent to v_{i+1} by an arc that goes from v_i to v_{i+1}. The integer k is called the *length* of the walk W. If the arcs of a walk are distinct, we call it a *trail*. If the arcs and vertices of a walk are both distinct, we call it a *path*. The vertices u, v in graph G are said to be *strongly connected* if there is a path from u to v, and another path from v to u in G which are also called a $(u, v)-$*path*, and a $(v, u)-$path respectively. Strong connection is an equivalence relation on graph G. Thus, we can find a partition for set $V(G)$, $V_1, V_2, , \dots, V_k$ such that every pair of vertices u, v are strongly connected if and only if they belong to the same partition set V_i. The subgraphs $G(V_1), G(V_2), \dots, G(V_k)$ are called *strongly connected components*.

A walk with positive length and the same source and sink is called a *closed walk*. A closed trail which is a trail with the same source and sink is called a *cycle*. We call a graph *bipartite* if it has no odd cycle.

The *shortest path problem* is one of the most common applications of paths and cycles in graphs. Consider a graph in which a positive integer $w(e)$ is assigned to each of its arcs e. We call G with these weights on its arcs a *weighted directed graph*. These weights can be several concepts in practice, such as the cost of establishing a connection between two nodes. The shortest path problem is defined as follows: given a weighted graph, find the shortest path from a specified source to a specified sink in that graph.

The algorithm we will use was developed by Dijkstra (1959) [51] and independently by Whiting and Hillier (1960) [52]. For a given vertex u_0, this algorithm finds the shortest path between u_0 and all the other nodes in graph G.

In terms of connectivity, not all nodes may have the same significance. It means, for instance, that removing a node from a graph can change the set of its connected components.

Definition 5 (Betweenness Centrality) *The* betweenness centrality *[53] for a given node is the fraction of shortest paths that go through the node. More formally, it is computed by the following formula:*

$$C_B(s) = \sum_{s \neq u \neq v} \frac{SP_{uv}(s)}{SP_{uv}}$$

where SP_{uv} is the total number of shortest paths from node u to node v and $SP_{uv}(s)$ is the number of those paths that pass through s.

Definition 6 (Feedback Arc Set) *A* feedback arc set (FAS) *is defined for a digraph as a set of arcs whose removal makes a digraph acyclic.*

There are some approximation algorithms [54] for finding the minimum feedback arc set that we will use later in this thesis.

2.2 Distributed System

A *distributed system* is a collection of independent processes that communicate with each other through passing messages or shared memory [55]. In this thesis, we represent a distributed system by a self-loopless undirected graph $G = (V, E)$ in which V represents the processes of the distributed system, denoted by $p_1, p_2 \ldots, p_n$, and E represents the communications among processes. The numbers $1, 2, \ldots, n$ are known as the *indices* of the processes. The messages between processes pass through these edges. If $(p, q) \in E$ for two processes p and q, we call them *neighbors*. We assume that each process knows its neighbors by their indices. Hence, for each process p, we represent its neighbors as their indices. The number of neighbors of p denotes the maximum number of processes p can directly communicate with.

Communication between processes can be carried out through message passing or shared memory. Shared memory communication is used for those distributed systems in which the processes are spatially located close to each other, such as processes occurring within a multi-tasking single processor computer. Here, we use a shared memory model in which the processes communicate with each other by using shared variables. Each process owns a set of variables in which it can write and a possibly different set of variables from which it can read. These two sets can be different from each other. Each variable can take on a value within its domain. We assume that the domain of variables are finite.

Definition 7 (Global State and State of a Process) *The* state of a process p_i *is the values of its variables. The* state of the system*, also known as* global state*, is defined as the states of its processes.*

For each process, we can design a local algorithm that changes the state of that process. We can deterministically define the future state of a process by knowing its current state. We present this local algorithm by using Dijkstra's guarded command [1] which is defined as follows:

Definition 8 (Action) *An* action *is indicated by the following formula:*

$$\langle label \rangle \;::\; \langle guard \rangle \;\longrightarrow\; \langle statement \rangle$$

Here, label is the name of a local action, guard is a Boolean expression containing a subset of variables that can be read by the process, and the statement is the future state of the process.

An action is called *enabled* if its guard is true. A process is called enabled if at least one of its local actions is enabled. A *distributed algorithm* is defined as a set of actions of all of the processes. The speed of execution of local actions for the processes within a distributed system might be different, which may result in uncertainty for the global state of the whole system, when execution of the local actions of processes influence each other. This occurs when one process has more than one enabled actions which lead it to different states. Therefore, scheduling the actions made by the processes influences their future states. We use the *interleaving model* to deal with this complication. In this model, only one action can happen at a specific time, which is called an *atomic step* or a *computation step*. It can be a send or receive for a message passing system or a write or read for the shared memory one. We observe that a system allows its processes to execute their local algorithms simultaneously; however, there is no effect of one of these parallel actions on the others. Thus, enabled actions are allowed to be executed simultaneously if they have no effect on each other. From now on, we use may the term step instead of action.

$$\begin{array}{lcl}
x = 0 & \longrightarrow & \textbf{print}\,(\texttt{``safe''}) \\
x = 1 & \longrightarrow & x := 0 \\
x = 2 & \longrightarrow & x := 1 \\
x = 3 & \longrightarrow & x := 2 \\
x = 3 & \longrightarrow & x := 1 \\
x = 3 & \longrightarrow & x := 0
\end{array}$$

Figure 2.1: A set of guarded commands.

Definition 9 (Computation) *A* computation *or* execution *of a distributed protocol π is a maximal sequence of its states in which every two consecutive states are represented by an edge in the graph representing the distributed system. Maximality means either it is infinite or it reaches a state where no further action is enabled. The set of all executions on π starting from any initial state is denoted by Σ_π.*

Example. We consider this example for the sake of explaining the technical concepts. Assume a simple distributed system consisting of only one process. This process has the variable x which ranges over the domain $\{0, 1, 2, 3\}$. The actions are defined in Figure 2.1. If we assume $x = 3$ as the initial state, 3210 is a computation for this distributed system, then it prints out safe. This sequence is maximal since there are no further enabled actions. Moreover, $\Sigma_\pi = \{30, 310, 3210\}$.

2.2.1 Daemon or Scheduler

The asynchrony of computations is managed by a *daemon* or a *scheduler* [56]. A *daemon* is a predicate on the computations of distributed algorithms for making them possible. More formally, given a distributed system g and distributed algorithm π on it, a daemon d is a predicate that associates to each π a subset of computations of π.

The schedulers are classified based on different measures [56]:

- Fairness
  - An *unfair daemon* can schedule any set of enabled processes for execution. It is called unfair since there is a possibility that an enabled process is never scheduled for execution.
  - A *weak fair daemon* guarantees that an enabled process eventually will be scheduled for execution.

  - A daemon is *strongly fair* if any process that is enabled infinitely often is eventually scheduled for execution.
  - A daemon is *Gouda fair* if from any state that appears infinitely often in an execution, every possible transition is eventually executed.

- Boundedness
  A *k-bounded* daemon guarantees no process can be scheduled more than k times between the scheduling of any two other processes.

- Enabledness
  A *k-enabledness* process is a particular process that cannot be enabled more than k times before being executed.

Different types of schedulers may result in different results. In order to incorporate the impact of the scheduler on our results, we can generate a Markov chain and consider an order for actions based on different schedulers. Also, note that a daemon can be probabilistic in the way that it assigns a non-uniform probability distribution over the possible executions, e.g., considering higher priorities for those executions that take the system to recovery paths. In Section 3.1.5, we explain how non-uniform probability distribution can influence the performance of a self-stabilizing algorithm.

2.3 Probabilistic Model Checking

To reason about the general behavior of a distributed system, we use model checking methods [57]. Model checking focuses on a logical view of the state space and the set of transitions in the system. However, before going through the technical definition of model checking, we define formal methods.

Definition 10 (Formal Methods) Formal methods *are techniques of employing mathematics for the sake of reasoning about complex systems.*

The goal of formal methods is to prove the correctness of these systems formally by mathematical tools. Thus, they are considered as a significant tool for verifying and guaranteeing the correct behavior in these systems.

Definition 11 (Model Checking) *Given a certain property and a model of a system,* model checking *checks all the possible states in the system to see whether that property holds for the system.*

A general term in probability theory which will be used in this thesis is probability distribution. The probability distribution can be defined for both discrete and continuous variables. Since we are mainly concerned with discrete variables here, we just define the probability distribution for discrete variables.

Definition 12 (Probability Distribution for a Discrete Variable) *Let X be a variable that can take values $\{x_1, x_2, \ldots, x_n\}$. The probability that X can take a specific value x_i is denoted by p_i. We note that the probability is denoted by $\mathbb{P}$, thus for instance, the probability that $X = i$ is indicated by $\mathbb{P}[X = i]$. We call $\{p_i \mid 1 \leq i \leq n\}$ the* probability distribution *for X if it satisfies the following conditions:*

- *The probability that X takes value i is p_i, that is:*

$$\mathbb{P}[X = i] = p_i$$

- *p_i is non-negative for all $1 \leq i \leq n$*
- *The sum of p_i over all possible values of i is 1, that is:*

$$\sum_{i=1}^{i=n} p_i = 1$$

$\mathbb{P}_{\sim\lambda}[X = i]$ where $\sim \in \{<, \leq, >, \geq\}$ and $0 \leq \lambda \leq 1$ means the probability that $X = i$ happens holds with probability threshold λ. For instance, $\mathbb{P}_{\geq .8}[X = i]$ means the probability that $X = i$ occurs is greater than or equal to .8. Another term that will be used in this thesis regarding probability theory is the expected value.

Definition 13 (Expected Value) *Let X be a variable that can take values $\{x_1, x_2, \ldots, x_n\}$ with probabilities $\{p_1, p_2, \ldots, p_n\}$ respectively. Then, the* expected value *of X is defined as:*

$$\mathbb{E}[X] = \sum_{i=1}^{n} x_i p_i \tag{2.1}$$

Another probabilistic measure we will use is skewness of a graph which is measure of the symmetry of a graph.

Definition 14 (Skewness) *The skewness of a value is a measure of asymmetry which can be positive or negative. If we consider X as the random variable and μ to be its expected value, and σ to be its standard deviation, the skewness denoted by γ is calculated by the following formula:*

$$\gamma = \mathbb{E}[(\frac{X-\mu}{\sigma})^3]$$

Although model checking guarantees the correctness of a system, we know that in practice no system acts completely correctly. There are many possible events that influence the system behavior and can interfere with its normal behavior. Hence, verification of probabilistic systems (which are defined as systems that have probabilistic behavior, namely that the transitions within the system happen with some probability) is more common in practice. Even if our system is not probabilistic, we can consider equal probability for all the outgoing edges from each state and assume that the system is probabilistic, which means the methods proposed here are not restricted to only probabilistic systems.

An atomic proposition is a statement that can be either true or false and cannot be broken into smaller propositions. A *transition relation* is denoted by set of (s, act, s') where s, s' are two states in the state space, and *act* is an action in the set of the actions in the system which changes the state of the system from s to s'. A labeling function L assigns a subset of atomic propositions AP to each state $s \in S$.

Definition 15 (Transition System) *A* transition system *TS is a tuple $(S, Act, \rightarrow, I, AP, L)$ where*

- *S is the set of all possible states*
- *Act is the set of actions*
- *$\rightarrow \subseteq S \times Act \times S$ is a transition relation*
- *$I \subseteq S$ is the set of initial states*
- *Set of atomic propositions AP*
- *$L : S \rightarrow 2^{AP}$ is a labeling function*

A TS is called finite *if S, AP, and Act are finite.*

Let TS be a transition system. For $s, s' \in S$ and $\alpha \in Act$, $s \xrightarrow{\alpha} s'$ is another representation of action in which α indicates the guard, s is the present state, and s' is the future state or the statement of the action. The set of *direct α-successors* is defined as:

$$Post(s, \alpha) = \{s' \in S \mid s \xrightarrow{\alpha} s'\}, \quad Post(s) = \bigcup_{\alpha \in Act} Post(s, \alpha)$$

The *set of α-predecessors* is defined as

$$Pre(s, \alpha) = \{s' \in S \mid s' \xrightarrow{\alpha} s\}, \quad Pre(s) = \bigcup_{\alpha \in Act} Pre(s, \alpha)$$

Definition 16 (Discrete-time Markov Chain [58]) *A* discrete-time Markov chain (DTMC) *is a tuple $D = (S, S^0, \mathbb{P}, L)$, where*

- *S is the finite state space*
- *$S^0 \subseteq S$ is the set of initial states*
- *$\{p_s \mid s \in S\} : S \times S \to [0, 1]$ is the probability distribution over the set of out-going arcs from $s \in S$ such that for all $s \in S$, we have*

  $$\sum_{s' \in S} \mathbb{P}_s(s, s') = 1$$

  which means the sum of probabilities of all outgoing arcs from s is 1.
- *$L : S \to 2^{AP}$ is a labeling function*

Each discrete-time Markov chain can be considered as a transition system. S and the set of initial states can be considered equivalent as both Markov chain and transition system have a set of states and another set indicating their initial states. A subset of atomic propositions AP is assigned to each state in both systems. The transition relation $\to$ and set of actions Act in the transition system can be considered as the $\mathbb{P}$ function in the Markov chain. If all the enabled actions in the transition system happen with the same probability, their respective arcs in the Markov chain are assigned equal probabilities. Since each transition happens with a probability, we can model the system as a discrete-time Markov chain, which is a transition system, with a probability distribution over outgoing transitions from each state. From now on in this thesis, we use the term Markov chain to refer to "Discrete-time Markov chain".

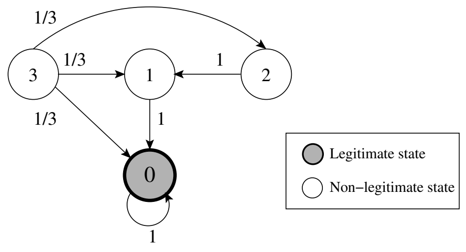


Figure 2.2: Markov chain of the guarded commands in Section 2.2.

We can easily see that a deterministic distributed system can be modeled as a Markov chain such that for two states s, s' between which there is an edge, we have $\mathbb{P}(s, s') \neq 0$. In particular, if a distributed algorithm is not probabilistic, then it can be modeled as a Markov chain, where the probabilities of all outgoing edges from each state are equal. Also, we assume that every arc in the system represents a single step. The step is the same as the action previously defined in Definition 8. Then if a process has t number of enabled actions, the probability of execution of each of its actions is $\frac{1}{t}$.

A Markov chain $D = (S, S^0, \mathbb{P}, L)$ can be represented as digraph $D' = (V(D), E(D))$ such that $V(D) = \{v_i \mid i \in S\}$, and each vertex is labeled by L in the same way as in the Markov chain. The arc $(s, s') \in E(D)$ for $s, s' \in S$ if and only if $\mathbb{P}(s, s') \neq 0$.

Example. The Markov chain of the example presented in Figure 2.1 is shown in Figure 2.2. Each state is labeled by the value of variable x. Each transition is annotated by the probability of its occurrence. In particular, the probability of outgoing transitions from states 0, 1, and 2 are 1, as these transitions are the only outgoing transitions from these states. All outgoing transitions from state 3 have probability $\frac{1}{3}$. The label of each state is written inside it. For instance, the label of not-legitimate states from left to right are: $x = 3$, $x = 1$, and $x = 2$. The label of the legitimate state is $x = 0$. The aim of representing a distributed system by a Markov chain is to perform both quantitative and qualitative verification of Markov chains [36] (quantitative verification is a technique for deciding about the quantitative properties of a system such as verifying whether the expected number of recovery steps is lower than a specific number, while qualitative verification is mainly concerned with verifying qualitative properties of a system such as verifying whether the system is in a legitimate state). We are mainly concerned with verifying the expected number of recovery steps here. The most appropriate logic for this purpose is probabilistic computation tree logic (PCTL) [59]. This logic is especially designed for reasoning about reliability.

Before defining the PCTL syntax, we define some terms for clarification:

- The property $\varphi = \varphi_1 \mathcal{U}^h \varphi_2$ means the property φ_1 must hold for h steps until φ_2 holds.

Definition 17 (PCTL Syntax) *Formulas in PCTL [59] are inductively defined (indicated by ::=) as follows:*

$$\varphi ::= p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \mathbb{P}_{\sim\lambda}(\varphi_1 \mathcal{U}^h \varphi_2)$$

where $p \in$ the set of atomic propositions AP, $\sim \in \{<, \leq, \geq, >\}$ is a comparison operator and $0 \leq \lambda \leq 1$ is the probability threshold. The sub-formula $\varphi_1 \mathcal{U}^h \varphi_2$ is the "until" operator [59].

Definition 18 (PCTL Semantics) *Let $\overline{\sigma} = s_0 s_1 \ldots$ be an infinite computation, i be a non-negative integer, and $\models$ denote the* satisfaction *relation. The semantics of PCTL is defined inductively as follows:*

$$\begin{array}{lll}
\overline{\sigma}, i \models true & & \\
\overline{\sigma}, i \models p & \textit{if and only if} & p \in L(s_i) \\
\overline{\sigma}, i \models \neg\varphi & \textit{if and only if} & \overline{\sigma}, i \not\models \varphi \\
\overline{\sigma}, i \models \varphi_1 \vee \varphi_2 & \textit{if and only if} & (\overline{\sigma}, i \models \varphi_1) \ \vee \ (\overline{\sigma}, i \models \varphi_2) \\
\overline{\sigma}, i \models \varphi_1 \mathcal{U}^h \varphi_2 & \textit{if and only if} & \exists k \geq i : (k - i = h) \wedge (\overline{\sigma}, k \models \varphi_2) \wedge \\
& \forall j : i \leq j < k : \overline{\sigma}, j \models \varphi_1 &
\end{array}$$

where $L(s_i)$ is the labeling function.
In addition, $\overline{\sigma} \models \varphi$ holds if and only if $\overline{\sigma}, 0 \models \varphi$ holds. Finally, for a state s we have $s \models \mathbb{P}_{\sim\lambda}\varphi$ if and only if the probability of taking a path from s that satisfies φ is $\sim \lambda$.

We use the usual abbreviation $\Diamond^i \varphi \equiv (true\, \mathcal{U}^i \varphi)$ for "eventually" formulas. The symbol i shows the upper bound on the number of steps until φ is achieved. Moreover, $\Box\varphi \equiv \neg\Diamond\neg\varphi$ is defined as the "globally" path formula which means φ always holds equals to saying it is not the case that eventually $\neg\varphi$ holds.

Example It is straightforward to see that the Markov chain in Figure 2.2 satisfies the following PCTL properties:

- $\mathbb{P}(x = 3) \Rightarrow \Diamond^3(x = 0)$
- $\mathbb{P}_{=1}\Diamond(x = 0)$, which has the same meaning as $\mathbb{P}_{=1}(1 \leq x \leq 3)\,\mathcal{U}\,(x = 0)$ (The value of h is arbitrary here)

Definition 19 *Let $D = (S, S^0, \mathbb{P}, L)$ be a Markov chain. A* state predicate *is a subset of S.*

Since each state in a Markov chain is labeled with a subset of atomic propositions, a state predicate can be considered as a subset of S (as the set of states) which are transformed to a PCTL formula by putting $\vee$ and $\neg$ around the atomic propositions.

2.4 Self-stabilization

A self-stabilizing system is the one that starting from any arbitrary initial state will eventually return to a set of states that are called "legitimate states". We denote this set by LS. The set of states outside LS is denoted by $\neg LS$. Every computation that occurs in a self-stabilizing system should contain a state in LS. Here we focus on two types of self-stabilizing methods. The first one is called strong self-stabilization, which guarantees that starting from any arbitrary initial state, the system will return to LS in a finite number of steps, whereas in the second one, called weak-stabilization, this finiteness is relaxed. In the following, we explain these in more detail.

2.4.1 Strong Self-stabilization

Intuitively, a (strong) self-stabilizing system is one that always reaches LS within a finite number of steps if its execution starts from any *arbitrary* state (called the *strong convergence* property), and after convergence it behaves normally unless its state is perturbed by transient faults (called the *closure* property). Since a self-stabilizing system can start executing from any arbitrary state, in the context of Markov chains we assume that $S = S^0$; i.e., an initial state can be any state in the state space. We formally define the notion of legitimate states using PCTL as follows.

The *closure property* ensures if the system is in a legitimate state, it remains there thereafter.

Definition 20 (Closure) *Given $D = (S, S^0, \mathbb{P}, L)$ as a Markov chain representing a distributed algorithm, and LS as the set of legitimate states, the* closure *property is defined as: For each state $s \in LS$, if there exists a state s' such that $\mathbb{P}(s, s') \neq 0$, then $s' \in LS$; i.e., execution of a transition in LS results in a state in LS.*

Definition 21 (Strong Convergence) *Given $D = (S, S^0, \mathbb{P}, L)$ as a Markov chain, representing a distributed algorithm, and LS as a state predicate presenting the set of legitimate states, the* strong convergence *predicate is defined as: starting from any arbitrary state, the probability of reaching the legitimate states is always 1, which implies D has no cycles in $\neg LS$. This condition ensures the finiteness of the recovery path in a strong self-stabilization.*

Definition 22 ((Strong) Self-stabilization) *Let $D = (S, S^0, \mathbb{P}, L)$ be a Markov chain, representing a distributed algorithm, and LS be a state predicate. We say D is* self-stabilizing *for* legitimate states *LS if and only if closure and convergence conditions hold for it.*

In Definition 21, the convergence property is also called *recovery*, and a computation that starts from a state in $\neg LS$ and reaches a state in LS is called a *recovery path*. Each arc in the recovery path is considered a *recovery step*.

Example It is straightforward to see that the Markov chain in Figure 2.2 is self-stabilizing for $LS \equiv (x = 0)$.

From now on, by self-stabilization we mean strong self-stabilization.

2.4.2 Weak Self-stabilization

Intuitively, a weak-stabilizing (WS) system is one in which for each execution starting from any *arbitrary* state, there exists an execution path that reaches the set of legitimate states, and after convergence it behaves normally unless its state is perturbed by transient faults. Since a WS system can start executing from any arbitrary state, in the context of Markov chains we assume that $S = S^0$; i.e., an initial state can be any state in the state space. Unlike the definition of strong self-stabilization, in weak-stabilization the existence of the

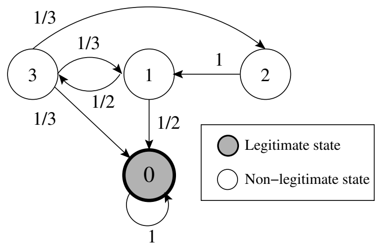


Figure 2.3: Markov chain of a weak self-stabilizing system.

recovery path suffices. Thus, a state in $\neg LS$ may reach cycles or connected components on its way. Following the result that a deterministic weak-stabilizing protocol can be turned into a probabilistic self-stabilizing one, we formally define the notion of legitimate states using PCTL as follows.

Definition 23 (Weak Convergence) *Given $D = (S, S^0, \mathbb{P}, L)$ as a Markov chain representing a distributed algorithm, and LS as a state predicate indicating the set of legitimate states, the* weak convergence property *says: starting from any arbitrary state, the probability of reaching the legitimate states converges to* 1*, which implies that the state may run into cycles in weak convergence.*

Definition 24 (Weak-stabilization) *Given $D = (S, S^0, \mathbb{P}, L)$ as a Markov chain representing a distributed algorithm, and LS as a state predicate denoting the set of legitimate states, we say that D is* weak-stabilizing *for set LS if and only if conditions closure and weak convergence hold for it.*

Similar to strong self-stabilization, in Definition 24, the convergence property is also called *recovery* and a computation that starts from a state in $\neg LS$ and reaches a state in LS is called a *recovery path*.

Example It is straightforward to see that the Markov chain in Figure 2.3 is weak-stabilizing for $LS \equiv (x = 0)$.

From now on, by weak-stabilization we mean weak self-stabilization.

Definition 25 (Snap-stabilizing Algorithm) *A* snap-stabilizing *algorithm ensures that starting from an arbitrary state in the system, it always is in LS. Thus, a snap-stabilizing algorithm is a self-stabilizing one that stabilizes in* 0 *steps.*

Definition 26 (Ideal-stabilizing Algorithm) *An* ideal-stabilizing *algorithm ensures that the system always is in its legitimate states. In other words, an algorithm is ideal-stabilizing if all of its states are legitimate ones.*

Ideal-stabilization is close to snap-stabilization except that in ideal-stabilizing, we are not restricted to any set of properties [60]. There are several model checkers for verification of a self-stabilizing algorithm. Since we focus on probabilistic model checking in this thesis, we use PRISM [34] as the model checker for implementing our methods.

PRISM [34] is a probabilistic model checker, which is used for model checking the Markov chain models as a probabilistic system. Given the Markov chain of the system D and a property as input, PRISM can verify whether D satisfies that property or not. In this thesis, we calculate the expected number of recovery steps using PRISM. First, we implement the self-stabilizing algorithm in PRISM. *Reward* is a feature in PRISM that can be defined by the user. We define *Reward* to be the number of steps. We consider all the possible states in the system as initial states. The algorithm executes to reach a legitimate state, and meanwhile, after each step, the *Reward* is updated as the number of steps executed so far. After the system stabilizes, the value of Reward is the number of recovery steps. We calculate the expected number of recovery steps for an initial state using the *Filter* feature in PRISM. Filter considers s as the initial state which is an arbitrary state in $\neg LS$ and LS as the set of all legitimate states. The symbol R denotes the number of recovery steps and avg indicates the expected number. We define our desired property which is the expected number of recovery steps here. We write it in the format of

$$filter(avg, R, LS, s)$$

We should note that studying the impact of place of occurrence and type of faults is a part of sensitivity analysis that is a crucial step in evaluating distributed systems in practice.

2.5 State Space Abstraction

A transition system (TS), as a model of a hardware or a software system, can model the behavior of the system in detail, which is a low-level implementation, or it can remove the details from the general model and keep just the significant and necessary information. This view is called the high-level modeling of the system.

We assume TS is an abstract transition system and TS' is a more detailed system model. TS' may have many and possibly an infinite number of states, whereas TS as an abstract model has a finite number of states such that TS preserves the desired property to be verified in TS'. Thus, we can model check TS instead of TS' for deciding about the satisfaction of the property. More formally, if $TS \models \alpha$ we may safely conclude that $TS' \models \alpha$.

Definition 27 (Bisimulation Equivalence) *Let $TS_i = (S_i, Act_i, \rightarrow_i, I_i, AP_i, L_i)$ be a transition system for $i = 1, 2$. A* bisimulation *for (TS_1, TS_2) is a binary relation $R \subseteq S_1 \times S_2$ such that:*

- *$\forall s_1 \in I_1 \exists s_2 \in I_2$ such that $(s_1, s_2) \in R$ and vice versa.*
- *for all $(s_1, s_2) \in R$:*
  1. *$L(s_1) = L(s_2)$ where $L(s)$ is the labeling function defined in Definition 16*
  2. *If $s'_1 \in Post(s_1)$ then there exists $s'_2 \in Post(s_2)$ with $(s'_1, s'_2) \in R$ and vice versa.*

  TS_1 and TS_2 are bisimulation equivalent *or* bisimilar*, which is indicated by $TS_1 \sim TS_2$ if there exists a bisimulation R for (TS_1, TS_2).*

It is straightforward that a Markov chain (MC) can be considered as the transition system of a distributed system. S is the state space set in both models. Each state in the MC has a subset of atomic propositions AP (which implies when the system is in that state, those atomic propositions are true), the sets Act and $\rightarrow$ can be considered to be equivalent to $\mathbb{P}$ set where the probability of all the transition relations are equal, since in a transition system, all the transitions happen with the same probability. L as the labeling function is also the same in both. Thus, the bisimulation equivalence relation can be easily generalized to MCs as well.

Definition 28 (PCTL Equivalence) *Let TS_1 and TS_2 be transition systems over AP.*

- *We define s_1 and s_2 to be $PCTL$* equivalent*, indicated by $s_1 \equiv_{PCTL} s_2$ if and only if:*

$$s_1 \models \phi \ if\ and\ only\ if\ \ s_2 \models \phi\ for\ all\ PCTL\ formulae\ over\ AP$$

- *TS_1 and TS_2 are PCTL equivalent denoted by $TS_1 \equiv_{PCTL} TS_2$ if and only if*

$$TS_1 \models \phi \; if \; and \; only \; if \; \; TS_2 \models \phi \; for \; all \; PCTL \; formulae \; over \; AP$$

It can be proved that the relation $R = \{(s_1, s_2) | s_1 \equiv_{PCTL} s_2\}$ is a bisimulation relation [2].

Definition 29 (Abstraction Function) *Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system and $\hat{S}$ be a set of abstract states. $f : S \rightarrow \hat{S}$ is an* abstraction function *if for any $s, s' \in S : f(s) = f(s')$ if and only if $L(s) = L(s')$.*

Definition 30 (Abstract Transition System) *Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system. The symbol $\hat{S}$ indicates a set of abstract states and $f : S \rightarrow \hat{S}$ is the abstract function. The* abstract transition system *$TS_f = (\hat{S}, Act_f, \rightarrow_f, I_f, AP_f, L_f)$ is defined as*

- *$\rightarrow_f$ is defined by the rule: $f(s) \xrightarrow{\alpha}_f f(s')$ if and only if $s \xrightarrow{\alpha} s'$*
- *$I_f = \{f(s) | s \in I\}$*
- *$L_f(f(s)) = L(s)$ for all the states $s \in S$*

Now, we are ready to discuss specific abstraction methods. One of them that will be used later in this thesis is counter abstraction, which is an abstraction method for distributed systems. We assume that every process P_i in the system executes the same set of actions with the domain variables $\{0, .., l\}$. The abstract state space $\hat{S}$ is defined by abstract variables $k_0, \ldots, k_l$ such that k_i is the number of processes that are currently in location $i, 0 \leq i \leq l$. In the case that the domain of processes is infinite, we use the $(0, 1, \infty)$ counter abstraction, which is less precise but still good. In this method, $k_i \in [0, 2]$ such that for $k < 2, k_i = k$ and otherwise $k_i = 2$.

Definition 31 (Counter Abstraction) *Consider a distributed system with a set of n processes P_i $1 \leq i \leq n$, each with a domain of variables $0, \ldots, L$. We define the counter abstraction function α with $\hat{S} = \{k_0, \ldots, k_L\}$ where k_i for $1 \leq i \leq L$ is a number counting processes that are in state i.*

Since in this thesis we are concerned about the finite distributed systems, the domain of variables should be finite.

2.6 Problems

In this section, we define some problems for which several self-stabilizing and weak-stabilizing algorithms have been suggested. The first problem is the propagation of information with feedback (PIF).

Definition 32 (PIF) *Given a distributed system D consisting of processes, the aim of* PIF *is to propagate a message request through processes, and get back a response. After getting the response, the processes get ready to propagate the next message.*

The second problem is known as token ring problem which is also the very first problem for which Dijkstra developed a self-stabilizing algorithm [1].

Definition 33 (Token Ring) *Given a distributed system consists of processes organized in a ring topology, and a property known as a token, the aim of* token ring *problem is to achieve a set of states in which only one process has the token.*

The third problem is the leader election problem which is among the attractive problems in self-stabilization.

Definition 34 (Leader Election) *Given a distributed system consisting of processes such that each process whether knows itself as the leader, or it knows one of its neighbors as its parent. The aim of* leader election *is to reach a set of states where the system has only one leader.*

Definition 35 (Mutual Exclusion) *Given a distributed system that consists of a number of processes, and a property known as* critical section*, the aim of* mutual exclusion *problem is to reach a set of states such that only one process is in the critical section.*

Chapter 3

Method, Implementation, and Experimental Results

This chapter includes five main parts. In the first part, we introduce a general method for performance evaluation of a self-stabilizing algorithm that is based on probabilistic model checking. This novel method removes the flaws in existing methods and is more precise. This is discussed more in the experimental results which are explained in Subsection 3.1.6. Also, we will discuss some of the benefits of this approach.

In the second part, we introduce a novel method for evaluation of a weak-stabilizing algorithm based on structural analysis of a Markov chain. We consider a Markov chain as a digraph and will use methods based on graph theory. This method is mainly aimed at investigating the topology of the underlying digraph of a distributed algorithm and the impact on its performance.

In the third part, we introduce and define encoding and propose three general algorithms for this purpose. The first algorithm works based on a feedback arc set. The structure of the second algorithm is based on betweenness centrality. These two algorithms work mainly for weak-stabilization. The third algorithm is a more general algorithm and works for every type of self-stabilization. We will show how we can improve the performance of a self-stabilizing algorithm by changing the encoding of some of its states. Then, we will support our claims with experimental results.

In the fourth part, first we propose an approach for evaluation of the performance of abstraction methods. The purpose of abstraction methods is to deal with the state explosion problem in a distributed system. We will address this problem, and after discussing its significance based on performance evaluation measurements, we propose an abstraction method on the state space of a distributed algorithm that is aimed to preserve the expected number of recovery steps. This abstraction method is exclusively designed for verification of reachability properties (which is mainly verification of expected number of recovery steps here) in distributed algorithms. Then, we will show how our algorithm works by running some experiments and comparing the results with existing methods of abstraction.

3.1 Performance Evaluation of Self-stabilizing Using Probabilistic Model Checking

As we discussed in Chapter 1, there are several shortcomings in the existing methods for performance evaluation of a self-stabilizing algorithm. Hence, we propose a new approach that works based on computing the average case performance in terms of the expected number of recovery steps for all the states outside LS.

3.1.1 Sketch of Our Approach

To obtain the expected number of recovery steps, we make use of probabilistic model checking methods to verify the expected number of recovery steps. More precisely, our proposed algorithm consists of the following steps:

1. For each state outside LS, we compute the probability of its reaching a state in LS.
2. For each state outside LS, we compute the expected number of recovery steps to a state in the set LS.
3. We calculate the expected number of recovery steps for all states outside LS.

We can clearly see that our result indicates how fast a self-stabilizing algorithm recovers. Furthermore, since we are using a model checking method and explore all the possible states, this method is a more fair measure for evaluation of an algorithm's performance than conventional methods. Hereafter, we will describe our method in a more formal and detailed manner.

3.1.2 Formal Description

Assume $D = (S, S^0, \mathbb{P}, L)$ is a Markov chain of a distributed system with LS as the set of legitimate states. Let $\mathfrak{s}$ be an arbitrary state in the set $\neg LS$. We should note that the symbol $\Rightarrow$ means implies, or it can be expressed as if the left part of the arrow is true then the right part of the arrow will be true. Consider the following PCTL property based on Definition 18,

$$\mathbb{P}_{\geq p}(\mathfrak{s} \Rightarrow \Diamond^h LS)$$

The property holds in D if starting from $\mathfrak{s}$ the probability of reaching LS within h steps is greater than or equal to p. For computing how fast the recovery is happening we should know the probability of the truthfulness of the following PCTL expression for each state in $\neg LS$.

$$(\mathfrak{s} \Rightarrow \Diamond^h LS)$$

which is the probability that the recovery is achieved for state $\mathfrak{s}$ within h steps.

We consider $\{\mu_{\overline{\sigma}} \mid \overline{\sigma}\ \textit{is a recovery path}\}$ to be the probability distribution on recovery paths (which is obtained by the multiplication of the probabilities of the path's arcs), and $R(\mathfrak{s})$ to represent the length of a recovery path that starts from $\mathfrak{s}$. Then, we assume $\overline{\sigma} = s_0 s_1 \dots$ to be a recovery path starting from s_0 and reaching LS within h steps ($\overline{\sigma} \models \Diamond^h LS$ in which $x \models y$ means x semantically entails y). Thus, $\mu(\overline{\sigma} = s_0 s_1 \dots)$ is the probability that path $\overline{\sigma}$ happens. We can calculate the probability of existence of recovery paths with length at most N from state $\mathfrak{s}$ by the following formula:

$$\mathbb{P}\{R(\mathfrak{s}) \leq N\} = \sum_{h=1}^{N} \{\mu(\overline{\sigma} = s_0 s_1 \dots) \mid (s_0 = \mathfrak{s}) \ \wedge \ (\overline{\sigma} \models \Diamond^h LS)\} \tag{3.1}$$

The above formula calculates the probability of recovery path of length at most N starting from s which is obtained by summation of the probabilities of all the recovery paths with length at most N.

Example. For the Markov chain in Figure 2.2, the probability of recovery of length at most two from state 3 to state 0 is $1/3 + 1/3 = 2/3$.

As a result, one can calculate the expected number of the recovery steps for all the recovery paths starting from $\mathfrak{s}$ by using the following formula.

$$\mathbb{E}[R(\mathfrak{s})] = \sum_{h=1}^{\infty} \{\mu(\overline{\sigma} = s_0 s_1 \dots) \times h \mid (s_0 = \mathfrak{s}) \ \wedge \ (\overline{\sigma} \models \Diamond^h LS)\} \tag{3.2}$$

Example. For the Markov chain shown in Figure 2.2, the expected number of recovery steps for state 3 is

$$1 \times \frac{1}{3} + 2 \times \frac{1}{3} + 3 \times \frac{1}{3} = 2$$

We note that different from conventional methods that consider the uniform probability distribution over states and assume all of states have the same probability of occurring, our approach works for all scenarios. Hence, we assign probability of occurrence $p_{(\mathfrak{s})}$ to each state $\mathfrak{s} \in \neg LS$. Since we are considering our initial state to be a fault, it should be in $\neg LS$. Therefore, the following formula holds:

$$\sum_{\mathfrak{s} \in \neg LS} p_{(\mathfrak{s})} = 1$$

The expected number of recovery steps for a self-stabilizing system D (where $R(D)$ denotes the number of recovery steps for it), denoted by $\mathbb{E}[R(D)]$, is calculated by the following formula:

$$\mathbb{E}[R(D)] = \sum_{\mathfrak{s} \in \neg LS} (\mathbb{E}[R(\mathfrak{s})] \times p(\mathfrak{s})) \tag{3.3}$$

Dissimilar to conventional methods, the above formula as performance evaluator takes into account several factors such as the probability of occurrence of each state and the number of states in $\neg LS$ from which the recovery is fast (or, respectively, is slow).

Example. For the Markov chain in Figure 2.2, assuming equal probability for all non-legitimate states, the expected mean value of recovery is

$$1 \times \frac{1}{3} + 2 \times \frac{1}{3} + 2 \times \frac{1}{3} = 1.66$$

We can easily see that if faults happen with a non-uniform probability distribution, which means the probability distribution over the states in $\neg LS$ is non-uniform, the expected number of recovery steps can significantly change. For instance, if the fault that reaches state 3 occurs with probability 90% and states 2 and 1 are each reached with probability 5%, then the expected number of recovery steps is

$$1 \times \frac{5}{100} + 2 \times \frac{5}{100} + 2 \times \frac{90}{100} = 1.95$$

To wrap up, the performance metric we introduced can be easily used for comparison of self-stabilizing algorithm by comparing the expected number of recovery steps in the Markov chains of the respective algorithms. The algorithm with the lower expected number recovery steps is expected to outperform the other one. This is because the algorithm with smaller expected number of recovery steps has faster recovery than the other algorithm. We will justify this claim in more detail through experiments in Section 3.1.4. More formally,

Definition 36 *Given $D_1 = (S_0, S_1^0, \mathbb{P}_1, L_1)$ and $D_2 = (S_2, S_2^0, \mathbb{P}_2, L_2)$ to be Markov chains of two self-stabilizing algorithms A_1 and A_2 executing on same network topology, we say that A_1* outperforms A_2 *if and only if:*

$$\mathbb{E}[R(D_1)] \ < \ \mathbb{E}[R(D_2)]$$

3.1.3 Some Benefits of Our Approach

In this subsection, we explain some of the benefits our approach provides. In addition to the average case analysis, we try to focus on its advantages in terms of some parameters that our approach considers.

The effect of the scheduler As explained in subsection 2.2.1, a scheduler is a predicate that determines the set of admissible executions in each state. To take into account the influence of a scheduler on performance analysis, we generate the Markov chain of a set of actions for a specific scheduler. For instance, if in a state several actions are enabled, then a distributed scheduler may potentially execute any enabled action. Such combinations can be captured in the resulting Markov chain in a straightforward fashion. We note that the scheduler itself can also be probabilistic.

The probability of a fault happening One of the most significant flaws in the asymptotic method for performance evaluation is that it assumes a uniform probability distribution over the state space. In practice, faults may happen with different probabilities, and this can have a major effect on the performance of a distributed algorithm. We note that Equation 3.3 addresses this shortcoming of asymptotic performance evaluation. It means that if faults that reach states with longer recovery steps are not probable, then the algorithm is more likely to perform much better than the worst case scenario.

Awareness of the effect of adversaries As discussed earlier, faults that perturb the system to states with recovery paths longer than the expected length are more severe faults. In other words, severe faults with more recovery steps will degrade the performance of the system, since it takes more steps for the system to recover to LS than for the case of an average fault. Thus, if an adversary attacks a system it can cause greater damage by taking the system to more severe faults. The benefit of our approach is since we are considering all the possible faults to happen, we know which faults are severe.

Blindness about the network topology and communication Another benefit of our approach is its generality in terms of topology and communication technology. The approach we present can be applied to any distributed algorithm with no concerns about its type of communication (be it shared memory or message passing), or network topology (be it a ring, a tree, or an arbitrary graph). Regardless of the presentation of an algorithm, we transform the algorithm into a Markov chain for verification. In this regard, it can be the case that an algorithm exhibits a different performance for similar topologies with small differences (e.g., a tree and a chain of processes). These details cannot be analyzed easily using abstract metrics such as asymptotic computational complexity in terms of rounds, since it does not take into account the behavior of every single state while evaluating the whole protocol. On the contrary, our method investigates the behavior of all states, which results in gaining a deep insight into the structure of the system.

Parametric analysis Although classic model checkers take a model as input, there have recently been advances on parametric verification, which is defined as verifying a system with an arbitrary number of processes. For instance, using parametric verification, one can analyze an algorithm for arbitrary size trees. Using such methods will allow us to analyze the performance of algorithms for more general classes of network topologies.

3.1.4 Implementation and Experimental Results

First, we present our case study on distributed PIF in rooted trees. Figure 3.1 shows a simple execution scenario of a PIF algorithm for three processes arranged on a line. Each process has three states: idle, request, and response (denoted $\{id, rq, rp\}$). In this scenario, initially all processes are idle (step 1). Then, the root makes a request (step 2). This request propagates all the way to the leaf, where a response is generated (steps 3 and 4). A process whose children are all responding also responds, except for the root,

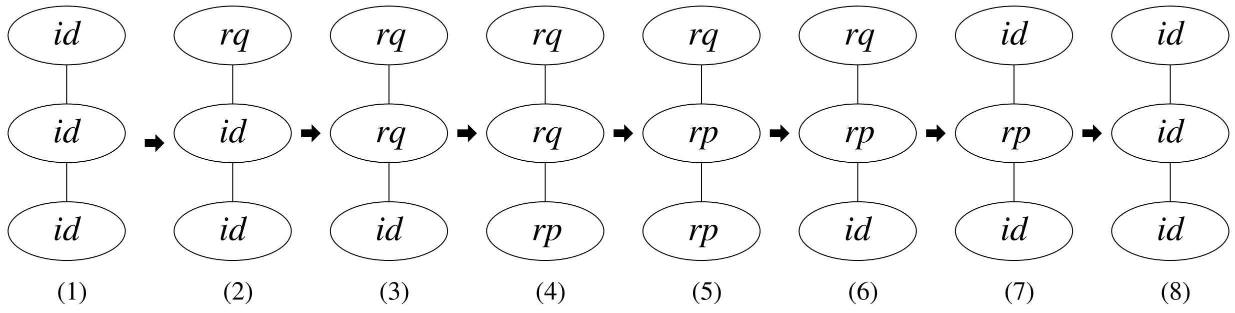


Figure 3.1: An execution scenario of PIF.

Algorithm 1 Snap-stabilizing algorithm for achieving PIF in rooted trees

```
Variable: $p_i \in \{id, rq, rp\}$ for internal processes
1: $p_i \in \{id, rq\}$ for the root processor
2: $p_i \in \{id, rp\}$ for leaf processors

Actions: {For Internal Processes}
3: Irq-action:: $S_p = id \wedge S_{A_p} = rq \wedge (\forall d \in D_p : S_d = id) \rightarrow S_p := rq$
4: Irp-action:: $S_p = rq \wedge (\forall d \in D_p : S_d = rp) \rightarrow S_p := rp$
5: Iid-action:: $S_p = rp \wedge (\forall q \in N_p : S_q \in \{rp, id\}) \rightarrow S_p := id$

Actions: {For Root Process}
6: Rrq-action:: $S_p = id \wedge (\forall q \in D_p : S_q = id) \rightarrow S_p := rq$
7: Rid-action:: $S_p = rq \wedge (\forall q \in D_p : S_q = rp) \rightarrow S_p := id$

Actions: {For Leaf Processes}
8: Lrp-action:: $S_p = id \wedge S_{A_p} = rq \rightarrow S_p := rp$
9: Lid-action:: $S_p = rp \wedge S_{A_p} \in \{rp, id\} \rightarrow S_p := id$
```

which becomes idle (steps 5 and 7). A responding process becomes idle when its parent is responding (steps 6 and 8).

We focus on three non-probabilistic self-stabilizing algorithms. These algorithms achieve PIF given any arbitrary state. The first algorithm [61], as shown in Algorithm 1, is snap-stabilizing with performance of $o(h^2)$ rounds, where h is the height of the rooted tree of the underlying network. In the sequel, we refer to this algorithm as Snap1. Observe that in the following algorithms, we indicate the ancestor of p by A_p, the descendants of p by D_p, and the neighbors of p by N_p. The variable of p is indicated by S_p, which has domain $\{id, rq, rp\}$. The topology we are considering here is line where we use *Linex* refer to x number of processes organized in a line structure together.

Algorithm 2 Snap-stabilizing algorithm for achieving the PIF in rooted trees

```
Variable: $p_i \in \{id, rq, rp\}$ for internal processes
 1: $p_i \in \{id, rq\}$ for the root processor
 2: $p_i \in \{id, rp\}$ for leaf processors

Actions: {For Internal Processes}
 3: Irq-action:: $S_p = id \wedge S_{A_p} = rq \wedge (\forall d \in D_p : S_d = id) \rightarrow S_p := rq$
 4: Irp-action:: $S_p = rq \wedge S_{A_p} = rq \wedge (\forall d \in D_p : S_d = rp) \rightarrow S_p := rp$
 5: Iid-action:: $S_p = rp \wedge (\forall q \in N_p : S_q \in \{rp, id\}) \rightarrow S_p := id$
 6: Iid correction-action:: $S_p = rq \wedge S_{A_p} \in \{rp, id\} \rightarrow S_p := id$

Actions: {For Root Process}
 7: Rrq-action::$S_p = id \wedge (\forall q \in N_p : S_q = id) \rightarrow S_p := rq$
 8: Rid-action::$S_p = rq \wedge (\forall q \in N_p : S_q = rp) \rightarrow S_p := id$

Actions: {For Leaf Processes}
 9: Lrp-action::$S_p = id \wedge S_{A_p} = rq \rightarrow S_p := rp$
10: Lid-action:: $S_p = rp \wedge S_{A_p} \in \{rp, id\} \rightarrow S_p := id$
```

In Algorithm 1, p_i is a process in the system that can be the root, an internal process, or the leaf. The topology for this algorithm can be any arbitrary tree; however, we assume the topology is a line here. Lines 1, 2 define the variables and their domains for each process. The next three lines define three actions for the internal processes. Line 3 defines the action which changes the state of an internal process to request. This happens when process is idle, the state of its ancestor is request, and the states of its descendants are idle. This line propagates the request of the message to all the way down to the leaf.

Next, Line 4 changes the state of an internal process to response. This happens when the state of the process is request and the descendant is in state response. This line takes the response message all the way up to the root.

Then, Line 5 makes the state of an internal process idle again. This happens when the state of the process is response, and the state of its neighbors is either response or idle. This line make the system ready again for propagation of a new message.

The next lines define actions for the root and leaf processes, similar to the explanations above.

The second algorithm [61], shown in Algorithm 2, is snap-stabilizing with performance of $o(1)$ rounds. This algorithm is obtained by modifying action *Irp-action* and adding *Iid correction-action* to Algorithm 1. In the sequel, we refer to this algorithm as Snap2.

This algorithm acts quite similar to Algorithm 1, except that it has an extra correction

Algorithm 3 Ideal-stabilizing algorithm for achieving the PIF network in rooted trees

```
Variable: $p_i \in \{id, rq, rp\}$ for internal processes
1: $p_i \in \{id, rq\}$ for the root processor
2: $p_i \in \{id, rp\}$ for leaf processors

Actions: {For Internal Processes}
3: Irq-action:: $S_p = id \wedge S_{A_p} = rq \wedge (\forall d \in D_p : S_d = id) \rightarrow S_p := rq$
4: Irp-action:: $S_p = rq \wedge S_{A_p} = rq \wedge (\forall d \in D_p : S_d = rp) \rightarrow S_p := rp$
5: Iid1-action:: $S_p \in \{rp, rq\} \wedge S_{A_p} = id \rightarrow S_p := id$

Actions: {For Root Process}
6: Rrq-action::$S_p = id \wedge (\forall q \in N_p : S_q = id) \rightarrow S_p := rq$
7: Rid-action::$S_p = rq \wedge (\forall q \in N_p : S_q = rp) \rightarrow S_p := id$

Actions: {For Leaf Processes}
8: Lrp-action::$S_p = id \wedge S_{A_p} = rq \rightarrow S_p := rp$
9: Lid1-action:: $S_p = rp \wedge S_{A_p} = i \rightarrow S_p := id$
```

line for making the state of an internal process idle again. Line 6 defines this action. This line says whenever the state of an internal process is request and the state of its ancestor is response or idle, the state of the internal process should be changed to idle. In the next section, we will see that this correction improves the performance of the algorithm.

The third algorithm [60], shown in Algorithm3, is an ideal-stabilizing algorithm with performance $o(h^2)$. We refer to this algorithm as Ideal.

The actions in Algorithm 3 are also similar to the previous ones, except for making the internal process idle again. Line 5 makes the state of an internal process idle when the state of the process is either request or response and the state of its ancestor is idle.

3.1.5 Implementation

The three aforementioned algorithms are modeled in the probabilistic model checker PRISM as discrete-time Markov chains. We use the *Reward* mechanism of PRISM to record and reason about the recovery steps from each state. Specifically, the reward "$\neg LS$: $steps + 1$", where *steps* is the number of recovery steps and initially $steps = 0$, specifies that in a computation, when a state in $\neg LS$ is reached, the recovery path has an additional step. Thus, the expected number of recovery steps can be computed using the value of *steps*. PRISM computes the expected value of *steps* only for a single initial state (i.e., Equation 3.2). However, using the *Filter* mechanism in PRISM, we can compute the expected value of

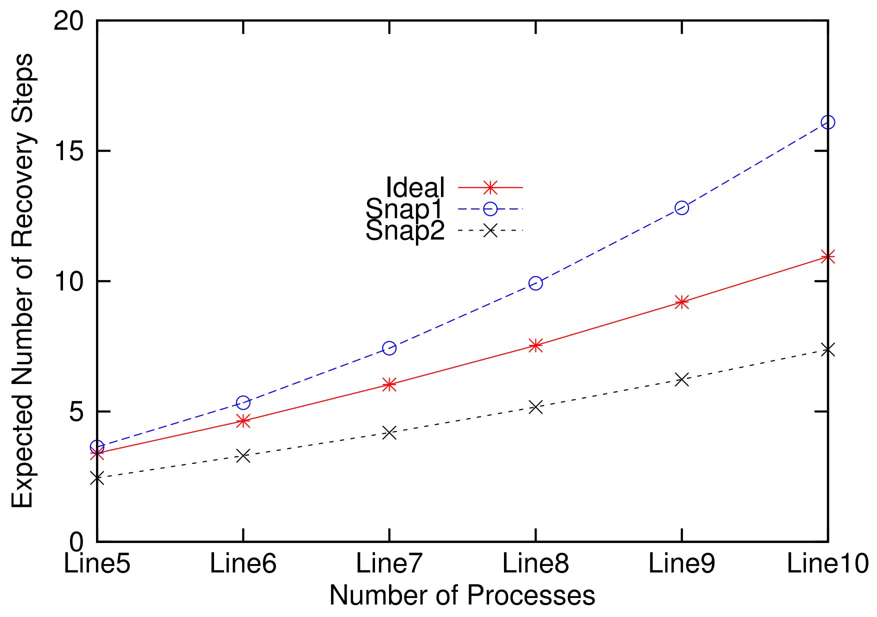


Figure 3.2: Expected number of recovery steps for three PIF algorithms.

steps for a set of initial states, namely, all states in $\neg LS$ (i.e., Equation 3.3). Likewise, using filters, one can compute the expected number of recovery steps from certain sets of states as well. Examples include sets of states reached by certain types of faults or faults that occur in certain processes.

3.1.6 Experimental Results

This subsection is organized in a top-down fashion. We first present our high-level results of comparing the performance of algorithms and then describe insights obtained by detailed analysis of the algorithms.

Expected number of recovery steps

Figure 3.2 shows the expected number of recovery steps of the three PIF algorithms for different numbers of processes. As can be seen, for the line topology of lengths 5–10, algorithm Snap2 exhibits faster recovery steps than Ideal and Snap1. However, algorithm Snap1's performance seems to diverge from the other two with a faster pace. This is because, in Snap1, as the number of processes increases the number of states from which recovery is slow grows more rapidly than the other two algorithms. Also, observe that although Snap1 and Ideal have identical complexity $\mathcal{O}(h^2)$, Ideal outperforms Snap1, and it does not diverge as fast as Snap1 as compared with Snap2.

Distribution of number of recovery steps

As mentioned above, the distribution of the number of recovery steps over states determines the overall recovery steps of an algorithm. To analyze the effect of this distribution, consider Figure 3.3, which shows the number of states in $\neg LS$ in terms of the number of recovery steps distribution of PIF for Line10 which is the line topology network consisting of 10 processes given to PIF algorithms as input. The skewness value of the graph can represent the performance of the algorithm. The positive value of skewness means the number of recovery steps is less than the expected value for more than half of the states (i.e. the mean is greater than the median). If the majority of states are concentrated on smaller numbers of recovery steps, then the expected number of recovery steps decreases. Moreover, a larger value of skewness implies better performance for the algorithm. That is, recovery for a larger number of states is faster. As we can observe, the skewness of Snap2 is towards the left, which causes the expected number of recovery steps to be less than that of the other algorithms. The same interpretation can be applied for comparing Ideal and Snap1. In particular, skewness values for Snap2, Ideal, and Snap1 are 2.26, 1.86, and 0.88, respectively. Hence, Snap2 has the best performance for Line10.

Impact of fault probability distribution

In practice, faults occur with a non-uniform probability distribution. As described in Equation 3.3, this probability distribution can significantly affect the performance of a self-stabilizing algorithm. We claim that since there are states from which algorithms Snap1 and Ideal show faster recovery than Snap2, if some faults reach a set of states with higher probability, then the expected number of recovery steps may change significantly. Figure 3.4 validates our claim (Region 1 refers to a state predicate in $\neg LS$, from which Snap1 performs better than Snap2). As can be seen in Figure 3.11(a) (respectively, Figure 3.11(b)), there are cases where Snap2 underperforms Snap1 (respectively, Ideal). This situation is clearer in the case of Ideal and Snap1 (see Figure 3.11(c)), i.e., there are more cases where Snap1 outperforms Ideal, although Ideal has better performance than Snap1 when the distribution is uniform (recall Figure 3.2). We observe that the topology for these experiments is Line8.

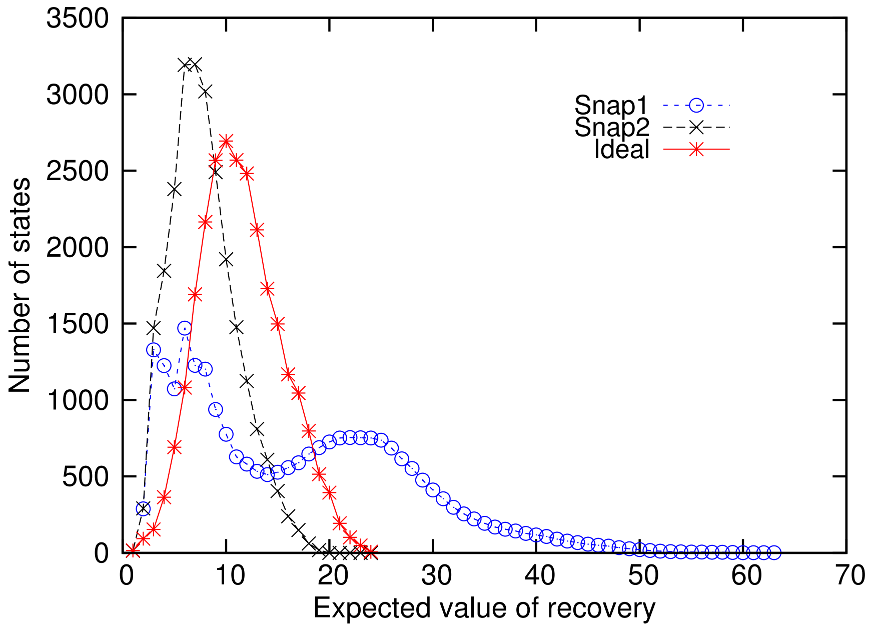
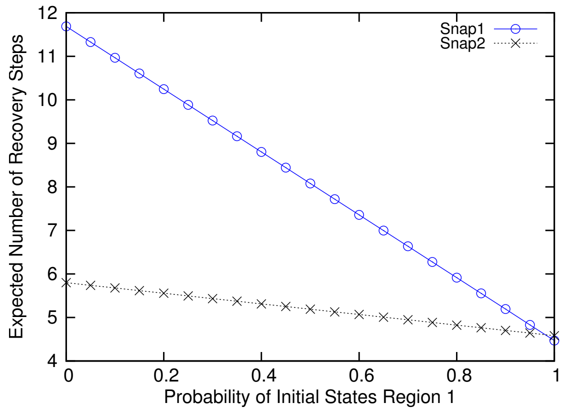


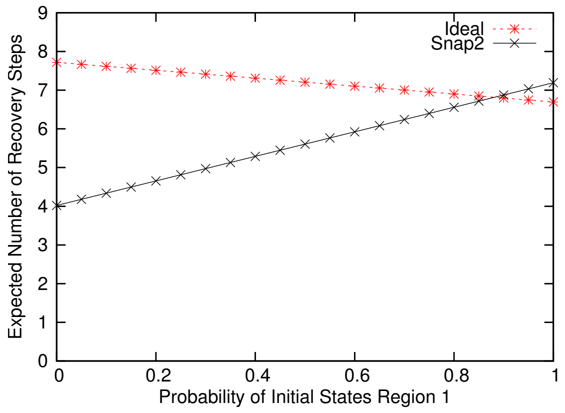
Figure 3.3: Recovery steps skewness of PIF for Line10.

Impact of type of and location of faults

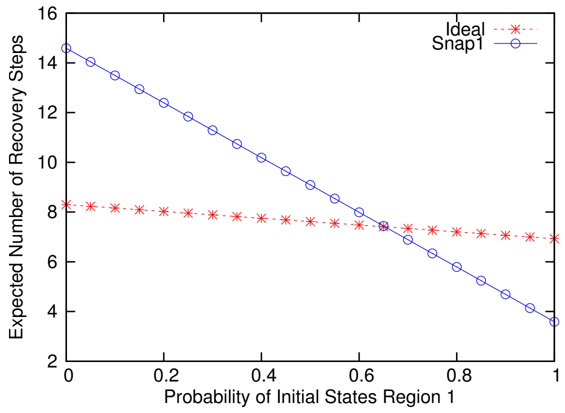
Figure 3.5 compares the overall expected number of recovery steps with the number of recovery steps when certain faults with respect to type and location occur, for all three algorithms for a line of size 12. For instance, if the state of the root process becomes rq in $\neg LS$, then in all three algorithms the recovery time from these states is less than the overall expected number of recovery steps of the algorithms. This is also the case when the state of an intermediate (i.e., non-root and non-leaf) process becomes rp. On the other hand, faults that change the state of intermediate processes to rq are more severe.



(a) Performance of Snap1 vs. Snap2



(b) Performance of Ideal vs. Snap2



(c) Performance of Ideal vs. Snap1

Figure 3.4: Impact of fault probability distribution (Line8).

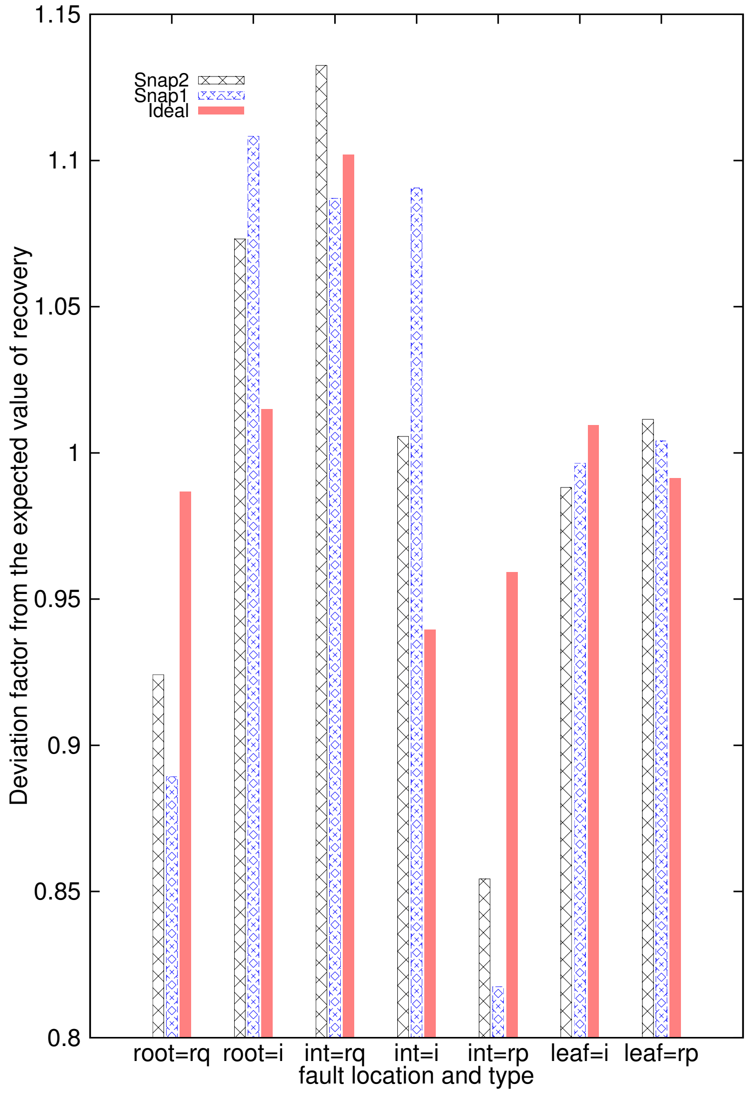


Figure 3.5: Comparison of expected number of recovery steps based on the place and type of faults (Line12).

Algorithm 4 Algorithm Dijkstra for token ring problem

```
Variable: $x_i \in \{0, 1, 2\}$ for each process $p_i$

Actions: {For process $p_0$}
 1: if $(x_0 + 1 = x_1)$ then
 2:     $x_0 := x_0 - 1$
 3: end if
Actions: {For process $p_i$, for $1 \leq i \leq n-2$}
 4: if $(x_i + 1 = x_{i-1})$ then
 5:     $x_i := x_{i-1}$
 6: end if
 7: if $(x_i + 1 = x_{i+1})$ then
 8:     $x_i := x_{i+1}$
 9: end if
Actions: {For process $p_{n-1}$}
10: if $(x_{n-2} = x_0)$ AND $(x_{n-2} + 1 \neq x_{n-1})$ then
11:     $x_{n-1} := x_{n-2} + 1$
12: end if
```

As another case study, we consider two self-stabilizing algorithms addressing the token ring problem. The first algorithm, proposed by Dijkstra [1], is indicated in Algorithm 4. We observe that in this algorithm, all the actions are considered in modulo 3.

Algorithm 4 suggests a self-stabilizing algorithm for the token ring problem on three-state processes organized in a ring. A process is said to have the token if its state exceeds the state of its last process by one. Since the topology is a ring, the previous process for p_0 is p_{n-1} and p_0 is the next process for p_{n-1}. The *top process* is known as p_{n-1}, and the *bottom process* is p_0. Line 1 first checks whether the state of p_1 exceeds the state of p_0 by one which means p_1 has the token. If this is true, then the state of p_0 will be reduced by one modulo three which means p_1 no longer has the token, and it passes the token either to its next or its last process. The next action for the middle processes says if p_i has the token, make its state as the state of its last process which means it no longer has the token, and passes it. The next action for the top process says if it does not have the token and the process of its left and right process are the same, then give it the token.

The second algorithm is described in Algorithm 5, which gives a modification on Algorithm 4 and provides a better lower bound [19].

Algorithm 5, known as algorithm BD, acts similarly to Dijkstra's with some modification on the actions of the algorithm. For instance, Line 2 adds an extra condition for changing the state of the process p_0. The state of the internal processes is also changed

Algorithm 5 Algorithm BD for token ring problem

Variable: $x_i \in \{0,1,2\}$ for each process p_i

```
1: Program for process $p_0$:
2: **if** $(x_0 + 1 = x_1)$ AND $(x_0 = x_{n-1})$ THEN **then**
3:     $x_0 := x_0 - 1$
4: **end if**
5: Program for process $p_i$, for $1 \leq i \leq n-2$:
6: **if** $(x_i + 1 = x_{i-1})$ AND $(x_i + 1 = x_{i+1})$ THEN **then**
7:     $x_i := x_i + 1$
8: **end if**
9: Program for process $p_{n-1}$:
10: **if** $(x_{n-1} = x_0 + 2)$ AND $(x_{n-1} \neq x_{n-2})$ THEN **then**
11:     $x_{n-1} := x_{n-1} + 1$
12: **else if** $(x_{n-1} = x_0)$ **then** THEN
13:     $x_{n-1} := x_{n-1} + 1$
14: **end if**
```

by adding an extra condition to be held. For the top process, the conditions have been modified.

The same measurements described above have been applied to these two algorithms as well.

Expected number of recovery steps Figure 3.6 shows the expected number of recovery steps of two token ring algorithms for different numbers of processes. As can be seen, the expected number of recovery steps for the BD algorithm is below Dijkstra's, as is claimed [19]. Also, the algorithms seem to have exponential growth as the size of the system increases.

Distribution of number of recovery steps To analyze the effect of this distribution, consider Figure 3.7 which indicates the number of states in $\neg LS$ in terms of the distribution of the number of recovery steps over the states in a token ring of size 10. As discussed earlier, the skewness value of a graph represents the performance of the algorithm. The skewness for the Dijkstra algorithm is 0.71 whereas it is 0.72 for the BD algorithm. We conclude that BD outperforms Dijkstra for this topology.

Impact of fault probability distribution As mentioned above, faults occur with a non-uniform probability distribution. Our claim is that since there exist states from which

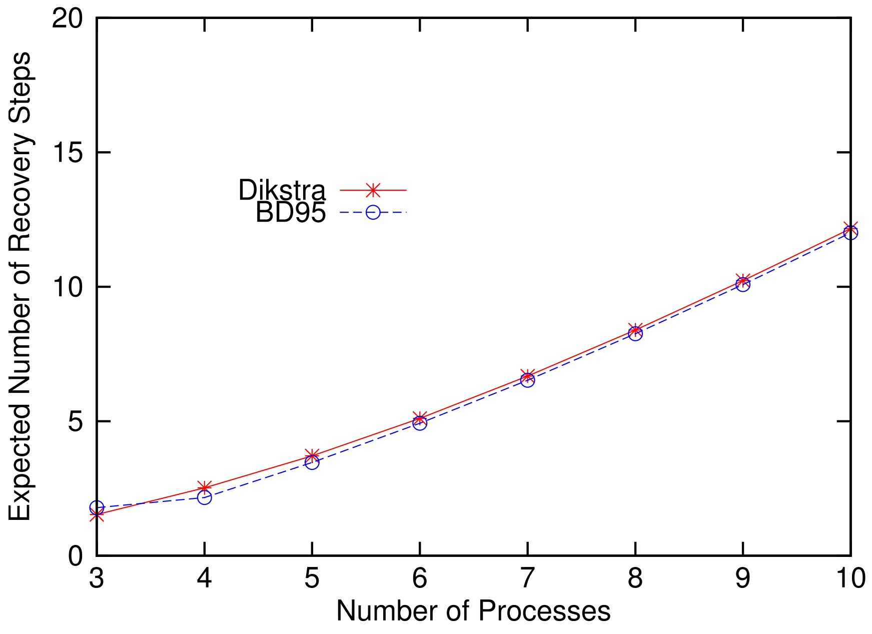


Figure 3.6: Expected number of recovery steps for two token ring algorithms.

Dijkstra's algorithm is faster than the BD algorithm, if some faults reach a set of states with higher probability, then the expected number of recovery steps may change significantly. Figure 3.8 validates our claim. As can be observed, in Figure 3.8 there are cases where the Dijkstra algorithm outperforms the BD algorithm, although in general, BD has better performance than Dijkstra.

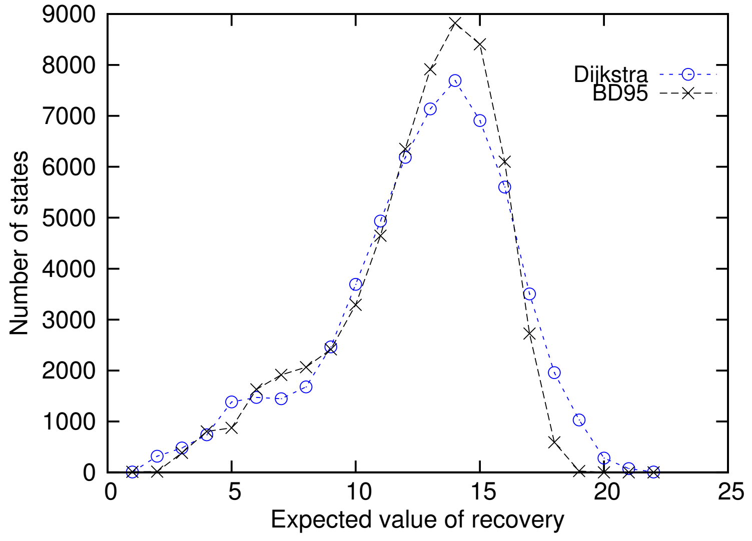


Figure 3.7: Recovery time skewness of token ring for tree size 10.

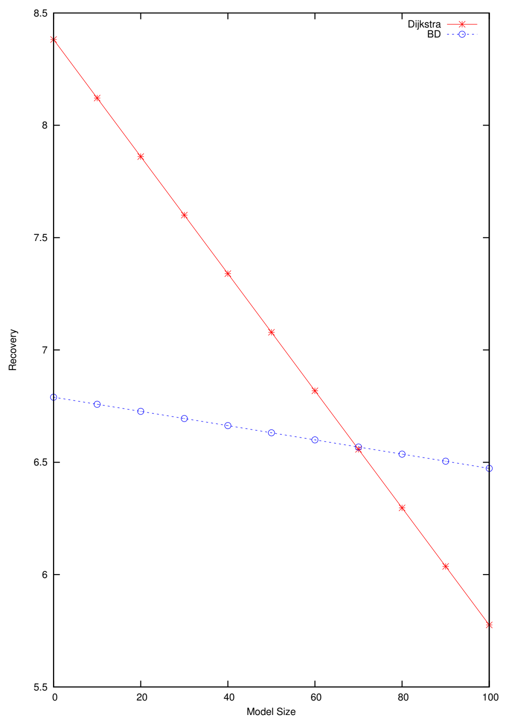


Figure 3.8: Performance of the BD algorithm vs. Dijkstra's algorithm.

3.2 Structural Analysis of Weak-Stabilization

In this section, we will describe our approach for evaluating the performance of a weak-stabilizing algorithm. This approach is exclusively designed for this set of self-stabilizing algorithms. The definition of weak-stabilizing algorithm ensures that starting from an arbitrary initial state, the probability of convergence to LS converges to 1, or in other words, there exists a path from the state to LS. It means that the number of recovery steps can be infinite in practice. This is because cycles and strongly connected components may be reached from the states. If the same probability for every outgoing execution from a state in $\neg LS$ is considered, it can be proved mathematically that the recovery path will be eventually selected [40]. However, in practice it can take a long time for the state to choose that path. In this case, the other paths may contain strongly connected components (or cycles) in the middle of them. As discussed in Chapter 1, by evaluating the performance of a weak-stabilizing algorithm we should analyze the structure of its cycles. Enumeration of all the cycles and computing their reachability together takes time exponential in the number of components that the system contains. Thus, we will focus on two graph-theoretic abstractions that can be approximate representatives of the existence of cycles and their connectivity.

Strongly Connected Components (SCCs) A SCC, as defined in Chapter 2, is a subgraph in which each two vertices are reachable from each other and which contains at least one cycle. We will consider the number of states and transitions in a SCC. We calculate the ratio of the number of transitions to the number of states. The higher the ratio, the more complicated the SCC. It is straightforward that the more complicated SCC can degrade the performance of a weak-stabilizing system. We will demonstrate this later in Section 3.2.1.

Betweenness Centrality (BC) As defined in Definition 5, the betweenness centrality of a vertex indicates how many times that node connects other nodes. We make use of this concept to obtain an idea about the centrality of the states in a Markov chain. The BC for the state s is denoted by $BC(s)$.

We compute the number of SCCs, and then for each SCC we calculate the BC for all the states in it. We developed a formula for evaluation of a weak-stabilizing algorithm based on SCC and BC of its Markov chain. We expect that as the number of SCCs in the Markov chain of an algorithm increases, it becomes more probable for the paths in it to get longer to recover, since they may get stuck in SCCs for a while. Next, one a path enters a SCC, the density of a SCC can determine how long it takes for the path to escape that SCC. The denser SCC makes the recovery path longer. Furthermore, again inside a SCC, the BC of the nodes determines how many times the recovery path may go through more nodes in a SCC. Thus, we expect it to have a direct impact on the number of recovery steps. To measure the BC of a SCC, we consider it to be the average of BC of all of its nodes. Then, we insert these values into the following formula, which is a characterization of the performance for a weak-stabilizing algorithm in Markov chain D:

$$\mathcal{R} = |D_{scc}| \cdot \sum_{scc \in D} (BC(scc) \cdot \frac{E_{scc}}{V_{scc}}) \tag{3.4}$$

where

- D_{scc} denotes the set of SCCs in D all of whose states are in $\neg LS$.
- $BC(scc)$ is the unweighted average BC of the states in scc.
- V_{scc} and E_{scc} denote the number of states (respectively, arcs) in scc.

The last fraction in Equation 3.4 indicates the *graph density* of a SCC. We observe that Equation 3.4 is an abstraction to explain the performance of a weak-stabilizing algorithm and may fail due to outliers. For instance, Equation 3.4 may fail to give an accurate measure of performance for the Markov chains with a small number of SCCs.

3.2.1 Implementation and Experimental Results

In this section, we describe our implementation for analyzing the performance of weak-stabilizing algorithms. Although our approach can be applied to any weak-stabilizing algorithm, we choose the weak-stabilizing leader election algorithm for anonymous trees [41] (by anonymous, we mean all the processes have the same set of variables and domain) for our study (see Algorithm 6). The state space of this algorithm includes cycles due to computations in which a process keeps changing its parent to its neighboring processes (i.e., in Line 2). Thus, the length of cycles in $\neg LS$ directly depends on the number of neighbors of processes. Δ_p is the degree of process p in the graph in this algorithm.

Algorithm 6 suggests a weak-stabilizing algorithm for solving the leader election problem. The algorithm defines the variable of the process p as Par_p, called the parent of p and its domain as $Neig_p \cup \{\perp\}$ which is the union of the neighbors of process p and the

Algorithm 6 WS leader election [41] for any process p

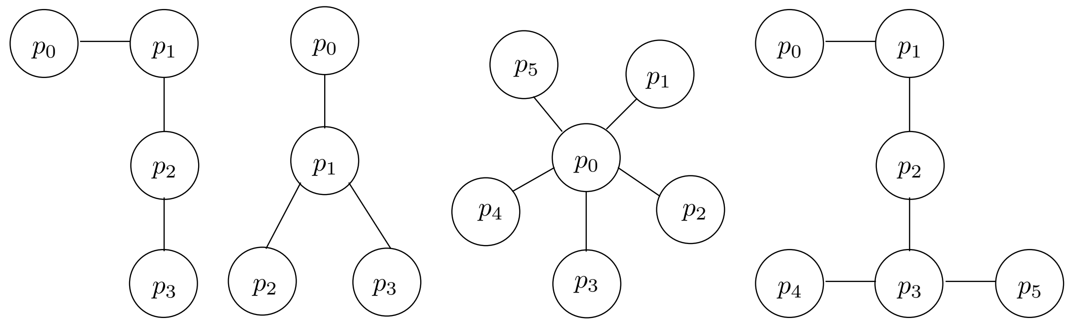
```
Variable: $Par_p \in Neig_p \cup \{\perp\}$
Macro: $Children_p = \{q \mid q \in Neig_p \wedge Par_q = p\}$
Predicate: $isLeader(p) \equiv (Par_p = \perp)$

Actions:
 1: $(Par_p \neq \perp) \wedge (|Children_p| = |Neig_p|)$ $\longrightarrow$ $Par_p := \perp$
 2: $(Par_p \neq \perp) \wedge [Neig_p \backslash (Children_p \cup \{Par_p\}) \neq \emptyset]$ $\longrightarrow$ $Par_p :=$ The next indexed neighbor of $p$
 3: $(Par_p = \perp) \wedge (|Children_p| < |Neig_p|)$ $\longrightarrow$ $Par_p := \min_{\prec p}(Neig_p \backslash Children_p)$
```

symbol $\perp$. Then, it defines the children of process p as q such that $Par_q = p$. If $Par_p = \perp$, process p is leader. The objective of the algorithm is to reach a state in which there is only one leader in the whole tree. Line 1 of the algorithm says if all the neighbors of p know p as their parent and process p is not leader, then we make it the leader by changing its state to $\perp$. Line 2 of the algorithm says if process p is not leader, and there exists some neighbors of p which does not know p as their parents, we switch the parent of p to its next neighbor modulo Δ_p. Finally Line 3 of the algorithm says if p knows itself as the leader and yet there are some neighbors of p which do not know p as their parent, we change the parent of process p to its minimum indexed neighbor.

Consider the network topologies shown in Figure 3.9 and their expected number of recovery steps obtained by the probabilistic model checker PRISM [34]. Although the expected number of recovery steps clearly show which topology converges faster when running Algorithm 6, we need more insightful information in order to analyze the behavior of the algorithm. We analyze the structure of the Markov chain of Algorithm 6 as a digraph running on each topology in Figure 3.9. We use the method of analysis of Section 3.2.

3.2.2 Analysis of WS Leader Election Algorithm

Figures 3.9(a) and 3.9(b) show the two possible topologies for a network of size 4. The Markov chain of the topology in Figure 3.9(a) (shown in Appendix A) includes three SCCs (Strongly Connected Components): one SCC with 3 nodes and six SCCs with 2 nodes. SCCs with 2 nodes are due to the fact that processes p_1 and p_2 can keep changing their leaders. For example, considering a uniform probability distribution for all enabled actions (probabilistic scheduler), in global state $(p_1, \perp, p_3, \perp)$ (i.e., $Par_{p_0} = p_1$, $Par_{p_1} = \perp$, $Par_{p_2} = p_3$, and $Par_{p_3} = \perp$) process p_2 has process p_1 as its neighbor, which is neither its parent nor its child. Hence, p_2 can execute Line 2 of Algorithm 6 and sets p_1 as its parent, where the global state becomes $(p_1, \perp, p_1, \perp)$. From this state, p_2 can again choose



(a) $\mathbb{E}\{R\} = 2.8$ (b) $\mathbb{E}\{R\} = 2.6$ (c) $\mathbb{E}\{R\} = 4.8$ (d) $\mathbb{E}\{R\} = 7.1$

Figure 3.9: Different topologies for Algorithm 6.

p_3 as its parent by executing the same action; hence, a cycle occurs.[1] There are $3 * 2 = 6$ combinations of such cycles. The SCC with 3 nodes is due to the following scenario. In state $s_0 = (\perp, p_0, p_3, \perp)$ since both processes p_1 and p_2 are not leaders, and they have neighbors which are not their children, by executing Line 2 of Algorithm 6, the system can reach either state $s_1 = (\perp, p_0, p_1, \perp)$ or state $s_2 = (\perp, p_2, p_3, \perp)$. Thus, there is a cycle of length 2 between s_0 and s_1 and another cycle of length 2 between s_0 and s_2; hence, there is a SCC of size 3 among s_0, s_1 and s_2.

In contrast, the Markov chain of the topology in Figure 3.9(b) includes only one SCC (due to process p_1 and its three neighbors) of size 18 states and 36 transitions. Since process p_1 has three neighbors and the other processes have only one neighbor, they all tend to choose p_1 as their parent, and thus, p_1 becomes the leader. This implies that this SCC looks more sparse than the SCCs for other topologies. Moreover, since there is only one SCC, there is no reachability of SCCs (cycle chains) together. These results make us expect a smaller number of recovery steps here.

We note that cycles with high graph density and chains of cycles tend to increase the expected number of recovery steps. This is because the existence of cycle chains and high-density cycles can make the recovery path more complicated, and there would be more paths that go through cycles and SCCs again and again. While the Markov chain of the distributed system with topology in Figure 3.9(b) has a long cycle, it does not have a high number of dense cycles and it does not have a chain of individual cycles. Unlike this case, in the topology in Figure 3.9(a), the SCCs form a relatively long chain of cycles, which causes a larger number of recovery steps. We note that because of the size of SCCs for four

[1]An alert reader can easily visualize the reachability graph of topologies of four processes, but we refer the reader to the Appendix for detailed visualization of the corresponding Markov chains.

processes, Equation 3.4 does not reflect the structure of its Markov chains properly. For these graphs, the number of SCCs and their connectivity can be calculated easily. In fact, we are more interested in the ability to analyze larger graphs and predict the recovery steps for larger systems. For six processes, we focus only on the best and worst case of topologies for the expected number of recovery steps. The star topology in Figure 3.9(c) exhibits the best performance among all six-process topologies as we can see the expected number of steps for it. Intuitively, this is because the network tends to quickly choose the center of the star (i.e., process p_0) as the leader. This intuition is reasonable, since the Markov chain of the star topology has only one SCC with low graph density 3.3, betweenness centrality 272, and $\mathcal{R} = 899$ (calculated by using Equation 3.4). The worst case recovery is for the topology shown in Figure 3.9(d). Our analysis shows that this topology creates 27 SCCs with dense cycles. In addition, some larger SCCs are highly dense with high betweenness centrality. For this topology $\mathcal{R} = 438,152$.

To summarize, the importance of the above discussion is threefold:

- Equation 3.3 provides us with fine-grained approach to evaluate the performance of weak-stabilizing algorithms.
- Equation 3.4 characterizes an approach to get a sense of the performance of a weak-stabilizing algorithm.
- These equations together are valuable tools for developers of distributed and network protocols to design more efficient weak-stabilizing algorithms.

3.3 State Encoding Employment for Improving Performance

Since we are investigating all the states in the system in our method of performance evaluation, we know the set of faults and can make the system more resilient to them. This can be done by changing the encoding for the states presenting severe faults and reducing the probability of their occurrence. *State encoding* is defined as assigning bit patterns to states. For example, two states s and s' can be assigned to bits `0` and `1`, respectively. Another possibility is to have bit patterns of size two and mapping `00` to s and `01`, `10`, and `11` to s'. In the first case, there is a bijection between bit patterns and abstract states. In the second case, the mapping is injective, but not surjective. The purpose of such state encoding is threefold:

1. To increase the proportion of LS as compared with $\neg LS$ states by making states appearing in $\neg LS$ less likely to appear (compared to the default choice of a bijective mapping) if bits can, for instance, randomly get flipped.

2. To decrease the expected number of recovery steps by making states that occur in the executions of long recovery paths less likely to appear than those belonging to short recovery paths' executions.

3. In the context of weak-stabilizing algorithms, our goal is to decrease the expected number of recovery steps by making states in $\neg LS$ which are in cycles less likely to appear. As a result, those states belonging to short recovery paths' executions will become more likely to occur.

Example. When the example shown in Figure 2.3 is considered, the expected value of recovery steps for each state can be calculated by using Equation 3.3. Since there is a cycle in the recovery path, the calculations become more complex. Thus, we implement this Markov chain in PRISM and calculate the expected number of recovery steps R from each state by using Filter. The results are:

$$\text{State 1 } \mathbb{E}[R] = 2.5, \text{ State 2 } \mathbb{E}[R] = 3.5, \text{ State 3 } \mathbb{E}[R] = 3.0$$

The expected mean value overall is

$$\mathbb{E}[R] = \frac{1}{3} \times 2.5 + \frac{1}{3} \times 3.5 + \frac{1}{3} \times 3.0 = 3.0$$

.

We apply an encoding for this example to improve the overall expected number of recovery steps for the system. The domain before encoding is $\{0, .., 3\}$. State 1 has the lowest expected number of recovery steps. Thus, since we want to make it more probable to occur, we map it to $\{1, 4, 5\}$. Then, we map state 3 which has the second smallest expected number of recovery steps to $\{3, 6\}$. Finally, we map state 2 with largest expected number of recovery steps to the same domain $\{2\}$. Thus, after applying the encoding, the new domain is $\{0, 1, \ldots, 6\}$. The probability of occurrence of the encoded state 1 is $\frac{1}{2}$, while it is respectively $\frac{1}{3}$ and $\frac{1}{6}$ for encoded states 3 and 2. Thus, the overall expected number of recovery steps will reduce to:

$$\mathbb{E}[R] = \frac{1}{2} \times 2.5 + \frac{1}{3} \times 3 + \frac{1}{6} \times 3.5 = 2.83$$

Algorithm 7 State encoding for SS algorithms (based on feedback arc set)

Input: Set of processes $\{p_0, \ldots, p_{n-1}\}$ and Markov chain $D = (S, S^0, \mathbb{P}, L)$ representing their respective distributed system

Output: Bit pattern state encoding

```
1: Let $G = (V, A)$ be the digraph, where $V = \{V_i \mid i \in S\}$ and for all $s, s' \in S$ $(v_s, v_{s'}) \in A$ if $\mathbb{P}(s, s') \neq 0$
2: Let $rank(v_s)$ be the length of the shortest path
   from vertex $v_s$, where $s \notin LS$, to a vertex $v_{s'}$, where $s' \in LS$
3: $A' := FAS(G)$
4: $V' := \{v \mid \exists u : (v, u) \in A'\}$
5: Let $U_1$ (respectively, $U_2$) be the set of states sorted by their $rank(v)$ values for each $v \in V'$
   (respectively, $rank(u)$ for each $u \notin V'$)
6: $U := U_1 \cdot U_2$ as the concat of $U_1$ and $U_2$ which is their ordered union
7: Divide $U$ into $n$ subsets $\{u_0 \ldots u_{n-1}\}$ based on the range of their ranks such that in each sublist we
   have equal range of ranks
8: for ($i = 0$ to $n$) do
9:     Encode($u_i$)
10: end for
```

.

Our claim in this section is that the insights gained through our performance analysis technique make it possible to design encodings that can improve the performance of an algorithm. We note that state encoding is only an implementation technique and does not change the behavior of the original algorithm. It is aimed to improve the performance of the algorithm. After applying the encoding, we implement the same set of actions for the new mapped domains. For instance process p_i with variable x with domain D in the algorithm will be changed to process p_i with variable x with new domain D' in which only the probability of occurrence is different.

We introduce three state encoding algorithms based on insights gained through our performance analysis.

3.3.1 Based on Feedback Arc Set

Algorithm 7 attempts to map more state bits to states that do not appear on a cycle. To this end, the algorithm utilizes the concept of a *feedback arc set* (FAS). The algorithm uses the sources of these arcs in Markov chain to generate a state bit mapping.

The algorithm takes n processes and a Markov chain D as input and generates a state bit mapping. First, it transforms D into a directed graph (Line 1). Then it ranks all states in $\neg LS$ based on the length of their shortest recovery path to a state in LS (Line 2). Next, it attempts to find an FAS (Line 3). Since any FAS will suffice, it does not matter that the problem of finding a minimum FAS is both NP-complete and APX-hard. Next, we sort the vertices that are sources of arcs in the FAS in the list U_1 and the rest of the vertices in the list U_2 (Line 6). The actual encoding (Line 9) occurs on n equal-ranged subsets of the concatenation of U_1 and U_2 which is obtained by putting elements of U_1 followed by elements of U_2 in order into the set U.

The function Encode works as follows. For each variable owned by process p_i, we define two domains for before and after encoding, denoted D_i and D'_i, respectively. D_i can be any finite subset of real numbers indicating local states of process p_i. Initially, D'_i is empty. For each u_i and value $j \in D_i$, we calculate the probability that process p_i takes value j. This is calculated by dividing the number of states in which p_i takes on value j by the number of all possible states. Then, we map each of the most probable states into $i+1$ states in D'_i (since we are encoding u_i and states are sorted decreasingly based on their expected number of recovery steps, as i increases in the for loop, the length of recovery becomes less, and we want to make it more probable to occur). Subsequently, we remove the original value from D_i. For instance, the states in u_0 consist of those with longest recovery paths. Thus, we map the most probable state in them to only one value in D'_i. In case that more than one process has the most probable state, we choose one process randomly. Since the states in $u_0 \cdots u_{n-1}$ are sorted in terms of their shortest recovery paths' length, we are guaranteed that for every process i each value in D_i will be mapped to at least 1 value in D'_i. Also, the most probable values in u_{n-1}, which includes the states with lowest length of shortest recovery paths' length, are mapped to n values in D'_i.

3.3.2 Based on Betweenness Centrality

Algorithm 8 is similar to Algorithm 7 except in choosing the set of sample states. In Algorithm 7 this set is obtained using a feedback arc set and in Algorithm 8, it is identified based on the betweenness centrality of strongly connected components (Lines 2 and 3). The rest of the algorithm works similarly to Algorithm 7.

3.3.3 Based on Expected Number of Recovery Steps

This algorithm works a bit differently from the two aforementioned algorithms. This algorithm works based on layers that are defined in terms of the length of shortest recovery paths. Each layer L_i consists of the states in $\neg LS$ with the same shortest path to a state in LS such that L_1 contains states with shortest recovery path of length 1, L_2 contains

Algorithm 8 State encoding for SS algorithms (based on BC)

Input: The set of processes $\{p_0 \dots p_{n-1}\}$ and its Markov chain $D = (S, S^0, \mathbb{P}, L)$
Output: Bit pattern state encoding

```
1: Let $G = (V, A)$ be the digraph, where $V = S$ and $(v_s, v_{s'}) \in A$ if $\mathbb{P}(s, s') \neq 0$ for all $s, s' \in S$
2: Let $SCC$ be the set of strongly connected components of $G$
3: Let $U$ be the sorted list of $C_B(scc) \cdot |scc|$ for each $scc \in SCC$
4: Divide $U$ into $n$ sub-sets $\{u_0 \dots u_{n-1}\}$ based on the range of $C_B(scc) \cdot |scc|$ value
5: for ($i = 1$ to $n$) do
6:     Encode($u_{i=1}$)
7: end for
```

states with shortest recovery path of length 2 and so on. Thus, the index of each layer indicates the length of recovery paths for its states. In the next step, we divide the layers into two subgroups: the first one contains those layers whose indices are less than or equal to the overall expected number of recovery steps, and the next one contains the layers with indices greater than the overall expected number of recovery steps. Then, for each group we do the following:

- Find the probability of each state happening for each process.
- For the first group: Sort the processes with their respective states in decreasing order of the above values and put them in S_1.
- For the second group: Sort the processes with their respective states in increasing order of the above values and put them in S_2.

Then we construct $S = S_1 \cdot S_2$ in which S is built by concatenation of S_1 and S_2. Starting from the beginning of S, we make the probabilities of the states of the corresponding process in increasing order such that the first state appearing in the list will become the least probable and as we continue the probability of the state will increase. Thus, the last element will be the most probable state (for instance, state 0 for process p_4). The logic behind this algorithm is by making the states with faster recovery more probable to happen, and making those with slower recovery less probable to occur, we can improve the overall performance of the system since after doing so, the expected number of recovery steps will be decreased.

Algorithm 9 State encoding for SS algorithms (based on layers)

```
Input: The set of processes $\{p_0 \ldots p_{n-1}\}$, each with domain $\{0, \ldots, l\}$ and Markov chain $D = (S, S^0, \mathbb{P}, L)$ representing the respective distributed system
Output: Bit pattern state encoding
1: Let $G = (V, A)$ be the digraph of MC, where $V = \{v_i \mid v_i \in S\}$ and $(v_s, v_{s'}) \in A$ if $\mathbb{P}(s, s') \neq 0$ for all $s, s' \in S$
2: Let $rank(v_s)$ be the length of the shortest path
   from vertex $v_s$, where $s \notin LS$, to a vertex $v_{s'}$, where $s' \in LS$
3: Put the nodes $v_s$ with the same shortest path in the same layer
4: Partition the layers into two groups: The layers with indices less than or equal to the expected number of recovery steps, and the layers with states having expected number of recovery steps greater than the overall expected value
5: For the first group, calculate the probability of each state for every process and sort the states and related processes in decreasing order of probabilities, namely $S_1$
6: Do the same for the second group with the difference that put the states and respective processes in increasing order of the probabilities, namely $S_2$
7: $S := S_1 \cdot S_2$
8: Encode elements of $S$ by assigning them probabilities in decreasing order such that the first element will become the most probable one and, respectively, the last element will be the least probable
```

3.3.4 Implementation and Experimental Results

The first section includes the experiments and results for strong self-stabilization, and the second part describes the experiments and results for weak-stabilizing algorithms.

Encoding Strong Self-stabilizing Algorithm

Consider a Line4 for the `Snap1` algorithm. We make the following observations (summarized in Table 3.1):

- In the case of the root process both states *id* or *rq* appear in $\neg LS$, but *id* appears 2.5 times more than *rq* in the root which is obtained by calculating the probabilities of each state in PRISM. Moreover, the expected number of recovery steps when the state of the root is *id* is 1.9, while the expected number of recovery steps when the state of the root is *rq* is 1.5. Thus, assigning more bit patterns to *rq* will result in a smaller-sized $\neg LS$ as well as decreasing the expected number of recovery steps. A 2-bit state mapping is the following: $\{\texttt{00}\} \mapsto id$ and $\{\texttt{01}, \texttt{10}, \texttt{11}\} \mapsto rq$. $\mathbb{E}'$ and $\mathbb{E}'$ respectively indicate the expected number of recovery steps before and after encoding. This encoding of the root will result in new expected number of recovery steps $\mathbb{E}' = 1.62$ for state id and $\mathbb{E}' = 1.4$ for state rq.

- The state of the second process can be id, rq, or rp, and all three appear in $\neg LS$. However, state rq appears in six global states, while the other two appear in four global states. Also, the expected number of recovery steps for states id and rp is 1.5, while the expected number of recovery steps for state rq is 2.2. Thus, reducing the occurrences of state rq should decrease the expected number of recovery steps. Two bits are necessary to encode three states, but since we want to make the state rq to be less probable to occur, we will not map two different bit patterns to state rq. Thus, we use a 3-bit encoding to decrease the proportion of non-legitimate states as follows: $\{\texttt{000}\} \mapsto rq$, $\{\texttt{001}, \texttt{010}, \texttt{011}\} \mapsto id$, and $\{\texttt{100}, \texttt{101}, \texttt{110}, \texttt{111}\} \mapsto rp$. Note that the choice for the state mapping of states id and rp is arbitrary.

- In the case of the third process, based on the expected number of recovery steps and number of states shown in Table 3.1, allocating more bit patterns to state id would be increasing the expected number of recovery steps since its expected number of recovery steps is more than overall expected value. In contrast, allocating more bit patterns to state rp decreases the expected number of recovery steps since its expected number of recovery steps is less than average. Based on these observations, we incorporate the state encoding shown in Table 3.1.

We observe that for some processes encoding makes recovery slower, but the technique is globally very effective. To clarify this, consider the expected number of recovery steps before and after encoding for each global state in Table 3.1. By taking into account that each global state may be replicated, the overall expected number of recovery steps for algorithm Snap1 before encoding is $\mathbb{E} = 2.36$ steps, while after encoding it is $\mathbb{E}' = 1.47$ steps. Figure 3.10 also shows the skewness of Snap1 graph before (0.81) and after (1.56) encoding.

Global states	$\mathbb{E}$	$\mathbb{E}'$	# of Encoded States
(id, id, rq, id)	2.0	2.66	3
(id, id, rq, rp)	1.5	1.33	9
(id, rq, id, id)	2.62	4.19	1
(id, rq, id, rp)	2.31	5.20	3
(id, rq, rq, id)	3.25	3.20	1
(id, rq, rq, rp)	2.5	2.20	3
(id, rq, rp, id)	1.0	1	6
(id, rq, rp, rp)	1.5	1.2	18
(id, rp, rq, id)	1.5	1.99	4
(id, rp, rq, rp)	1.0	1	12
(rq, id, rq, id)	1.5	1.99	9
(rq, id, rq, rp)	1.0	1	27
(rq, rp, rq, id)	2.0	2.36	12
(rq, rp, rq, rp)	1.5	1.14	36

Table 3.1: Impact of encoding on PIF expected number of recovery steps (Line4).

3.3.5 Encoding Weak-stabilizing Algorithm

Consider Algorithm 6 with four processes with the topology shown in Figure 3.9(a), where range $\{-1, 0, \ldots, 3\}$ represents $\{\perp, p_0, p_1, p_2, p_3\}$. Thus, considering the neighboring processes in the topology, we have $D_0 = \{-1, 1\}$, $D_1 = \{-1, 0, 2\}$, $D_2 = \{-1, 1, 3\}$, and $D_3 = \{-1, 2\}$. By applying Algorithm 7 we obtain the following subsets (see Figure A.1):

$$u_0 = \{(-1,0,-1,-1), (-1,0,1,-1), (-1,0,3,-1), (-1,2,-1,-1), (-1,2,3,-1), (1,-1,3,-1), (1,0,3,-1)\}$$

$$u_1 = \{(-1,-1,1,-1), (-1,-1,3,-1), (-1,0,-1,2), (-1,0,3,2), (-1,2,-1,2), (1,2,3,2)\}$$

$$u_2 = \{(-1,2,1,-1), (1,2,-1,-1), (1,2,1,-1), (1,0,1,2), (1,2,1,2)\}$$

$$u_3 = \{(-1,-1,-1,-1), (-1,-1,-1,2), (-1,-1,1,2), (-1,-1,3,2), (-1,2,1,2), (1,-1,-1,-1), (1,-1,-1,2), (1,-1,3,2), (1,0,-1,-1), (1,0,-1,2), (1,0,3,2)\}$$

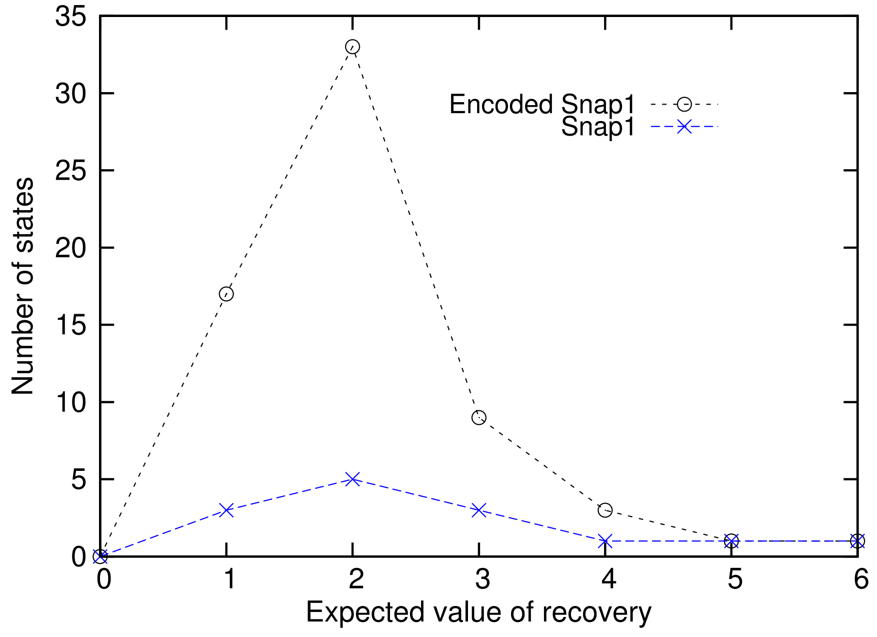


Figure 3.10: Skewness of Snap1 before and after encoding (line4).

Now, for u_0 we have the following (initially $D'_0 = D'_1 = D'_2 = D'_3 = \emptyset$):

$$\mathbb{P}(Par_{p_0} = -1) = \frac{5}{7} \qquad \mathbb{P}(Par_{p_0} = 1) = \frac{2}{7}$$
$$\mathbb{P}(Par_{p_1} = -1) = \frac{1}{7} \qquad \mathbb{P}(Par_{p_1} = 0) = \frac{4}{7} \qquad \mathbb{P}(p_1 = 2) = \frac{2}{7}$$
$$\mathbb{P}(Par_{p_2} = -1) = \frac{2}{7} \qquad \mathbb{P}(Par_{p_2} = 1) = \frac{1}{7} \qquad \mathbb{P}(Par_{p_2} = 3) = \frac{4}{7}$$
$$\mathbb{P}(Par_{p_3} = -1) = \frac{7}{7} \qquad \mathbb{P}(Par_{p_3} = 2) = \frac{0}{7}$$

The probabilities are simply calculated by counting. For instance the probability of $Par_{p_0} = -1$ is the number of states in u_0 in which $Par_{p_0} = -1$ divided by number of all states in u_0. The rest are calculated in the same approach. Thus, we map $-1 \in D_0$ to only one value in D'_0 since it is the most probable state in the longest recovery path. Thus, we update D_0 by removing -1 and D'_0 by adding -1. The application of this mapping to domain $D_1 \cdots D_3$ and $D'_1 \cdots D'_3$ is illustrated in Table 3.2. We observe that $\max \mathbb{P}$ indicates the state which is the most probable one in that iteration, and D and D' are the domain of the processes respectively before and after encoding.

	$i=1$			$i=2$			$i=3$			$i=4$		
	$\max \mathbb{P}$	D	D'	$\max \mathbb{P}$	D	D'	$\max \mathbb{P}$	D	D'	$\max \mathbb{P}$	D	D'
p_0	-1	$\{1\}$	$\{-1\}$		$\{1\}$	$\{-1\}$		$\{1\}$	$\{-1\}$	1	$\emptyset$	$\{-1,0,1,2\}$
p_1	0	$\{-1,2\}$	$\{0\}$		$\{-1,2\}$	$\{0\}$	2	$\{-1\}$	$\{0,2,5,6\}$	-1	$\emptyset$	$\{-1,0,1,2,3,4,5,6\}$
p_2	3	$\{-1,1\}$	$\{3\}$	-1	$\{1\}$	$\{-1,0,3\}$		$\{1\}$	$\{-1,0,3\}$	1	$\emptyset$	$\{-1,0,1,2,3,4\}$
p_3	-1	$\{2\}$	$\{-1\}$		$\{2\}$	$\{-1\}$		$\{2\}$	$\{-1\}$	2	$\emptyset$	$\{-1,0,1,2,3\}$

Table 3.2: Iterations of state encoding for $i = 0$ to 3 (Line 9 of Algorithm 7 for WS leader election and 4 processes for the topology shown in Figure 3.9(a)).

For u_1, the probabilities are the following:

$$\mathbb{P}(Par_{p_0} = 0) = \frac{0}{6} \qquad \mathbb{P}(Par_{p_1} = -1) = \frac{2}{6} \qquad \mathbb{P}(Par_{p_1} = 2) = \frac{2}{6}$$
$$\mathbb{P}(Par_{p_2} = 1) = \frac{1}{6} \qquad \mathbb{P}(Par_{p_2} = 3) = \frac{3}{6} \qquad \mathbb{P}(Par_{p_3} = 2) = \frac{4}{6}$$

For process p_0 and p_3 there is also only one value in the domain D remaining to be mapped to D'. Thus, we will decide on their mapping when processing u_2. For process p_2, we map 3 in D_2 to two values $\{3, 4\}$ in D'_2 and update D_2 by removing 3 from it. The remainder of this iteration is summarized in Table 3.2.

For u_2, we have

$$\mathbb{P}(Par_{p_0} = 1) = \frac{4}{5} \qquad \mathbb{P}(Par_{p_1} = -1) = \frac{0}{5} \qquad \mathbb{P}(Par_{p_1} = 2) = \frac{4}{5}$$
$$\mathbb{P}(Par_{p_2} = 1) = \frac{4}{5} \qquad \mathbb{P}(p_3 = 2) = \frac{2}{5}$$

Finally, for u_3, we have

$$\mathbb{P}(Par_{p_0} = 1) = \frac{6}{11} \qquad \mathbb{P}(Par_{p_1} = -1) = \frac{7}{11} \qquad \mathbb{P}(p_3 = 2) = \frac{8}{11}$$

The result of the iterations for u_2 and u_3 are summarized in Table 3.2. Using the above state encoding technique reduces the expected recovery steps of the weak-stabilizing leader election for the topology shown in Figure 3.9(a) (respectively, Figure 3.9(b)) to 1.6 (respectively, 1.7) from 2.8 (respectively, 2.6). We note that the encoded models use 26 times more space than the model before encoding. Thus, the problem is to find a trade-off between the expected number of recovery steps and memory usage.

As mentioned in Section 3.3, Algorithm 8 is similar to Algorithm 7 except in choosing the set of states to be encoded. Algorithm 8 works based on the betweenness centrality whereas Algorithm 7 is based on the shortest path from states in $\neg LS$ to LS. Applying Algorithm 8 reduces the expected recovery steps of WS leader election for the topology shown in Figure 3.9(a) (respectively, Figure 3.9(b)) to 1.5 (respectively, 2.1) from 2.8 (respectively, 2.6).

3.4 State Space Abstraction

An abstraction function, as defined in Definition 29, is intended to give a less detailed yet representative model of the system such that we can verify a desirable property in the abstract model instead of the original one. But how can we measure the performance of an abstraction method? In the first part of this section we propose a method for evaluation of an abstraction function.

3.4.1 Performance Evaluation of an Abstraction Function

In general, we can say that an abstraction function can be evaluated by two main points:

- How much memory does it save for us?

  One of the most significant motivations for designing an abstraction function is memory. We know that the size of the Markov chain (MC) grows exponentially in terms of the number of processes the distributed system has. Since model checking methods should investigate all the possible states in the system to decide about the verification of a given property, it takes an exponential amount of computation. Thus, it is not a good algorithm. For instance, let D be a distributed system consisting of n processes each with domain $\{0, \ldots l\}$. The number of the states in the Markov chain modeling this system is l^n. Using the counter abstraction method, the size of the abstract function will be calculated by using the following steps:

  - The counter abstraction has abstract variables $k_0, \ldots, k_l$ such that $0 \leq k_i \leq n$ for $0 \leq i \leq l$.
  - The number of possible states is the number of possible solutions for the following formula since each process should be in one state $0, 1, \ldots, l$ and there are n processes

$$k_0 + k_1 + \cdots + k_l = n \quad \text{such that} \quad 0 \leq k_i \leq n$$

The number of solutions for the above equation is $\begin{pmatrix} n+l-1 \\ l-1 \end{pmatrix}$, which is of order $\theta(n^l)$ [62].

Since in the systems we are considering usually $l < n$, as we know by the order of growth of functions, for large enough l, n is $n^l < l^n$. Thus, we save memory by using the counter abstraction method.

- How representative is the abstract system?

  We observe that as we remove more details of the system when the size of the D grows, the abstract system will become less representative and the verification will become less precise. Thus bisimilarity does not hold anymore. The counter abstraction method is one of the existing methods that suffer from this problem. The distance between the functions n^l and l^n grows as n and l get larger. Therefore, it would be good if we could stop somewhere before this distance gets too large and have a measurement of how much we can reduce the size of the system such that the error of the verification is negligible. For this purpose, we define a measurement, namely the *critical ratio*, which tells us how much memory saving is possible for a given system such that the difference between the expected number of recovery steps in the abstract model and the model before abstraction is less than 5%.

Definition 37 (Critical Ratio) *Let A be the smallest possible size of an abstract system in which the error in the expected number of recovery steps is less than or equal to* 5%*, and let S be the size of the original state space. The* critical ratio *is defined as*

$$CR = \frac{A}{S}$$

It is straightforward that different systems and abstraction algorithms may have various values of CR. Therefore, as an evaluation for an abstraction algorithm, the CR determines how helpful an abstraction algorithm can be for model checking a distributed system and algorithm.

Based on these two evaluation methods for an abstraction algorithm, we propose a new method of abstraction that always keeps the size of the abstract system within the critical ratio and yet saves us a noticeable amount of memory.

3.4.2 Our Method of Abstraction

The method we propose is designed to ensure verification of reachability properties. The basic idea is that processes whose states act similarly in terms of the number of recovery steps can be unified. However, how can we specify these processes? First, we calculate the expected length of the shortest recovery path for every state $s \in \neg LS$ by Dijkstra's algorithm. Then we take the floor function for these recovery steps and put the states with the same floor of shortest recovery steps in the same layer. Thus, we have Max(Lengths of Shortest Recovery Paths) $-$ Min(Lengths of Shortest Recovery Paths) $+ 1$ layers. Then, within each layer i, we calculate the probability of each possible state for each process and construct its *probability interval* which is defined as the interval between the lower and upper bound of its probability of occurrence. If the probability interval of two processes fall within each other's value with an difference of less than 5% for upper and lower bound of the intervals, we put those processes in the same group G. For instance if the probability interval for $p-1$ is $[.25, .5]$ and the probability interval for p_2 is $[.2, .45]$, we put p_1 and p_2 in the same group. At the end, if at least half of the layers construct the same group of processes, we coalesce them in the abstract system. Since this method ensures we have an error less than 5% in unifying processes, the size of the abstract system is at most equal to the value of CR. In the next step, we find the best way of mapping (an abstract function) for each group of processes. If a group contains a single process, we map its state to the same state (configuration) as that process. Otherwise, we decide on the mapped value based on the following:

Let S be a global state in which:

- $p_1, \ldots, p_g$ be the processes in the same group G,
- S_{p_i} indicate the state of the process p_i, and
- $count(S_{p_i})$ be the number of processes that currently are in state S_{p_i} in state S

.

For finding the best mapping value for group G in state S, we calculate the following for every process p_i where $1 \leq i \leq g$

$$val_{p_i} = \mathbb{P}(S_{p_i}) \times count(S_{p_i}) \tag{3.5}$$

The above formula works based on both the probability, and number of processes that are in state S_{p_i}. For instance if the state is $(0, 0, 0, 1)$ and the probability of occurrence of

Algorithm 10 State abstraction for SS algorithms (based on $E(R)$ for every state in $\neg LS$)

```
Input: The set of processes $\{p_0 \dots p_{n-1}\}$ and Markov chain $D = (S, S_0, \mathbb{P}, L)$ representing the respective
    distributed system
Output: Abstract Markov chain $\hat{D} = (\hat{S}, \hat{S_0}, \mathbb{P}, L)$
 1: Let $G = (V, A)$ be the digraph, where $V = \{v - i \mid i \in S\}$ and $(v_s, v_{s'}) \in A$ if $\mathbb{P}(s, s') \neq 0$ for all
    $s, s' \in S$
 2: Let $rank(v_s)$ be the expected length of recovery path
    from vertex $v_s$, where $s \notin LS$, to a vertex $v_{s'}$, where $s' \in LS$
 3: $L := \{L_i \mid L_i$ is a layer in $G\}$ such that layer $L_i$ consists of states with expected number of recovery
    steps $i$
 4: for ($j = 1$ to size of(Layers)) do
 5:     for (k=1 to size of(Layer $j$)) do
 6:         Calculate the lowest and upper bound of the probabilities of states of the processes
 7:         Construct probability interval for every $P_i$ in layer $j$
 8:     end for
 9:     Put the processes with similar probability intervals in the same group
10: end for
11: $C$ = Number of similar groups of processes in all layers
12: if ($C \geq sizeof(Layers)/2$) then
13:     Unify those processes in $\hat{D}$
14: end if
15: for ($j = 1$ to size of(groups)) do
16:     Calculate $val - i$ for each process $p_i$ in group $j$
17:     Extract $p - max$ and map group $j$ to process $p_{max}$
18: end for
```

each state for each process is .25, we map it to 0. However, when we have a state such as $(0, 0, 1, 1)$ where counting the state does not help to verify the state it should be mapped to, the probability determines how to map this state. For instance if the probability if 1 has the probability of occurrence .6 and 0 happens with probability .4, we map it to 1.

We map the state of the processes in G for state S to $S_{p_{max}}$ where $S_{p_{max}}$ is the state of a process that has the maximum value of val_{p_i}. The remaining processes will hold their original value for state.

Algorithm 10 takes the Markov chain of a distributed system as input and gives the abstract Markov chain as the output. Line 2 defines the rank for each vertex such that $rank(v)$ is the expected length of the recovery path from that vertex. Then, Line 3 defines layers as states with the same rank. For instance, L_i contains all the vertices whose rank is i. Then for each layer, Line 6 constructs the Max and Min of the probabilities (respectively p_{uj} and p_{lj}) of each state for all the processes in that layer. Then we construct the probability interval for that process as $[p_{lj}, p_{uj}]$. Then in layer j, we put the processes with the same probability intervals (with negligible error of 5%) in the same temporary groups. Then in Line 11 we count the similar temporary groups, and if the number of similar groups is larger than half of the number of layers, we put the processes in one final group. Then, in Line 17, we map the state of final grouped processes to the appropriate value based on val_i defined in Equation 3.5.

3.4.3 Implementation and Experimental Results

In this section the experiments for the abstraction have been executed for the PIF network. We will explain the implementation and description of each in separate sections.

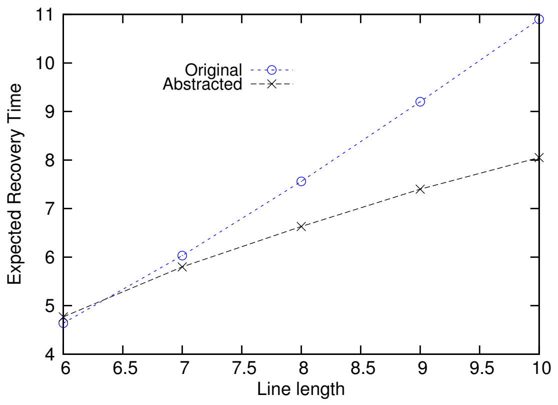
Implementation and Experiments on PIF Tree Networks

Implementation As described in Section 3.1.4, we implement the PIF protocols for several sizes of model in PRISM and then calculated the expected length of the recovery path by a step-based reward. We implemented both counter abstraction and our method of abstraction described in Algorithm 10 in the package igraph for Python and Scilab.

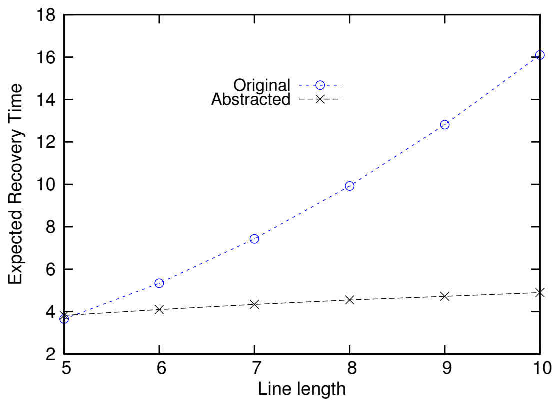
Experiments First, we calculate the critical ratio for counter abstraction for algorithms Snap1, Snap2, and Ideal. Figure 3.11 shows visualization of the CR. In Figure 3.11(a), we see how the expected recovery time (expected number of recovery steps) grows as the size of the system increases for the abstract and the original system (the system before abstraction). We can see that the difference between the abstract model and the original one increases as the system gets larger. When the difference between these two gets larger than 5%, the abstract model is no longer representative of the original system. Figure 3.11(b), and Figure 3.11(c) have the same interpretation.

Then, we implemented the counter abstraction method for the three algorithms Ideal, Snap1, and Snap2 and calculated the critical ratio as defined in Definition 37. Then, for different sizes of PIF models, we conducted experiments for both counter and our method of abstraction. The results are shown in Table 3.3.

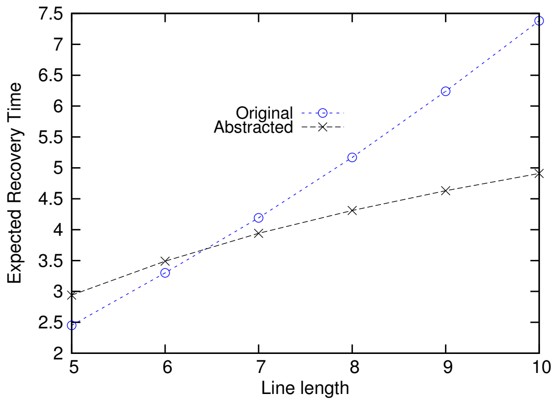
As we can observe in Table 3.3, as the size of the model increases, our method has much less error than counter abstraction, since its prediction value of the expected recovery steps is relatively closer to the original value. On average, for the experiments we have done the average error of our method is 4% whereas it is 16% for the counter abstraction method.



(a) Critical ratio for Ideal = 0.07



(b) Critical ratio for Snap1 = 0.4



(c) Critical ratio for Snap2 = 0.09

Figure 3.11: Critical ratio for Id, Snap1, Snap2 on PIF network.

Model Size	Original value of $E(R)$	Counter Abstraction	Our Method
Line7	6.03	5.8	5.8
Line8	7.56	6.63	7.59
Line9	9.25	7.4	9.6
Line10	10.9	8	9.9

Table 3.3: Experimental results for counter abstraction and our method of abstraction in different sizes of PIF tree networks.

Chapter 4

Conclusion and Future Work

In this thesis, we focused on performance analysis of distributed self-stabilizing and weak-stabilizing algorithms using probabilistic model checking techniques. We argued that the traditional methods of evaluation such as asymptotic computational complexity measurements abstract away many factors that are practically significant and are worthy of notice. Furthermore, the conventional methods cannot take into account real constraints such as the likelihood of occurrence of faults. We proposed a new metric, namely, the expected number of recovery steps to complete stabilization. This metric can be measured straightforwardly by using off-the-shelf probabilistic model checkers such as PRISM, which we used here. We supported our claims by employing experiments on three self-stabilizing algorithms for distributed propagation of information with feedback (PIF) in rooted trees. Another case study we considered is two self-stabilizing protocols proposed by the Dijkstra [1] and BD [19] algorithms. Moreover, the insights gained by our experiments led us to more efficient implementations using state encoding.

As for future work, there are several new research directions. For instance, one can design parametric model checking techniques to analyze the performance of algorithms when the exact number of processes is unknown. Yet another challenging problem is to devise probabilistic synthesis techniques that can revise an existing protocol to improve its performance, providing an automatic method which can improve the algorithm performance regardless of the number of processes it contains, or the problem it solves.

Next, we argued that evaluation of the performance of weak-stabilizing (WS) algorithms has not been studied and reported in the literature because such algorithms may exhibit behavior that never stabilizes. For this reason conventional metrics for evaluating the performance of self-stabilization (e.g., asymptotic computation complexity) become irrelevant in the context of WS algorithms. We proposed an automated method for evaluating the performance of distributed WS algorithms using a probabilistic model checking verification method. Our method computes the expected number of recovery steps of each state of a WS algorithm using rigorous state exploration from which the overall expected number of recovery steps of the algorithm can be obtained. To our knowledge, this is the first attempt to study the performance of WS systems. Since the expected number of recovery steps as just a value is not particularly insightful, we also proposed a graph-theoretic method to analyze the structure of the underlying graph of WS algorithms. This method is based on identifying strongly connected components and their betweenness centrality in the reachability graph of a WS algorithm. All our claims were supported by experiments on a WS leader election algorithm [41]. Using our insights, we also introduced two algorithms that generate state bit maps (called *state encoding*) that improve the expected number recovery steps in an implementation of a WS system. These algorithms are automated; however, they can be applied only to weak-stabilizing protocols as they are working based on the structure of the strongly connected components that exist in the structure of the underlying graph. Then, based on all encoding results, we proposed an automated encoding algorithm that works for all types of self-stabilizing algorithms. This algorithm works based on the length of the shortest path of the states to the set of legitimate states.

As for future work for this part, we are planning to design algorithms that design WS systems with given the expected number of recovery steps as the objective. Another interesting direction is to design parametric model checking techniques to analyze the performance of WS algorithms when the exact number of processes is not given.

Finally, we discussed the state explosion problem that exists in all types of model checking, including probabilistic model checking which was our main concern here. We investigated using existing methods for solving this problem. There are several methods ranging from bisimulation that aim to minimize the size of the system while being representative as well as abstraction methods. We argued that there is a lack of a sufficient measure to compare these methods. Thus, we suggested a measure called the critical ratio for evaluating and comparing abstraction methods. The critical ratio measures how much memory we can save such that our abstract model is still a good representative of the original system. Based on this measurement, we proposed a novel technique of abstraction. Our method works based on the shortest recovery paths and coalescing the states with the same length of shortest recovery path in the same layer. Then, within each layer we unify the processes with similar behavior. Finally, if more than half of the layers unify the same set of processes, we construct the groups and construct the abstract system. Thus, our method gives a partition over the processes of the distributed system. This abstraction method is exclusively designed for the verification of quantative reachability properties. We employed our method of evaluation and abstraction for three self-stabilizing algorithms on PIF over tree networks. Then, we compared the results of our method to the counter abstraction method.

As future work for this part, we are going to design an automated approach of abstraction for an arbitrary property to be verified. Also, designing an abstraction function that works for a system of unbounded size is desirable.

APPENDICES

Appendix A

WS Leader Election for Line4

Figures A.1 and A.3 show the cycles of Algorithm 6 for the topology shown in Figures 3.9(a) and 3.9(b), respectively. The range $\{-1, 0, \ldots, 3\}$ represents $\{\perp, p_0, p_1, p_2, p_3\}$.

Figures A.2 and A.4 show the complete Markov chain of Algorithm 6 for the topology shown in Figures 3.9(a) and 3.9(b), respectively. The black states are legitimate states, grey states are non-legitimate states that participate in a cycle, and white states are non-legitimate states that are not on a cycle.

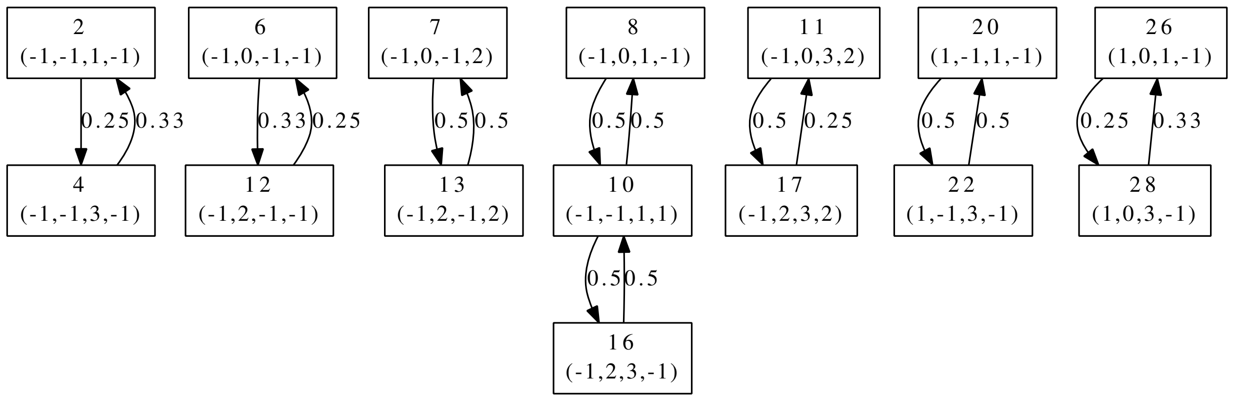


Figure A.1: Cycles of the topology shown in Figure 3.9(a).

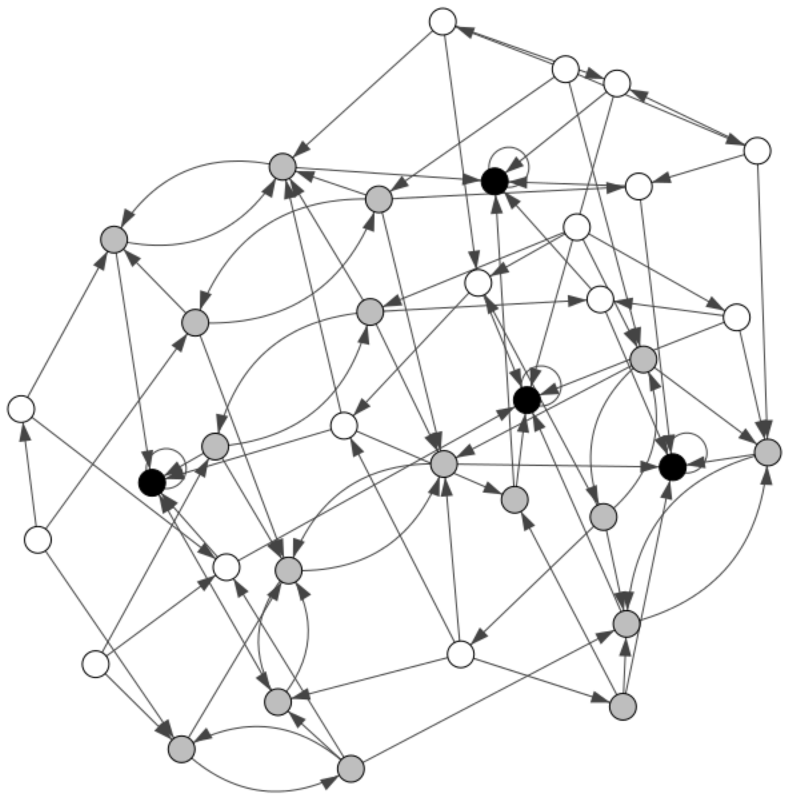


Figure A.2: Complete Markov chain of Algorithm 6 for the topology shown in Figure 3.9(a).

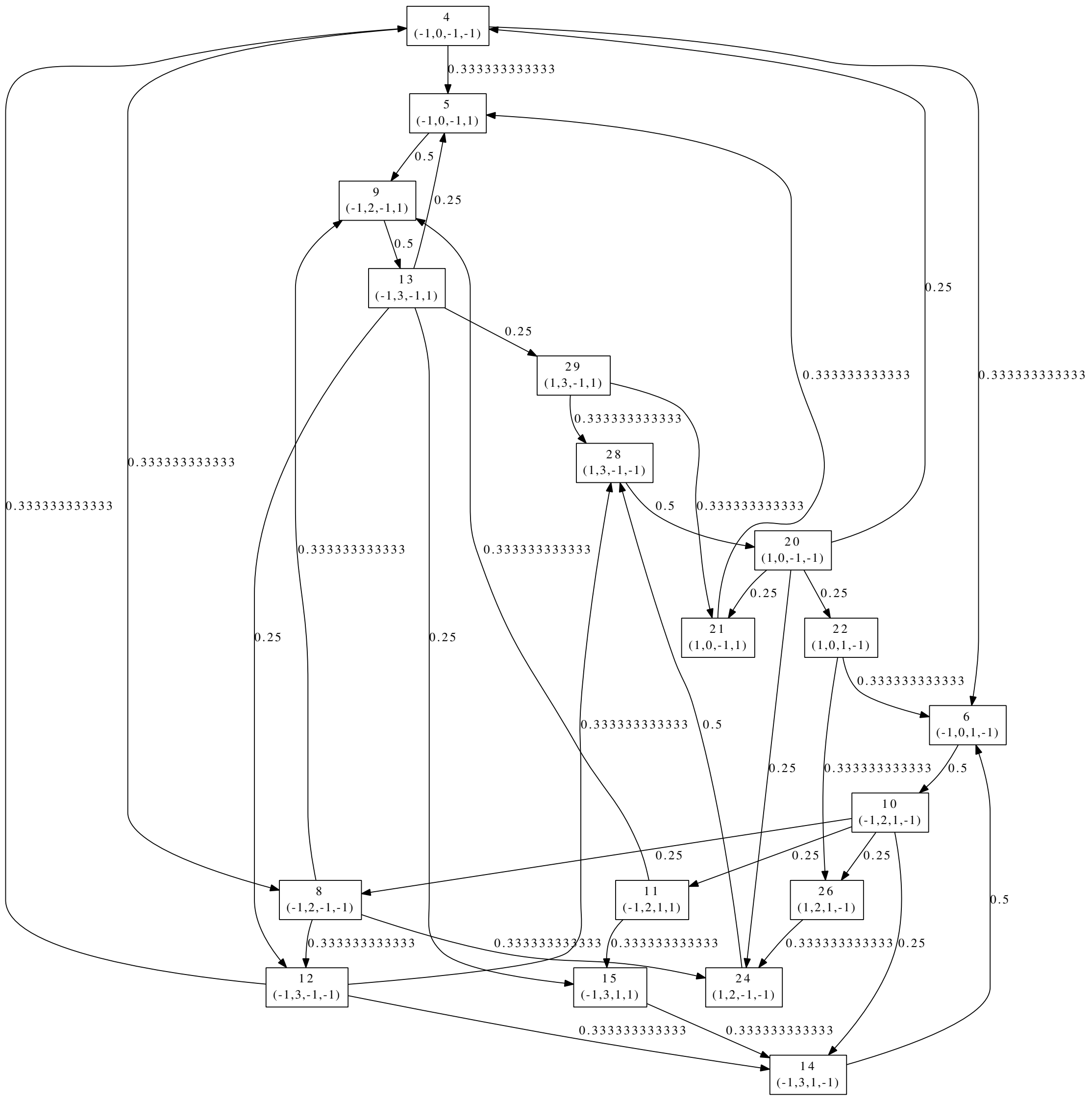


Figure A.3: All SCCs of the topology shown in Figure 3.9(b).

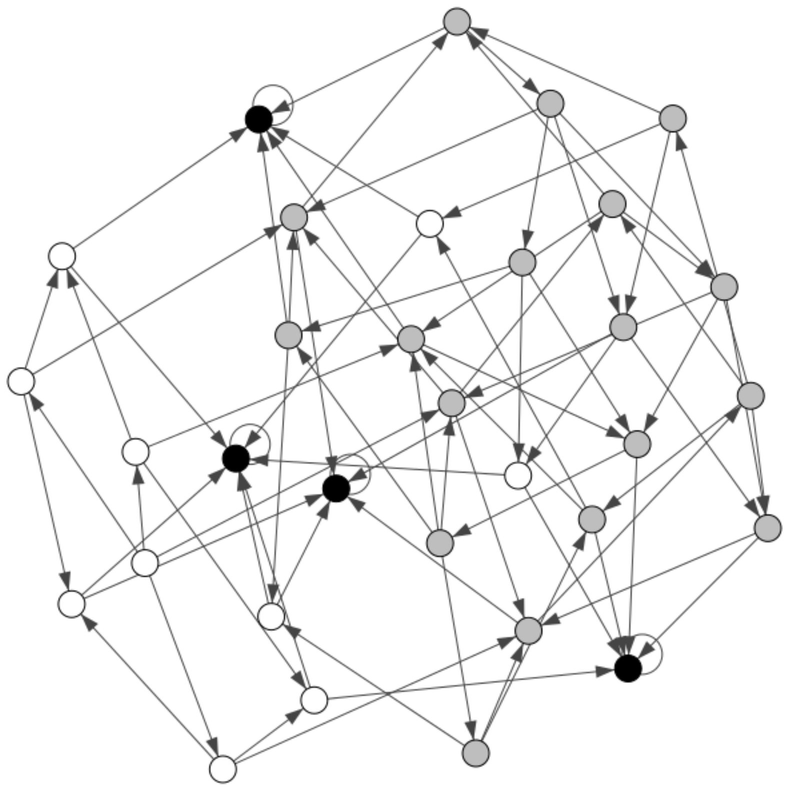


Figure A.4: Complete Markov chain of Algorithm 6 for the topology shown in Figure 3.9(b).

References

[1] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

[2] S. Dolev. *Self-Stabilization.* MIT Press 2000, March.

[3] S. Tixeuil. Self-stabilizing algorithms. In *Algorithms and Theory of Computation Handbook, Second Edition*, pages 26.1–26.45. CRC Press, Taylor & Francis Group, November 2009.

[4] A. Arora and M. G. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19(11):1015–1027, 1993.

[5] G. Varghese and M. Jayaram. The fault span of crash failures. *Journal of the ACM*, 47(2):244–293, 2000.

[6] A. K. Datta, L. L. Larmore, and P. Vemula. Self-stabilizing leader election in optimal space under an arbitrary scheduler. *Theoretical Computer Sciience*, 412(40):5541–5561, 2011.

[7] V. Chernoy, M. Shalom, and S. Zaks. A self-stabilizing algorithm with tight bounds for mutual exclusion on a ring. In *Proceedings of International Conference on Distributed Computing (DISC)*, pages 63–77, 2008.

[8] L. Blin, M. Gradinariu Potop-Butucaru, S. Rovedakis, and Sébastien Tixeuil. A new self-stabilizing minimum spanning tree construction with loop-free property. In *Proceedings of the International Conference on Distributed Computing (DISC)*, pages 407–422, September 2009.

[9] J. Adamek, M. Nesterenko, and S. Tixeuil. Using abstract simulation for performance evaluation of stabilizing algorithms: The case of propagation of information with

feedback. In *Proceedings of 14th International Conference on Stabilization, Safety, and Security in Distributed Systems (SSS)*, pages 126–132, 2012.

[10] Shlomi Dolev and Ted Herman. Superstabilizing protocols for dynamic distributed systems. page 255, 1995.

[11] E. Caron, F. Desprez, F. Petit, and C. Tedeschi. Snap-stabilizing prefix tree for peer-to-peer systems. *Parallel Processing Letters*, 20(1):15–30, 2010.

[12] N. Mitton, B. Séricola, S. Tixeuil, E. Fleury, and I. Guérin-Lassous. Self-stabilization in self-organized multihop wireless networks. *Ad Hoc and Sensor Wireless Networks*, 11(1-2):1–34, January 2011.

[13] F. Ooshita and S. Tixeuil. In *Proceedings of the International Conference on Stabilization, Safety, and Security in Distributed Systems*, pages 49–63, 2012.

[14] E. W. Dijkstra. Solution of a problem in concurrent programming control. In *Pioneers and Their Contributions to Software Engineering*, pages 289–294. Springer, 2001.

[15] L. Lamport. What good is temporal logic? In *Information Processing 83: Proceedings of the IFIP 9th World Congress*, Lecture Notes in Computer Science. North-Holland, September 1983.

[16] J. L. W. Kessels. An exercise in proving self-stabilization with a variant function. *Information Processing Letters*, 29(1):39–42, 1988.

[17] J. E. Burns and J. K. Pachl. Uniform self-stabilizing rings. *ACM Transactions on Programming Languages and Systems*, 11(2):330–344, 1989.

[18] M. Flatebo and A. K. Datta. Two-state self-stabilizing algorithms for token rings. *IEEE Transactions on Software Engineering*, 20(6):500–504, 1994.

[19] J. Beauquier and O. Debas. An optimal self-stabilizing algorithm for mutual exclusion on bidirectional non uniform rings. In *Proceedings of the Second Workshop on Self-Stabilizing Systems*, volume 17, pages 1–13, 1995.

[20] S. Dolev, A. Israeli, and S. Moran. Uniform dynamic self-stabilizing leader election. *IEEE Transactions on Parallel Distributed Systems*, 8(4):424–440, 1997.

[21] S. Ghosh and A. Gupta. An exercise in fault-containment: self-stabilizing leader election. *Information Processing Letters*, 59(5):281–288, 1996.

[22] P. Humblet. Selecting a leader in a clique in $\mathcal{O}(nlogn)$ messages. In *Memo, Laboratory for Information and Decision Systems*, LIDS Technical Reports. Laboratory for Information and Decision Systems, M.I.T., Cambridge, Mass., 1984.

[23] R. G. Gallager. *Finding a Leader in Networks With O(E) + O(N log N) Messages, Unpublished Note*. M.I.T., Cambridge, Mass.

[24] E. Chang. Echo algorithms: depth parallel operationson general graphs. *IEEE Transactions on Software Engineering*, (6):391–401, 1982.

[25] A. Segall. Distributed network protocols. *IEEE Transactions on Information Theory*, (29):23–35, 1983.

[26] A. Bui, A. Datta, F. Petit, and V. Villain. Snap-stabilizing pif algorithm in tree networks without sense of direction. In *The 6th International Colloquium on Structural Information and Communication Complexity Proceedings*, pages 32–46. Carleton University Press, 1999.

[27] A. Bui, A. Datta, F. Petit, and V. Villain. Space optimal PIF algorithm: Self-stabilizing with no extra space. In *IPCCC'99, IEEE International Performance, Computing, and Communications Conference*, pages 20–26. IEEE Computer Society Press, 1999.

[28] A. Bui, A. Datta, F. Petit, and V. Villain. State-optimal snap-stabilizing PIF in tree networks. In *Proceedings of the Fourth Workshop on Self-stabilizing Systems*, pages 78–85. IEEE Computer Society Press, 1999.

[29] A. Cournier, F. Petit, V. Villain, and A. K. Datta. Self-stabilizing PIF algorithm in arbitrary rooted networks. In *Proceedings, 21st International Conference on Distributed Computing Systems*, pages 31–48, 2001.

[30] B. Bonakdarpour N. Fallahi and S. Tixeuil. Rigorous performance evaluation of self-stabilization using probabilistic model checking. In *Reliable Distributed Systems (SRDS), 2013 IEEE 32nd International Symposium on*, pages 153–162, 2013.

[31] M. Y. Vardi. Automatic verification of probabilistic concurrent finite-state programs. pages 327–338, 1985.

[32] M. Flatebo and A. K. Datta. Simulation of self-stabilizing algorithms in distributed systems. In *Annual Simulation Symposium*, pages 32–41, 1992.

[33] N. Mullner, A. Dhama, and O. Theel. Derivation of fault tolerance measures of self-stabilizing algorithms by simulation. In *Simulation Symposium. (ANSS 2008). 41st Annual*, pages 183 –192, April 2008.

[34] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Computer Aided Verification (CAV)*, pages 585–591, 2011.

[35] M. Fischer and H. Jiang. Self-stabilizing leader election in networks of finite-state anonymous agents. In *Principles of Distributed Systems*, volume 4305 of *Lecture Notes in Computer Sciences*, pages 395–409. Springer Berlin Heidelberg, 2006.

[36] T. Herman. Probabilistic self-stabilization. *Information Processing Letters*, 35(2):63–67, 1990.

[37] D. Angluin. Local and global properties in networks of processors. In *Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing*, pages 82–93, 1980.

[38] J. Beauquier, C. Genolini, and S. Kutten. Optimal reactive k-stabilization: the case of mutual exclusion. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing. ACM*, pages 209–218, 1999.

[39] A. Bui, A. K. Datta, F. Petit, and V. Villain. State-optimal snap-stabilizing PIF in tree networks. In *Workshop on Self-Stabilizing Systems, 1999. Proceedings. 19th IEEE International Conference on Distributed Computing*, pages 78–85, 1999.

[40] M. G. Gouda. The theory of weak stabilization. In *International Workshop on Self-Stabilizing Systems*, pages 114–123, 2001.

[41] S. Devismes, S. Tixeuil, and M. Yamashita. Weak vs. self vs. probabilistic stabilization. In *Proceedings of the 28th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 681–688, 2008.

[42] O. Kupferman and M. Y. Vardi. Model checking of safety properties. In *Computer Aided Verification (CAV)*, pages 172–183, 1999.

[43] E. M. Clarke, O. Grumberg, and D. E. Long. Symmetry reductions in model checking. In *Computer Aided Verification*, Lecture Notes in Computer Science, pages 147–158. Springer, Volume 1427 1998.

[44] E Allen Emerson and Richard J Trefler. From asymmetry to full symmetry: New techniques for symmetry reduction in model checking. In *Correct Hardware Design and Verification Methods*, pages 142–157. Springer, 1999.

[45] A. Carruth. Real-time UNITY. Technical Report CS-TR-94-10, University of Texas at Austin, January 1994.

[46] A. Pnueli, J. Xu, and L. Zuck. Liveness with (0, 1,∞)-counter abstraction. In *Proceedings of the 14th International Conference on Computer Aided Verification*, volume 2403 of *Lecture Notes in Computer Science*, pages 107–122. Springer, 2002.

[47] Gérard Basler, Michele Mazzucchi, Thomas Wahl, and Daniel Kroening. Symbolic counter abstraction for concurrent software. In *Computer Aided Verification*, pages 64–78. Springer, 2009.

[48] P. Cousot and R. Cousot. Refining model checking by abstract interpretation. *Automated Software Engineering*, 6(1):69–95, 1999.

[49] B. Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *Computer Supported Cooperative Work (CSFW)*, pages 82–96, 2001.

[50] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications*, volume 6. Macmillan London, 1976.

[51] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[52] S. E. Dreyfus. An appraisal of some shortest-path algorithms. *Operations Research*, 17(3):395–412, 1969.

[53] M. Barthelemy. Betweenness centrality in large complex networks. *The European Physical Journal B-Condensed Matter and Complex Systems*, 38(2):163–168, 2004.

[54] P. Charbit, S. Thomassé, and A. Yeo. The minimum feedback arc set problem is NP-hard for tournaments. *Combinatorics, Probability and Computing*, 16(1):1–4, 2007.

[55] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[56] S. Dubois and S. Tixeuil. A taxonomy of daemons in self-stabilization. Technical Report 1110.0334, ArXiv, October 2011.

[57] C. Baier and J-P. Katoen. *Principles of Model Checking.* The MIT Press, Cambridge, 2008.

[58] K. R. Rohloff. *Computations on distributed discrete-event systems.* PhD thesis, University of Michigan, 2004.

[59] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspect of Computing*, 6(5):512–535, 1994.

[60] M. Nesterenko and S. Tixeuil. Ideal stabilization. In *Proceedings of the 25th IEEE International Conference on Advanced Information Networking and Applications (AINA)*, pages 224–231, 2011.

[61] A. Bui, A. K. Datta, F. Petit, and V. Villain. Snap-stabilization and pif in tree networks. *Distributed Computing*, 20(1):3–19, 2007.

[62] R. Mahmoudvand, H. Hassani, A. Farzaneh, and G. Howell. The exact number of nonnegative integer solutions for a linear diophantine inequality. *International Journal of Applied Mathematics*, 40(1), 2010.