How Good is Weak-Stabilization?

Narges Fallahi and Borzoo Bonakdarpour

School of Computer Science
University of Waterloo
200 University Avenue West, Waterloo, Ontario, Canada, N2L 3G1
{nfallahi,borzoo}@cs.uwaterloo.ca

Abstract. A *weak-stabilizing* system is one that guarantees only the possibility of convergence to a correct behavior; *i.e.*, a recovery path may visit an execution cycle before reaching a good behavior. To our knowledge, there has been no work on analyzing the power and performance of weak-stabilizing algorithms. In this paper, we investigate a metric for characterizing the recovery time of weak-stabilizing algorithms. This metric is based on expected mean value of recovery steps for resuming a correct behavior. Our method to evaluate this metric is based on probabilistic state exploration. We show that different weak-stabilizing algorithms perform differently during recovery, because of their structure (e.g., the length and reachability of cycles). We also introduce an automated technique that can improve the performance of implementation of weak-stabilizing algorithms through state encoding.

Keywords: Weak stabilization, Performance evaluation, Recovery time.

1 Introduction

A *self-stabilizing* [4, 5, 13] (SS) system is one that always recovers a correct behavior when the system is hit by transient faults and consequently reaches some arbitrary state [15]. Such recovery is guaranteed to reach a set of *legitimate states* within a *finite* number of execution steps. Self-stabilization is a strong property and there have been several results on impossibility of designing SS algorithms for token circulation and leader election in anonymous networks. To tackle this problem, several relaxations have been introduced. Examples include *probabilistic* SS [11], where recovery is guaranteed with probability 1, and k-stabilization [1], which assumes some arbitrary states cannot be reached and recovery takes place within k local state changes for each process. *Weak-stabilization* [9] requires that starting from any arbitrary state, there *exists* an execution path that eventually reaches a legitimate state. Thus, in a weak-stabilizing (WS) system, a recovery path may reach a cycle on its way to legitimate states (see Figure 1(b)).

There has been little attention to weak-stabilization in the literature due to relaxation of finiteness of stabilization. It is also unclear how one would evaluate the performance of a WS algorithm. In fact, conventional metrics for evaluating the performance of self-stabilization (e.g., asymptotic computation complexity)

T. Higashino et al. (Eds.): SSS 2013, LNCS 8255, pp. 148–162, 2013.

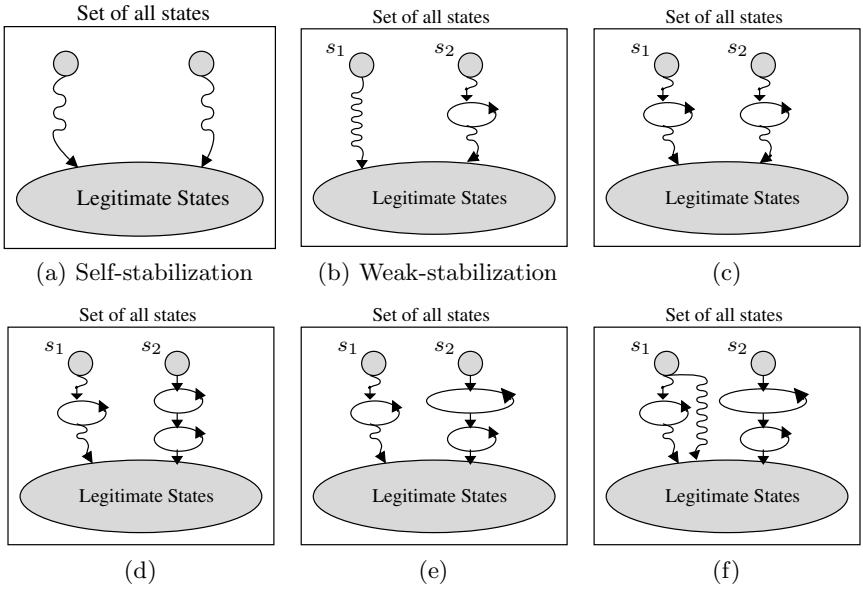


Fig. 1. Self-stabilization and different algorithm structure for weak-stabilization

become irrelevant in the context of WS algorithms. We argue that the relaxation of convergence finiteness by itself is not a plausible reason to reject weak-stabilization as a practical solution to deal with fault recovery in distributed systems for the following reasons:

- A WS algorithm in the presence of a fair scheduler is guaranteed to reach a correct behavior within a finite number of execution steps.
- 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.
- Recovery in a WS 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 WS algorithm.

To explain the latter reason in more detail, consider the structures in Figure 1. In Figure 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. Such an algorithm is expected to perform better than the one shown in Figure 1(c), since in the latter both recovery paths reach a cycle. Likewise, the algorithm in Figure 1(c) is expected to perform better than the one shown in Figure 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(d) is expected to perform better than the one shown in Figure 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(d) and 1(f), since the structure in Figure 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 where in Figure 1(b) faults do not perturb the system to a state from where the cycle is reachable. In such a scenario, the existence of the cycle becomes irrelevant.

The above analysis clearly shows that one cannot reject weak-stabilization only based on the argument of "existence of cycles during recovery". With this motivation and in the absence of performance metrics for weak-stabilization, in this paper, we focus on developing such a metric. Our metric is based on the method introduced in [8] for computing the expected mean value of recovery time (i.e., steps). Unlike the conventional asymptotic metrics (in terms of, for instance, the number of rounds [6]), the expected mean value of recovery time can be computed in WS systems, as the probability of leaving a cycle always converges to 1. In particular, 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. We utilize probabilistic model checking [14] to compute the probability of n-step reachability and subsequently the expected mean value of recovery speed. The average recovery speed computed using this technique represents the overall performance of a WS algorithm.

Since the absolute value of recovery time does not reveal much about the structure of a WS algorithm, we also propose a graph-theoretic metric that characterizes the performance of a WS algorithm by identifying its strongly connected components and incorporating their size, density, centrality, and reachability. Our analysis and experiments on a WS leader election algorithm [3] clearly show that the performance of a WS algorithm heavily depends on its structure as well as the location and length of cycles. We also introduce an algorithm that automatically generates a *state encoding* scheme for a given WS algorithm that can reduce the recovery time of the algorithm without changing its functional behavior.

Organization. In Section 2, we recap the preliminary concepts. Our performance evaluation method is described in Section 3. We introduce our graph-theoretic metric in Section 4. The algorithm for state encoding are discussed in Section 5. Finally, we make concluding remarks in Section 6.

2 Preliminaries

2.1 Distributed Systems

We model a *distributed system* as a simple self-loopless *static* undirected graph $\mathcal{G} = (V, E)$, where V is a finite set of vertices representing *processes* and E is a finite set of *edges* representing bidirectional communication, such that for all $(p, q) \in E$, we have $p, q \in V$. In this case, p and q are called *neighbors*. We

$x = 0 \longrightarrow$ **print** (‘‘safe’’)
$x = 1 \longrightarrow x := 0$
$x = 1 \longrightarrow x := 3$
$x = 2 \longrightarrow x := 1$
$x = 3 \longrightarrow x := 2$
$x = 3 \longrightarrow x := 1$
$x = 3 \longrightarrow x := 0$

Fig. 2. A set of guarded commands

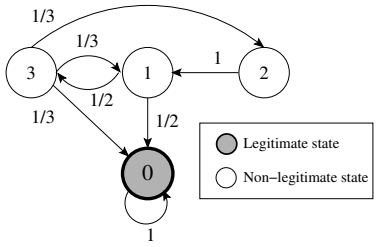


Fig. 3. Markov chain of guarded commands in Figure 2

assume that each process can distinguish all its neighbors using local indices. These indices are stored in $Neig_p = \{0, \ldots, \Delta_p - 1\}$.

The communication between processes are carried out using *locally shared variables*. Each process owns a finite set of locally shared variables, henceforth, referred to as *variables*. Each variable ranges over a fixed domain and the process can read and write them. Moreover, a process can also read variables of its neighbors in one atomic step. The *state* of a process is defined by a mapping from each variable to a value in its finite domain[1]. A process can change its state by executing its *local algorithm*. The local algorithm of a process is described using a finite set of Dijkstra's *guarded commands* (also called *actions*) of the form:

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

The *guard* of an action at process p is a Boolean expression involving a subset of variables of p and its neighbors. The *statement* of an action of p updates a subset of variables of p.

Example. We utilize the following running example to describe the concepts. Consider a system that consists of only one process. This process has a variable x that ranges over domain $\{0, 1, 2, 3\}$. The process actions are shown in Figure 2.

A *global state* s of a distributed system is an instance of the local state of its processes. We denote the set of all states of a distributed system G by S (called its *state space*). The concurrent execution of the set of all local algorithms defines a *distributed algorithm*. We say that an action of a process p is *enabled* in a state s if and only if its guard is true in s. By extension, process p is said enabled in s if and only if at least one of its actions is enabled in s. An action can be executed only if its guard is enabled. If *atomic* execution of an action in state s results in state s', we call (s, s') a *transition*.

Definition 1 (Computation). *A* computation *of a distributed algorithm is a maximal sequence of states* $\overline{\sigma} = s_0 s_1 \ldots$, *such that for all* $i \geq 0$, *each pair*

[1] We note that finiteness of processes, variables, and domains is due to the fact that our approach in this paper is based on model checking.

(s_i, s_{i+1}) is a transition. Maximality of a computation means that the computation is either infinite or eventually reaches a terminal state; i.e., *a state where no action is enabled.* □

2.2 Probabilistic Model Checking

In order to reason about distributed systems, we focus on their state space and set of transitions. Let AP be a set of atomic propositions.

Definition 2 (Markov Chains). *A* discrete-time Markov chain *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*
- $\mathbb{P} : S \times S \to [0, 1]$ *is a function such that for all* $s \in S$, *we have*

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

- $L : S \to 2^{AP}$ *is a labeling function assigning to each state a set of atomic propositions.* □

It is straightforward to see that one can represent a distributed algorithm as a Markov chain. In a Markov chain, the fact that $\mathbb{P}(s, s') \neq 0$ for two states $s, s' \in S$ stipulates there is a transition from s to s' that can be executed with some probability. We emphasize that representing a distributed algorithm in our framework based on a Markov chain does not make our approach in this paper suitable for only probabilistic distributed algorithms (e.g., [11]). In particular, if a distributed algorithm is not probabilistic, then it can be modeled as a Markov chain, where the probability of all outgoing transitions from each state are equal, unless the system scheduler imposes certain probability constraints. Also, as argued in [8], for a distributed system, its corresponding Markov chain can be constructed in such a way that it is augmented with a certain type of scheduler (i.e., central, distributed, fair, etc).

Example. The Markov chain of the guarded commands in Figure 2 is shown in Figure 3. Each state is labeled by the value of variable x. Each transition is annotated by the probability of its occurrence.

Probabilistic model checking is based on the definition of a probability measure over the set of paths that satisfy a given property specification. Our specification language in this paper is the Probabilistic Computation Tree Logic (PCTL).

Definition 3 (PCTL Syntax). *Formulas in* PCTL *[10] are inductively defined 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 $p \in AP$, $\sim \in \{<, \leq, \geq, >, =\}$ *is a comparison operator, and* λ *is probability threshold. The sub-formula* $\varphi_1 \mathcal{U}^h \varphi_2$ *is the classic (bounded/unbounded) "until" operator.* □

Definition 4 (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. Semantics of* PCTL *is defined inductively as follows:*

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

In addition, $\overline{\sigma} \models \varphi$ *holds iff* $\overline{\sigma}, 0 \models \varphi$ *holds. Finally, for a state s, we have* $s \models \mathbb{P}_{\sim\lambda}\varphi$ *iff the probability of taking a path from s that satisfies* φ *is* $\sim \lambda$. □

Following Definition 4, we use the usual abbreviation $\Diamond\varphi \equiv (\mathit{true}\,\mathcal{U}\,\varphi)$ for "eventually" formulas. Moreover, $\Box\varphi \equiv \neg\Diamond\neg\varphi$ is the classical "globally" path formula.

Example. It is straightforward to see that the Markov chain in Figure 3 satisfies the following PCTL properties:

- $\mathbb{P}_{=\frac{1}{3}}(x = 3) \Rightarrow \Diamond^1(x = 0)$.
- $\mathbb{P}_{=\frac{2}{9}}(x = 3) \Rightarrow \Diamond^3(x = 0)$.
- $\mathbb{P}_{=1}\Diamond(x = 0)$, which is logically equal to $\mathbb{P}_{=1}(1 \leq x \leq 3)\,\mathcal{U}\,(x = 0)$

Definition 5. *Let* $D = (S, S^0, \mathbb{P}, L)$ *be a Markov chain. A* state predicate *of D is a subset of S.* □

Observe that a state predicate is a PCTL formula constructed by only atomic propositions and Boolean operators $\neg$ and $\vee$.

2.3 Weak-Stabilization

Intuitively, a weak-stabilizing (WS) system is one that if its execution starts from any *arbitrary* state, then from that state there *exists* an execution path that reaches a good behavior (called the *convergence* property) and after convergence, it behaves normally unless its state is perturbed by transient faults (called *closure* property). The so-called "good behavior" is normally modeled by a state predicate called *legitimate states*. 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. Following the result in [3] that a non-probabilistic finite-state weak-stabilizing protocol in the presence of a probabilistic scheduler can be turned into a probabilistic self-stabilizing protocol, we formally define the notion of weak-stabilization as follows.

Definition 6 (Weak-stabilization). *Let $D = (S, S^0, \mathbb{P}, L)$ be a Markov chain, representing a distributed algorithm, and LS be a non-empty state predicate. We say that D is* weak-stabilizing *for* legitimates states *LS iff the following two conditions hold:*

- (Closure) *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.*
- (Convergence) *We have $\mathbb{P}_{=1}\Box\Diamond LS$;* i.e., *starting from any arbitrary state, the probability of reaching the legitimate states is always* 1. □

In Definition 6, 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 3 is weak-stabilizing for $LS \equiv (x = 0)$.

3 Rigorous Computation of Recovery Time in Weak-Stabilization

Let $D = (S, S^0, \mathbb{P}, L)$ be a Markov chain with legitimate states LS and $\mathfrak{s}$ be a state in $\neg LS$. Following Definition 4, the PCTL property

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

holds in D, if starting from $\mathfrak{s}$ the probability of reaching LS in h steps is greater than or equal to p. In order to analyze the speed of recovery of a self-stabilizing algorithm, we consider the other side of the coin by computing the probability of truthfulness of the following expression

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

for each state $\mathfrak{s} \in \neg LS$. Let μ be the probability distribution on recovery paths and $R(\mathfrak{s})$ denote the length of a recovery path that starts from $\mathfrak{s}$. One can compute the probability of existence of recovery paths of length at most N from state $\mathfrak{s}$ as follows:

$$\mathbb{P}\{R(\mathfrak{s}) \leq N\} = \sum \left\{ \mu(\overline{\sigma} = s_0 s_1 \dots) \,\middle|\, (s_0 = \mathfrak{s}) \wedge (\overline{\sigma} \models \Diamond^h LS) \wedge (h \leq N) \right\} \quad (1)$$

Example. For the Markov chain in Figure 3, the probability of recovery of length at most 2 from state 3 to state 0 is: $\frac{1}{3} \times \frac{1}{2} + \frac{1}{3} = \frac{1}{2}$.

One can also compute the expected mean value of $R(\mathfrak{s})$, that is, the average length of all recovery paths that start from state $\mathfrak{s}$. Again, this value can be computed by direct summation as follows:

$$\mathbb{E}\{R(\mathfrak{s})\} = \sum \left\{ \mu(\overline{\sigma} = s_0 s_1 \dots) \times h \,\middle|\, (s_0 = \mathfrak{s}) \wedge (\overline{\sigma} \models \Diamond^h LS) \right\} \quad (2)$$

Example. For the Markov chain in Figure 3, the expected mean value of recovery steps for the three states in $\neg LS$ are $\mathbb{E}\{R(1)\} \approx 2.5$, $\mathbb{E}\{R(2)\} \approx 3.5$, and $\mathbb{E}\{R(3)\} \approx 3$. These values can be obtained by computing infinite Taylor series that handle convergence of cycles in Figure 3.

In order to compute the expected value of recovery time, one can also incorporate the likelihood of execution from an arbitrary initial states. Thus, we associate a probability $p(\mathfrak{s})$ to each state $\mathfrak{s} \in \neg LS$, such that

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

Subsequently, the mean expected value of recovery steps for a weak-stabilizing algorithm represented by a Markov chain D is the following:

$$\mathbb{E}\{R(D)\} = \sum \left\{ \mathbb{E}\{R(\mathfrak{s})\} \times p(\mathfrak{s}) \;\middle|\; \mathfrak{s} \in \neg LS \right\} \tag{3}$$

Example. For the Markov chain in Figure 3, assuming uniform probability distribution for all non-legitimate state, the expected mean value of recovery is

$$2.5 \times \frac{1}{3} + 3.5 \times \frac{1}{3} + 3 \times \frac{1}{3} = 3$$

Another observation is that removing the transition from state 3 to 2 will break one of the cycles and results in expected recovery time 2.3 and removing the transition from state 3 to 1 will break all cycles and reduces the recovery time to 1.5.

Given $D_1 = (S_0, S_1^0, \mathbb{P}_1, L_1)$ and $D_2 = (S_2, S_2^0, \mathbb{P}_2, L_2)$, we say that D_1 outperforms D_2 iff

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

4 Structural Analysis of Weak-Stabilization

Although our approach in Section 3 can be applied to any WS algorithm, we choose the WS leader election algorithm for anonymous trees introduced in [3] for our study (see Algorithm 1). 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.

Consider the network topologies shown in Figure 4 and their expected recovery times obtained from Equation 3 through implementing the algorithms in the probabilistic model checker PRISM [12]. Although the recovery times clearly show which topology converges faster when running Algorithm 1, one needs more insightful information in order to analyze the behavior of the algorithm. To this end, we analyze the structure of the Markov chain of Algorithm 1 running on each topology in Figure 4. Before analyzing Algorithm 1, we introduce our structural analysis technique for performance evaluation of WS algorithms.

Algorithm 1. WS Leader Election [3] for any Process p

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

Actions:

1: $(Par_p \neq \bot) \wedge (|Children_p| = |Neig_p|) \longrightarrow Par_p := \bot$
2: $(Par_p \neq \bot) \wedge [Neig_p \backslash (Children_p \cup \{Par_p\}) \neq \emptyset] \longrightarrow Par_p := (Par_p + 1) \mod \Delta_p$
3: $(Par_p = \bot) \wedge (|Children_p| < |Neig_p|) \longrightarrow Par_p := \min_{\prec p}(Neig_p \backslash Children_p)$

4.1 A Graph-Theoretic Metric for Performance Evaluation of WS Algorithms

As mentioned earlier, measuring the performance of WS algorithms boils down to analyzing the structure of its cycles outside legitimate states. Since enumerating all cycles and computing their reachability from each other incurs an exponential time complexity in the size of Markov chain, we focus on two graph-theoretic abstractions that represent the existence and connectivity of cycles:

- *Strongly connected components (SCC).* An SCC is a subgraph in which each pair of vertices are reachable from each other. An SCC contains at least one cycle. In order to analyze nested cycles in an SCC, we also consider the number of states and transitions in the SCC. That is, the higher the number of transitions, the more likely it is for the SCC to have nested cycles.
- *Betweenness centrality (BC) [2].* Betweenness centrality quantifies the number of times a vertex acts as a bridge along the shortest path between two other vertices. We, in particular, employ vertex betweenness to compute the centrality of states in Markov chains. Formally, for a Markov chain with set of states S, BC of a state $s \in S$ is defined as follows:

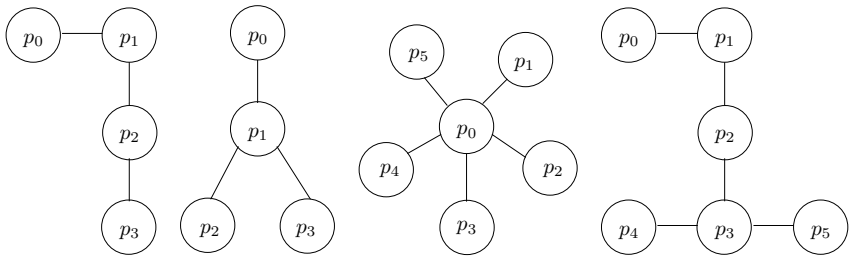
$$C_B(s) = \sum_{s \neq s' \neq s''} \frac{SP_{s's''}(s)}{SP_{s's''}}$$

where $SP_{s's''}$ is the total number of shortest paths from state s' to state s'' and $SP_{s's''}(s)$ is the number of those paths that pass through s. In the context of our problem, we compute betweenness of states in SCCs in order to identify the number of paths that pass through cycles.

Thus, in general, we characterize the performance of a WS algorithm modeled by Markov chain D using the above factors as follows:

$$\mathcal{R} = |D_{scc}|. \sum_{scc \in D} (C_B(scc) \cdot \frac{E_{scc}}{V_{scc}}) \tag{4}$$

where D_{scc} denotes the set of SCCs in D whose states are in $\neg LS$, $C_B(scc)$ is the average BC of the states in scc, and V_{scc} and E_{scc} denote the number of states (respectively, transitions) in scc. The last fraction in Equation 4 incorporates the *graph density* of an SCC. We note that since Equation 4 is an abstraction



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

Fig. 4. Different topologies for Algorithm 1

to explain to the performance of a WS algorithm, it may fail by outliers. For instance, Equation 4 fails in Markov chains whose SCCs are small (*i.e.*, less than 4 states).

4.2 Analysis of WS Leader Election Algorithm

Figures 4(a) and 4(b) show the two possible topologies for a network of size 4 for Algorithm 1. The Markov chain of the topology in Figure 4(a) includes three SCCs: one SCC of size 3 states and 6 SCCs of size 2. SCCs of size two are due to the fact that processes p_1 and p_2 can keep changing their parents. For example, 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 1 and choose p_1 as its parent, where the new global state is $(p_1, \perp, p_1, \perp)$. From this state, p_2 can again choose p_3 as its parent by executing the same action, hence, the cycle. There are $3 * 2 = 6$ combinations of such cycles. The SCC of size 3 is due to the following scenario. In state $s_0 = (\perp, p_0, p_3, \perp)$, 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, an SCC of size 3.

On the contrary, the Markov chain of the topology in Figure 4(b) includes only one SCC (due to process p_1 and its three neighbors) of size 18 states and 36 transitions. This implies that although this SCC has at least one (not necessarily simple) cycle of length 18, it is quite sparse. In other words, the SCC does not contain a high number of nested cycles and, hence, does not incur a high recovery time. Nested cycles and chain of cycles tend to increase expected recovery time exponentially. While the topology in Figure 4(b) has a long cycle, it does not have a high number of nested cycles and it does not have chain of individual cycles. Unlike this case, in the topology in Figure 4(a), the SCCs form a relatively long chain of cycles, which causes higher recovery time. We note that due to the size of SCCs for 4 processes, Equation 4 does not reflect the structure of its Markov chains properly.

For 6 processes, we only focus on the best and worst cases. The star topology in Figure 4(c) exhibits the best performance among all 6-process topologies. Intuitively, this is because the network tends to choose the center of the star (*i.e.*, process p_0) as the leader faster. This intuition is reasonable since the Markov chain of the star topology has only one SCC with low density 3.3, BC 272, and $\mathcal{R} = 899$. The worst case recovery is due to the topology shown in Figure 4(d). Our analysis shows that this topology creates 27 SCCs with nested cycles chained throughout the non-legitimate states. In addition, some larger SCCs are highly dense with high BC. For this topology $\mathcal{R} = 438,152$.

Likewise, our analysis shows that even for larger sizes of network, the best and worst case performance belongs to similar topologies. In particular, for 7 processes, the best performance is due to the star topology ($\mathbb{E}\{R\} = 5.8$ and $\mathcal{R} = 10,664$) and the worst performance is due to the line topology with ending branching ($\mathbb{E}\{R\} = 8.2$ and $\mathcal{R} = 1,474,708$).

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

- Equation 3 provides us with a rigorous fine-grained approach to evaluate the performance of WS algorithms.
- Equation 4 characterizes a coarse-grained approach to obtain the performance of a WS algorithm, but provides us the means to analyze and reason about the performance and structure of WS algorithms.
- These equations together are valuable tools for developers of distributed and network protocols to design more efficient WS algorithms.

5 Automated State Encoding for Weak-stabilizing Algorithms

State encoding consists in assigning bit patterns to abstract algorithm states [8]. For example, two states s and s' can be associated to bits `0` and `1`, respectively. Another possibility is to have bit patterns of size two and mapping s to {`00`} and s' to {`01, 10, 11`}. In the first case, there is a bijection between bit pattern and abstract states. In the second case, the mapping is injective, but not surjective.

In the context of WS algorithms, the purpose of such state encoding is twofold:

1. increase the proportion of LS as compared to $\neg LS$ 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. decrease the average expected recovery time by making states that are engaged in cycles outside legitimate states and incur long recovery time executions less likely to appear than those belong to short recovery time executions.

Example. As shown for the Markov chain in Figure 3, $\mathbb{E}\{R(1)\} = 2.5$, $\mathbb{E}\{R(2)\} = 3.5$, $\mathbb{E}\{R(3)\} = 3.0$, and $\mathbb{E}\{R\} = 3$. The set of states before encoding is $\{0, \ldots, 3\}$. Let the new domain of states be $\{0, \ldots, 6\}$. Since state 1 has the smallest expected value for recovery, we map it to values $\{1, 4, 5\}$. Likewise, we map state 3

Algorithm 2. State Encoding for WS Algorithms (based on feedback arc set)

Input: Processes $\{p_0 \dots p_{n-1}\}$ and corresponding 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 $rank(v_s)$ 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 sorted list of $rank(v)$ for each $v \in V'$ (respectively, $rank(u)$ for each $u \notin V'$)
6: $U := U_1 \cdot U_2$
7: Divide U into n sub-lists $\{u_0 \dots u_{n-1}\}$
8: **for** $(i = 0$ **to** $n - 1)$ **do**
9: Encode(u_i)
10: **end for**

to $\{3, 6\}$ and state 2 to $\{2\}$. By applying this encoding, the new expected mean value of recovery will reduce to $\mathbb{E}\{R\} = 2.83$.

We now introduce a state encoding algorithm based on the insights gained through our performance analysis techniques presented in Sections 3 and 4. We emphasize that state encoding is only an *implementation* technique and does not change the functional behavior of the original algorithm. Intuitively, the algorithm attempts to map more state bits to states that do not appear on a cycle. To this end, the algorithm utilizes the concept of *feedback arc set* (FAS); *i.e.*, a set of arcs whose removal makes a digraph acyclic. The algorithm uses the source vertices of these arcs to generate a state bit mapping.

The input to Algorithm 2 are n processes and a Markov chain D. 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 path to some state in LS (Line 2). Next, it attempts to find the minimum FAS (Line 3). Since finding minimum FAS is NP-complete, we utilize an approximation algorithm [7] to find a near-optimal solution. Next, we sort the vertices that originate from an arc in the FAS in the list U_1 and the rest of the vertices in the list U_2 (Line 6) based on their ranks. The actual encoding (Line 9) occurs on n equal-size slices of concatenation of U_1 and U_2.

The function Encode works as follows. For each variable in process p_i, we define two domains for before and after encoding, denoted D_i and D'_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. Then, we map the most probable value onto i values in D'_i and subsequently remove the original value from D_i. Since the states in $u_0 \cdots u_{n-1}$ are sorted, 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} are mapped to n values in D'_i. If D_i has only one element left and there are still some u_j's left to analyze, we compare the probability of that value in different

u_i's, and decide on the mapping with the most probable one. In case of a tie, we skip the slice.

Example. Consider Algorithm 1 with 4 processes with the topology shown in Figure 4(a), where range $\{-1, 0, \ldots, 3\}$ represents $\{\perp, p_0, p_1, p_2, p_3\}$. Thus, considering the neighborig 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 2, we obtain the following slices:

$u_0 = \{(-1,0,-1,-1),(-1,0,1,-1),(-1,0,3,-1),$
$(-1,2,-1,-1),(-1,2,3,-1),(1,-1,3,-1),(1,0,3,-1)\}$
$u_1 = \{(-1,-1,1,-1),(-1,-1,3,-1),(-1,0,-1,2),$
$(-1,0,3,2),(-1,2,-1,2),(1,2,3,2)\}$
$u_2 = \{(-1,2,1,-1),(1,2,-1,-1),(1,2,1,-1),(1,0,1,2),(1,2,1,2)\}$
$u_3 = \{(-1,-1,-1,-1),(-1,-1,-1,2),(-1,-1,1,2),(-1,-1,3,2),(-1,2,1,2),$
$(1,-1,-1,-1),(1,-1,-1,2),(1,-1,3,2),(1,0,-1,-1),(1,0,-1,2),(1,0,3,2)\}$

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

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

Thus, we map $-1 \in D_0$ to only 1 value in D'_0 and update D_0 by removing -1 and D'_0 by adding -1. The application of this mapping to domains $D_1 \cdots D_3$ and $D'_1 \cdots D'_3$ are illustrated in Table 1.

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 remaining. 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 1.

For u_2, we have:

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

Table 1. Iterations of state encoding for $i = 0$ to 3 (Line 9 of Algorithm 2 for WS leader election and 4 processes for the topology shown in Figure 4(a))

	$i=0$			$i=1$			$i=2$			$i=3$		
	$\max \mathbb{P}$	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	$\emptyset$	$\{-1,0,1,2,3\}$

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 in summarized in Table 1. Using the above state encoding technique, the expected recovery time of WS leader election for the topology shown is Figure 4(a) (respectively, Figure 4(b)) reduces 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 original model.

6 Conclusion

Evaluating the performance of weak-stabilizing (WS) algorithms has not been studied in the literature, since such algorithms may exhibit behaviors that never stabilize. In this paper, we proposed an automated method for evaluating the performance of distributed WS algorithms using probabilistic verification. Our method computes the expected recovery time of each state of a WS algorithm using rigorous state exploration from which the overall expected mean value of recovery time of the algorithm can be obtained. To our knowledge, this is the first attempt in studying the performance of WS systems.

Since the expected recovery time as an absolute value is not particularly insightful, we also proposed a graph-theoretic method to analyze the structure of WS algorithms. This method is based on identifying strongly connected components and their betweenness centrality in the reachability graph of a WS algorithm. We showed that the corresponding metric is in strong correlation with expected recovery time in WS algorithms. All our claims are backed by experiments on a WS leader election algorithm [3]. Using our insights, we also introduced an algorithm that generate state bit maps (called *state encoding*) that improve the recovery time in an implementation of a WS system.

As for future work, we are planning to design algorithms that synthesize WS systems that satisfy given recovery time objectives. 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.

Acknowledgements. This research was supported in part by Canada NSERC Discovery Grant 418396-2012 and NSERC Strategic Grant 430575-2012.

References

1. Beauquier, J., Genolini, C., Kutten, S.: k-stabilization of reactive tasks. In: Proceedings of the 17th ACM Symposium on Principles of Distributed Computing (PODC), pp. 318–327 (1998)
2. Brandes, U.: On variants of shortest-path betweenness centrality and their generic computation. Social Nnetworks 30(2) (2008)
3. Devismes, S., Tixeuil, S., Yamashita, M.: Weak vs. self vs. probabilistic stabilization. In: Proceedings of the 28th IEEE International Conference on Distributed Computing Systems (ICDCS), pp. 681–688 (2008)
4. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Communications of the ACM 17(11), 643–644 (1974)
5. Dolev, S.: Self-stabilization. MIT Press (March 2000)
6. Dolev, S., Israeli, A., Moran, S.: Uniform dynamic self-stabilizing leader election. IEEE Transactions on Parallel Distributed Systems 8(4), 424–440 (1997)
7. Eades, P., Lin, X., Smyth, W.F.: A fast effective heuristic for the feedback arc set problem. Information Processing Letters (IPL) 47, 319–323 (1993)
8. Fallahi, N., Bonakdarpour, B., Tixeuil, S.: Rigorous performance evaluation of self-stabilization using probabilistic model checking. In: Proceedings of the 32nd IEEE International Conference on Reliable Distributed Systems, SRDS (to appear 2013)
9. Gouda, M.G.: The theory of weak stabilization. In: Datta, A.K., Herman, T. (eds.) WSS 2001. LNCS, vol. 2194, p. 114. Springer, Heidelberg (2001)
10. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Formal Aspect of Computing (FAOC) 6(5), 512–535 (1994)
11. Herman, T.: Probabilistic self-stabilization. Information Processing Letters 35(2), 63–67 (1990)
12. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011)
13. Tixeuil, S.: Self-stabilizing Algorithms. In: Algorithms and Theory of Computation Handbook, Chapman & Hall/CRC Applied Algorithms and Data Structures, 2nd edn., pp. 26.1–26.45. CRC Press, Taylor & Francis Group (November 2009)
14. Vardi, M.Y.: Automatic verification of probabilistic concurrent finite-state programs. In: Proceedings of the 26th Annual Symposium on Foundations of Computer Science (FOCS), pp. 327–338 (1985)
15. Varghese, G., Jayaram, M.: The fault span of crash failures. J. ACM 47(2), 244–293 (2000)