Dependability Engineering of Silent Self-Stabilizing Systems*

Abhishek Dhama[1], Oliver Theel[1], Pepijn Crouzen[2], Holger Hermanns[2],
Ralf Wimmer[3], and Bernd Becker[3]

[1] System Software and Distributed Systems, University of Oldenburg, Germany
{abhishek.dhama, theel}@informatik.uni-oldenburg.de
[2] Dependable Systems and Software, Saarland University, Germany
{crouzen, hermanns}@cs.uni-saarland.de
[3] Chair of Computer Architecture, Albert-Ludwigs-University Freiburg, Germany
{wimmer, becker}@informatik.uni-freiburg.de

Abstract. Self-stabilization is an elegant way of realizing non-masking fault-tolerant systems. Sustained research over last decades has produced multiple self-stabilizing algorithms for many problems in distributed computing. In this paper, we present a framework to evaluate multiple self-stabilizing solutions under a fault model that allows intermittent transient faults. To that end, metrics to quantify the dependability of self-stabilizing systems are defined. It is also shown how to derive models that are suitable for probabilistic model checking in order to determine those dependability metrics. A heuristics-based method is presented to analyze counterexamples returned by a probabilistic model checker in case the system under investigation does not exhibit the desired degree of dependability. Based on the analysis, the self-stabilizing algorithm is subsequently refined.

1 Introduction

Self-stabilization has proven to be a valuable design concept for dependable systems. It allows the effective realization of non-masking fault-tolerant solutions to a problem in a particularly hostile environment: an environment subject to arbitrarily many transient faults potentially corrupting the self-stabilizing system's run-time state of registers and variables. Consequently, designing a self-stabilizing system is not an easy task, since many scenarios due to faults must correctly be handled beyond the fact that the system has to solve a given problem when being undisturbed by faults. The formal verification of a self-stabilizing solution to a given problem is therefore often quite complicated. It consists of a 1) convergence proof showing that the system eventually returns to a set of system states (called *safe* or *legal states*) where it solves the given problem and 2) a closure proof showing that once within the set of legal states, it does not leave this set voluntarily in the absence of faults occurring. Whereas the closure proof is often not too complicated, the convergence proof may become extremely challenging. It requires some finiteness argument showing the return of the system

* This work was supported by the German Research Foundation (DFG) under grant SFB/TR 14/2 "AVACS," www.avacs.org.

to the legal state set in a finite number of computational steps in the absence of newly manifested faults.

As discussed, finding a self-stabilizing solution to a given problem as well as proving its self-stabilization property are generally not easy and present areas of agile research. But what, if multiple self-stabilizing solutions to a problem are already known? Which solution should be preferred and therefore be chosen? Clearly, many criteria do exists and their relevance depends on the concrete application scenario.

In this paper, we focus on dependability properties of those systems. For example: "Does the given self-stabilizing system exhibit a system availability of at least p?" with system availability being only an example of a dependability metrics. Other metrics are, e.g., reliability, mean time to failure, and mean time to repair. Based on the evaluation of relevant dependability metrics, a decision should be taken of which solution out of the set of present solutions should be chosen and put to work for ones purposes. By building on [1], we present useful dependability metrics for differentiating among self-stabilizing solutions and show how to evaluate them. For this purpose, we propose the modeling of a self-stabilizing algorithm together with the assumed fault model in terms of a discrete-time Markov decision process or a discrete-time Markov chain. Whereas the former modeling allows – with the help of a probabilistic model checker – to reason about the behavior of the system under any fair scheduler, the latter modeling is suitable if concrete information about the scheduler used in the system setting is available. The self-stabilizing solution exhibiting the best dependability metrics value can then easily be identified and used.

Furthermore, we show a possible way out of the situation where all available self-stabilizing solutions to a given problem have turned out to fail in the sense described above: if the dependability property under investigation cannot be verified for a particular system, then an automatically generated counterexample (being a set of traces for which, as a whole, the property does not hold) is prompted. By analyzing the counterexample, the self-stabilizing algorithm is then refined and again model-checked. This refinement loop is repeated until the dependability property is finally established or a maximal number of refinement loops has been executed.

In the scope of the paper, wrt. abstraction scheme and system refinement, we restrict ourselves to *silent* self-stabilizing algorithms and a dependability metrics being a notion of limiting (system) availability called *unconditional limiting availability* in a system environment where faults "continuously keep on occurring." Silent self-stabilizing algorithms do not switch among legal states in the absence of faults. Unconditional limiting availability is a generalization of limiting availability in the sense that any initial state of the system is allowed. Finally, with the more general fault model, we believe that we can analyze self-stabilizing systems in a more realistic setting: contrarily to other approaches, we do not analyze the system only after the last fault has already occurred but always allow faults to hamper with the system state.

The paper is structured as follows: in Section 2, we give an overview of related work. Then, in Section 3, we introduce useful dependability metrics for self-stabilizing systems. Additionally, we state the model used for dependability metrics evaluation based on discrete-time Markov decision processes or discrete-time Markov chains. Section 4 describes the refinement loop and thus, dependability engineering based on probabilistic model-checking, counterexample generation, counterexample analysis, and silent self-stabilizing system refinement along with an abstraction scheme to overcome scalability problems. Section 5, finally, concludes the paper and sketches our future research.

2 Related Work

The body of literature is replete with efforts towards the engineering of fault-tolerant systems to increase dependability. In [2], a formal method to design a – in a certain sense – multitolerant system is presented. The method employs *detectors* and *correctors* to add fault tolerance with respect to a set of fault classes. A detector checks whether a state predicate is satisfied during execution. A corrector ensures that – in the event of a state predicate violation – the program will again satisfy the predicate. It is further shown in [3] that the detector-corrector approach can be used to obtain masking fault-tolerant from non-masking fault-tolerant systems. But, despite its elegance, the fault model used in their applications admits only transient faults.

Ghosh *et al.* described in [4] an approach to engineer a self-stabilizing system in order to limit the effect of a fault. A transformer is provided to modify a non-reactive self-stabilizing system such that the system stabilizes in constant time if a single process is faulty. However, there is a trade-off involved in using the transformer as discussed in [5]. The addition of such a transformer to limit the recovery time from a single faulty process might lead to an increase in stabilization time.

A compositional method, called "cross-over composition," is described in [6] to ensure that an algorithm self-stabilizing under a specific scheduler converges under an arbitrary scheduler. This is achieved by composing the target "weaker" algorithm with a so-called "strong algorithm" such that actions of the target algorithm are synchronized with the strong algorithm. The resultant algorithm is self-stabilizing under any scheduler under which the strong algorithm is proven to be self-stabilizing. However, the properties of the strong algorithm determine the class of schedulers admissible by the composed algorithm.

Recent advances in counterexample generation for stochastic model checking has generated considerable interest in using the information given by the counterexamples for debugging or optimizing systems. An interactive visualization tool to support the debugging process is presented in [7]. The tool renders violating traces graphically along with state information and probability mass. It also allows the user to selectively focus on a particular segment of the violating traces. However, it does not provide any heuristics or support to modify the system in order to achieve the desired dependability property. Thus, the user must modify systems by hand without any tool support. In addition to these shortcomings, only models based on Markov chains are handled by the tool. In particular, models containing non-determinism cannot be visualized.

We will next describe the method to evaluate dependability metrics of self-stabilizing systems with an emphasis on silent self-stabilizing systems.

3 Dependability Evaluation of Self-Stabilizing Systems

We now present a procedure with tool support for evaluating dependability metrics of self-stabilizing systems. A self-stabilizing BFS spanning tree algorithm given in [8] is used as a working example throughout the sections to illustrate each phase of our proposed procedure. Note that the method nevertheless is applicable to any other self-stabilizing algorithm as well.

3.1 Dependability Metrics

The definition and enumeration of metrics to quantify the dependability of a self-stabilizing system is the linchpin of any approach for dependability evaluation. This becomes particularly critical in the case of self-stabilizing systems as the assumptions made about the frequency of faults may not hold true in a given implementation scenario. That is, faults may be intermittent and the temporal separation between them, at times, may not be large enough to allow the system to converge. In this context *reliability*, *instantaneous availability*, and *limiting availability* have been defined for self-stabilizing systems in [1]. An important part of these definitions is the notion of a system doing "something useful." A self-stabilizing system is said to do something useful if it satisfies the safety predicate (which in turn specifies the set of legal states) with respect to which it has been shown to be self-stabilizing.

We now define the mean time to repair (MTTR) and the mean time to failure (MTTF) along with new metrics called *unconditional instantaneous availability* and *generic instantaneous availability* for self-stabilizing systems. These metrics are *measures* (in a measure theoretic sense) provided the system under study can be considered as a stochastic process. It is natural to consider discrete-state stochastic processes, where the set of states is divided into a set of operational states ("up states") and of disfunctional states ("down states").

The basic definition of instantaneous availability at time t quantifies the probability of being in an "up state" at time t [9]. Some variations are possible with respect to the assumption of the system being initially available, an assumption that is not natural in the context of self-stabilizing systems, since these are designed to stabilize from any initial state. Towards that end, we define *generic instantaneous availability* and *unconditional instantaneous availability* and apply them in the context of self-stabilizing systems. Our natural focus is on systems evolving in discrete time, thus where the system moves from states to states in steps. Time is thus counted in steps, and s_i refers to the state occupied at time i.

Generic instantaneous availability at step k $A_{\mathcal{G}}(k)$ is defined as probability $Pr(s_k \models \mathcal{P}_{\mathrm{up}} \mid s_0 \models \mathcal{P}_{\mathrm{init}})$, where $\mathcal{P}_{\mathrm{up}}$ is a predicate that specifies the states where the system is operational, doing something useful, $\mathcal{P}_{\mathrm{init}}$ specifies the initial states.

Unconditional instantaneous availability at step k $A_{\mathcal{U}}(k)$ is defined as the probability $Pr(s_k \models \mathcal{P}_{\mathrm{up}} \mid s_0 \models \mathrm{true})$.

Unconditional instantaneous availability is the probability that the system is in "up state" irrespective of the initial state. Generic instantaneous availability is the probability that the system is in "up state" provided it was started in some specific set of states. As k approaches ∞ – provided the limit exists –

instantaneous, unconditional, and generic instantaneous availability are all equal to limiting availability.

The above definitions can be readily used in the context of silent self-stabilizing systems by assigning $\mathcal{P}_{\text{up}} = \mathcal{P}_{\mathcal{S}}$, where $\mathcal{P}_{\mathcal{S}}$ is the safety predicate of the system. Hence, unconditional instantaneous availability of a silent self-stabilizing system is the probability that the system satisfies its safety predicate at an instant k irrespective of its starting state. Generic instantaneous availability of a silent self-stabilizing system is the probability of satisfying the safety predicate provided it started in any initial state characterized by predicate $\mathcal{P}_{\text{init}}$.

Mean time to repair (MTTR) of a self-stabilizing system is the average time (measured in the number of computation steps) taken by a self-stabilizing system to reach a state which satisfies the safety predicate $\mathcal{P}_{\mathcal{S}}$. The average is taken over all the executions which start in states not satisfying the safety predicate $\mathcal{P}_{\mathcal{S}}$. As mentioned earlier, a system has "recovered" from a burst of transient faults when it reaches a safe state. It is also interesting to note that the MTTR mirrors the average case behavior under a given implementation scenario unlike bounds on convergence time that are furnished as part of convergence proofs of self-stabilizing algorithms.

Mean time to failure (MTTF) of a self-stabilizing system is the average time (again measured in the number of computation steps) before a system reaches an unsafe state provided it started in a safe state. This definition may appear trivial for a self-stabilizing system as the notion of MTTF is void given the closure property of self-stabilizing systems. However, under relaxed fault assumptions, the closure is not guaranteed because transient faults may "throw" the system out of the safe states once it has stabilized. Thus, MTTF may well be finite.

There is an interplay between MTTF and MTTR of a self-stabilizing system since its limiting availability also agrees with MTTF/(MTTR + MTTF) [9]. That is, a particular value of MTTF is an environment property over which a system designer has often no control, but the value of MTTR, in the absence of on-going faults, is an intrinsic property of a given implementation of a self-stabilizing algorithm alongwith the scheduler used (synonymously referred to as self-stabilizing *system*). One can modify the self-stabilizing system leading to a possible decrease in average convergence time. The above expression gives a compositional way to fine tune the limiting availability by modifying the MTTR value of a self-stabilizing system despite a possible inability to influence the value of MTTF.

3.2 Model for Dependability Evaluation

The modeling of a self-stabilizing system for performance evaluation is the first step of the toolchain. We assume that the self-stabilizing system consists of a number of concurrent components which run in parallel. These components cooperate to bring the system to a stable condition from any starting state. Furthermore, we assume that at any time a fault may occur which brings the system to an arbitrary state

Guarded command language. We can describe a self-stabilizing system using a *guarded command language* (GCL) which is essentially the language used by the probabilistic model checker PRISM [10].

The model of a component consists of a finite set of variables describing the state of the component, initial valuations for the variables and a finite set of guarded commands describing the behavior (state change) of the component. Each guarded command has the form

```
[label] guard -> prob1 : update1 + ... + probn : updaten
```

Intuitively, if the guard (a Boolean expression over the set of variables) is satisfied, then the command can be executed. One branch of the command is selected probabilistically and the variables are updated accordingly. Deterministic behaviour may be modeled by specifying only a single branch. The commands in the model may be labeled. Figure 1 gives a sketch of a self-stabilizing BFS algorithm of [8] with three components representing a process each. Fault inducing actions are embedded in every component. There are a number of important properties inherent in the model in Figure 1. First, at every step, it is open whether a fault step or a computational step occurs. If a computational step occurs, it is also unclear which component executes a command. Finally, in the case of a fault step, it is unclear which fault occurs, i.e. what the resulting state of the system will be. The model in Figure 1 is thus *non-deterministic* since it does not specify how these choices must be resolved.

Schedulers. To resolve the non-determinism in the model, and thus to arrive at a uniquely defined stochastic process, one usually employs *schedulers*. In essence, a scheduler is an abstract entity resolving the non-determinism among the possible choice options at each time step. A set of schedulers is called a *scheduler class*. For a given scheduler class, one them aims at deriving worst-case and best-case results for the metric considered, obtained by ranging over all stochastic processes induced by the individual schedules in the class. This computation is performed by the probabilistic model checking machinery.

Schedulers can be characterized in many ways, based on the power they have: A scheduler may make decisions based on only the present state (*memoryless* scheduler) or instead based on the entire past history of states visited (*history-dependent* scheduler). A scheduler may be *randomized* or simply be *deterministic*. A randomized scheduler may use probabilities to decide between choice options, while deterministic ones may not. For instance, we can consider the class of randomized schedulers that, when a fault step occurs, chooses the particular fault randomly with a uniform distribution. When adding this assumption to the GCL specification of the fault model, the resulting system model becomes partially probabilistic as shown in Figure 2 for the root module. It is still non-deterministic with respect to the question whether a fault step occurs, or which component performs a step. Here, we encoded the probabilistic effect of the schedulers considered inside the GCL specification, while the remaining non-determinism is left to the background machinery. It would also be possible to specify a choice according to a probability distribution that is obtained using information collected from the history of states visited (*history-dependent* scheduler), or according to a distribution gathered from statistics about faults occuring in real systems.

```
module root
  variable x02,x01 : int ...;
  [stepRoot] true -> 1: x01’ = 0 & x02’=0
  [faultRoot1] true -> 1: x01’ = 0 & x02’=1
      ...
  [faultRootn] true -> 1: x01’ = 2 & x02’=2 ;
 endmodule

module proc1
  variable x10,x12,dis1 : int ...;
  [stepProc1] true -> 1: (dis1’=
        min( min(dis1,x01,x21)+1,N))
       & (x10’ = ...) & (x12’=...) ;
  [faultProc11] true -> 1: (dis1’=0)
       &(x10’=0)&(x12’=1)
                        ...
  [faultProc1n] true -> 1:(dis1’=2)
       &(x10’=2)&(x12’=2);
endmodule

module proc2
  ...
endmodule
```

Fig. 1. Non-deterministic self-stabilizing BFS algorithm with faults.

Markov decision processes. The formal semantics of a GCL model is a *Markov decision process* (MDP). A MDP is a tuple $\mathscr{D} = \{\mathtt{S}, \mathtt{A}, P\}$ where $\mathtt{S}$ is the set of states, $\mathtt{A}$ is the set of possible actions, and $P \subseteq \mathtt{S} \times \mathtt{A} \times \mathit{Dist}(\mathtt{S})$ is the transition relation that gives for a state and an action the resulting probability distribution that determines the next state. In the literature, MDPs are often considered equipped with a reward structure, which is not needed in the scope of this paper.

Intuitively, we can derive a MDP from a GCL model in the following way. The set of states of the MDP is the set of all possible valuations of the variables in the GCL model. The set of actions is the set of labels encountered in the GCL model. For each state we find a set of commands for which the guard is satisfied. Each such command then gives us an entry in the transition relation where the action is given by the label associated with the command and the resulting distribution for the next state is determined by the distribution over the updates in the GCL description. In Figure 3 (left), we see an example of a MDP state with its outgoing transitions for our example model from Figure 1 (where the choice of fault is determined probabilistically as in Figure 2). We see that in every state either a fault may occur, after which the resulting state is chosen probabilistically or a computational step may occur. The choice between faults or different computational steps is still non-deterministic.

```
module root
  variable x02,x01 : int ...;
  [stepRoot] true -> 1: x01’ = 0 & x02’=0
  [faultRoot] true -> 1/n: x01’ = 0 & x02’=1 +
                         ...
                       1/n: x01’ = 2 & x02’=2;
 endmodule
```

Fig. 2. Root module with a randomized scheduler for n distinct faults.

Markov chain. When we consider a specific scheduler that resolves all non-deterministic choices either deterministically or probabilistically, we find a model whose semantics is a particular kind of MDP, namely that has for each state s exactly one transition (s, a, μ) in the transition relation P. If we further disregard the actions of the transitions, we arrive at a model which can be interpreted as a *Markov chain.* We define a Markov chain as a tuple $\mathscr{D} = \{\mathtt{S}, P\}$ where $\mathtt{S}$ is the set of states and $P \subseteq \mathtt{S} \rightarrow \mathit{Dist}(\mathtt{S})$ is the transition relation that gives for a state the probability distribution that determines the next state. A Markov chain is a stochastic process which is amenable to analysis.

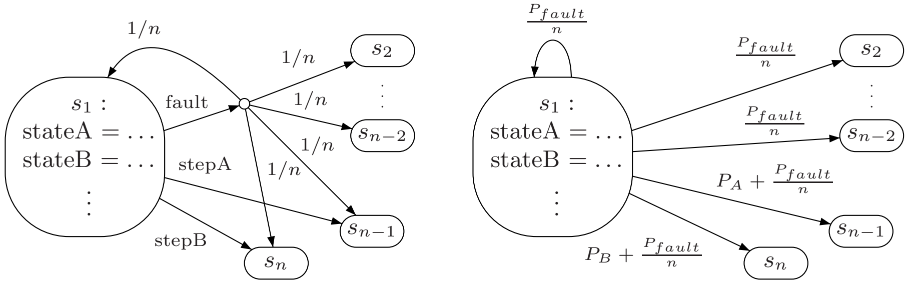


Fig. 3. Segment of a self-stabilizing system modeled as an MDP (left) or a Markov Chain (right)

For our example, we can find a Markov chain model if we assume a scheduler that chooses probabilistically whether a fault occurs, which component takes a step in case of normal computation and which fault occurs in case of a fault step. Figure 4 shows the probabilistic model for the root module and Figure 3 (right) shows part of the resulting model, where P_A, P_B and P_{fault} denote the probabilities that, respectively, component A takes a step, component B takes a step, or a fault occurs.

```
module root
  variable x02,x01 : int ...;
  [stepRoot] true -> STEP_PROB: x01' = 0 & x02'=0+
                    (1-STEP_PROB)/n: x01' = 0 & x02'=1+
                    ...
                    (1-STEP_PROB)/n: x01' = 2 & x02'=2;
 endmodule
```

Fig. 4. Root module modeled to have a fully randomized scheduler.

Choosing a scheduler class. Scheduler classes form a hierarchy, induced by set inclusion. For MDPs, the most general class is the class of history-dependent randomized schedulers. Deterministic schedulers can be considered as specific randomized schedulers that schedule with probability 1 only, and memoryless schedulers can be considered as history-dependent schedulers that ignore the history apart from the present state.

In the example discussed above (Figure 2 and Figure 4), we have sketched how a scheduler class can be shrinked by adding assumptions about a particular probabilistic behaviour. We distinguish two different strategies of doing so: *Restricted resolution* refers to scheduler classes where some non-determinstic options are pruned away. In *partially probabilistic resolution* some of the choices are left non-deterministic, while others are randomized (as in Figure 3, left). A *fully randomized* scheduler class contains a single scheduler only, resolves all non-determinism probabilistically. Recall that deterministic schedulers are specific randomized schedulers. Figure 5 provides an overview of the different resolution strategies.

Choosing a class of schedulers to perform analysis on is not trivial. If we choose too large a class, probability estimations can become so broad as to be unusable (e.g. the model checker may conclude that a particular probability measure lies somewhere between 0 and 1). Choosing a smaller class of schedulers results in tighter bounds for our probability measures. However, choosing a small scheduler class requires very precise information about the occurrence of faults and the scheduling of processes. Furthermore, such analysis would only inform us about one very particular case. A more general result is usually desired, that takes into account different fault models or scheduling schemes. More advanced scheduler classes are also possible. For the scheduling of n processes, we allow only those schedules where each process performs a computational step at least every k steps. This is akin to assuming that the fastest process is at most twice as fast as the slowest one. While such assumptions are interesting to investigate, they also make analysis more difficult. To implement such a *k-bounded* scheduler, it is required to track the last computational step of every process. Though, the size of the state space does not scale well for large n and k.

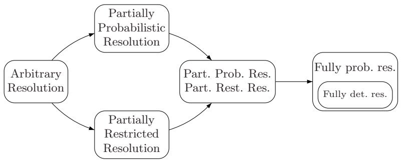

Partially Probabilistic Resolution
Arbitrary Resolution
Part. Prob. Res. Part. Rest. Res.
Fully prob. res.
Fully det. res.
Partially Restricted Resolution

Fig. 5. Overview of different resolution strategies.

Model checking. A model checker, such as PRISM [10] may be employed to answer reachability questions for a given MDP model. The general form of such a property is $P_{<p}\left[A \cup^{\leq k} B\right]$ which checks whether the probability that a state with property B is reached within k steps via a path consisting of states in which A holds only, is smaller than p. In this way instantaneous availability properties can be checked. If the property does not hold, a *counterexample* is generated.

We next explain the methods to re-engineer a self-stabilizing system based on a counterexample provided by a probabilistic model checker.

4 Dependability Engineering for Self-Stabilizing Systems

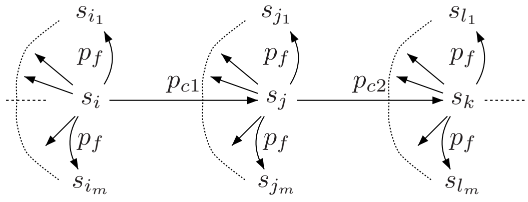
In order to meet the quality of service requirements, the counterexample returned by the model checker can be used to optimize the system. An important distinction between counterexample generation of qualitative model checking versus quantitative model checking is the fact that quantitative model checking returns a *set of paths* as counterexample. This distinction needs to be taken into account while devising a method for exploiting the counterexample. We explain a heuristics-based method to modify a system given such a set of paths. The self-stabilizing BFS spanning tree algorithm implemented on a three-process graph under a fully randomized scheduler is used as an illustrative example. We used the stochastic bounded model checker `sbmc` [11] alongwith PRISM to generate counterexamples. Please note that this particular method applies beneficently only to scenarios where faults follow a uniform probability distribution over the system states.

4.1 Counterexample Structure

An understanding of the structure of the elements of the set of paths returned as counterexample is important to devise a method to modify the system. In the scope of this section, we are interested in achieving a specific unconditional instantaneous availability $A_{\mathcal{U}}(k)$ which is basically the step-bounded reachability probability of a legal state. However, the tool used to generate the counterexample can only generate counterexamples for queries that contain an upper bound on the probability of reaching a set of certain states. Therefore, a reformulated query is presented to the model checker. Instead of asking queries of the form "Is the probability of reaching a legal state within k steps greater than p?," i.e. $P_{>p}\left[\text{true} \cup^{\leq k} \text{legal}\right]$ the following query is given to the model checker: $P_{\leq(1-p)}\left[\neg\text{legal}\,\mathcal{W}^{\leq k}\,\text{false}\right]$. The reformulated query ascertains whether the probability of *reaching non-legal* states with in k steps is *less than* $1-p$. The probability p used in the queries is equal to the desired value of $A_{\mathcal{U}}(k)$, namely unconditional instantaneous availability at step k.

In case the probability of reaching non-legal states is larger than the desired threshold value, the probabilistic model checker returns a set of paths of length k. This set consists of k-length paths such that all the states in the path are non-legal states. The probability of these paths is larger than the threshold specified in the query. This set of paths constitutes a counterexample because the paths as a whole violate the property being model-checked.

In order to devise a system optimization method we "dissect" a generic path – annotated with transition probabilities – of length 2 (shown below) for a system with a uniform fault probability distribution.



p_{c1} and p_{c2} are probabilities of state transitions due to a computation step whereas p_f is the probability of a fault step. Note that due to the uniform fault probability distribution there is a pair of fault transitions between each pair of system states. If the above path is seen in contrast with a fault-free computation of length 2, one can identify the reason for the loss of probability mass. Consider a path that reaches state s_k from state s_i in two steps.

$$\cdots\cdots\quad s_i \xrightarrow{1} s_j \xrightarrow{1} s_k \cdots\cdots$$

Such a path can be extracted from a MDP-based model by choosing a specific scheduler. It results in a fully deterministic model because of the absence of fault steps, thereby leaving the model devoid of any stochastic behavior. The probability associated with each of the two transitions is 1 and therefore the probability of the path is 1 as well [12]. However, the addition of fault steps to the model reduces probabilities associated with computation steps and thus, reduces the probability of the path. In the light of this discussion, we next outline a method to modify the system in order to achieve a desired value of $A_{\mathcal{U}}(k)$.

4.2 Counterexample-guided System Re-engineering

We consider the set of paths of length k returned by the probabilistic model checker. In Step 1, we remove the extraneous paths from the counterexample. In Step 2, we add and remove certain transitions to increase $A_{\mathcal{U}}(k)$.

Step 1. As explained above, a counterexample consists of all those paths of length k whose probability in total is greater than the threshold value. This set also consists of those k-length paths where some of the transitions are fault steps. The number of possible paths grows combinatorially as k increases because the uniform fault model adds transitions between every pair of states. For example, as there are (fault) transitions between each pair of states, the probabilistic model checker can potentially return all the transitions of the Markov chain for $k = 1$. Hence, the problem becomes intractable even for small values of k. Therefore, such paths are removed from the set of paths. The resultant set of paths consists of only those k-length paths where all the transitions are due to computation steps. The self-stabilizing BFS spanning tree algorithm was model checked to verify whether the probability of reaching the legal state within three steps is higher than 0.65. The example system did not satisfy the property and thus, the conjunction of PRISM and `sbmc` returned a set of paths as counterexample. This set contains 190928 paths in total out of which a large number of paths consist of fault steps only. An instance of such a path is shown below.

$$\langle 2,2,1,2,2,1,1,2\rangle \rightarrow \langle 2,2,1,2,2,0,0,0\rangle \rightarrow \langle 2,2,0,0,0,0,0,0\rangle \rightarrow \langle 2,2,0,0,0,0,0,1\rangle$$

A state in the path is represented as a vector $s_i = \langle x_{01}, x_{02}, x_{12}, x_{10}, \mathrm{dis}_1, x_{20}, x_{21}, \mathrm{dis}_2\rangle$ where x_{ij} is the communication register owned by process proc_i and dis_i is the local variable of proc_i. The removal of such extraneous paths lead to a set of 27 paths.

Step 2. The probability of a path without a loop is the product of the individual transition probabilities. Due to the presence of fault steps and associated transition probabilities, one cannot increase the probability measure of the path without decreasing the path length. Consider a path

$$s_i \xrightarrow{p_i} s_j \xrightarrow{p_j} s_k \xrightarrow{p_k} s_l$$

and the modified path obtained by 1) adding a direct transition between states s_i and s_l and 2) disabling the transition between states s_i and s_j.

$$\begin{tikzcd}
s_i \arrow[rrr, bend left, "p_i"] & s_j \arrow[r, "p_j"] & s_k \arrow[r, "p_k"] & s_l
\end{tikzcd}$$

The addition of a direct transition to state s_l and thereby the reduction of the path length leads to an increase in probability of reaching state s_l from state s_i. The method thus strives to increase the value of $A_{\mathcal{U}}(k)$ by reducing the length of the paths. As we have no control over the occurrence of fault steps, such transitions can neither be removed nor the probabilities associated with these transitions be altered. Thus, in essence, we increase the number of paths with length less than k and decrease k-length paths to the legal state.

The paths in the counterexample are arranged in decreasing order of probability. The following procedure is applied to all the paths starting with the most probable path. We begin with the first state s_0 of a path. In order to ensure that a transition is feasible between s_i and s_j, we determine the variables whose valuations need to be changed to reach state s_j from state s_i. A transition is deemed feasible for addition to a system if the variable valuations can be changed in a single computation step under a specific sequential randomized scheduler. If a transition from state s_0 to the legal state s_l is deemed feasible, then a guarded command to effect that state transition is added to the system. In case such a direct transition is not feasible, then transitions are added to modify the local states of the processes to decrease the convergence time. This method can be iterated over the initial states of the paths returned till the desired threshold is achieved or all the returned paths are used up.

Addition of such transitions, however, requires some knowledge of the algorithm under consideration. For instance, a state transition to s_l that leads to a maximal decrease in convergence time might require change of variables belonging to more than one process. Such a transition is not feasible if an algorithm is implemented with a sequential scheduler. Infeasibility of a direct state transition to s_l may also result from the lack of "global knowledge." Let $s_i \longrightarrow s_l$ be the transition that leads to a maximal decrease in convergence time and let proc_x be the process whose local state must be changed to effect the aforementioned state transition. Process proc_x, therefore, needs a guarded command that changes its *local state* if system is in a *specific global state*. However, process proc_x cannot determine local states of *all* processes in a system unless the communication topology of system is a completely connected graph. Transition $s_i \longrightarrow s_l$, in this, is infeasible for communication topologies which are not completely connected. This is, however, an extremal case because usually processes – instead of global knowledge – require knowledge of their extended "neighborhood."

We applied the above procedure on the example system by analyzing the resultant set of paths after removing paths with fault steps. The state $s_b = \langle 2, 2, 1, 2, 2, 1, 1, 2 \rangle$ was the most probable *illegal* state. The paths having this state as the initial state were inspected more closely; a direct transition to the legal state $\langle 0, 0, 1, 1, 1, 1, 1, 1 \rangle$ was not feasible because it required changes in variable valuations in all three processes in a *single* step. However, a transition could be added to correct the local state of the non-root processes so that if the system is in state s_b, then the (activated) process corrects its local state. The communication topology of the example system allows each process to access the local states of all the process. Thus, guarded commands of the form

```
[stepstateB] state=stateB -> state'= correctstate
```

were added to the processes proc_1 and proc_2. The modification of the system led to an increase in probability (of reaching the legal state from state s_b) from 0.072 to 0.216.

The method described above can be used to modify the system for a given scheduler under a fault model with ongoing faults. However, the very fact that the scheduler is fixed limits the alternatives to modify the system. For instance, many transitions which could have potentially increased $A_{\mathcal{U}}(k)$ were rendered infeasible for the example system. This, in turn, can lead to an insufficient increase in $A_{\mathcal{U}}(k)$ or a rather large number of iterations to achieve the threshold value of $A_{\mathcal{U}}(k)$. The problem can be circumvented if one has leeway to fine-tune the randomized scheduler or modify the communication topology.

4.3 Randomized Scheduler Optimization

The probabilities with which individual processes are activated in each step by a scheduler affects the convergence time and thus the unconditional instantaneous availability of the system. However, a counterexample can be exploited to identify the processes whose activation probabilities need to be modified. For instance, consider a path returned by the conjunction of PRISM and `sbmc`:

$$\langle 2,2,1,2,2,1,1,2\rangle \rightarrow \langle 0,0,1,2,2,1,1,2\rangle \rightarrow \langle 0,0,1,1,1,1,1,2\rangle \rightarrow \langle 0,0,1,1,1,1,1,2\rangle$$

In the second last state of the path, activation of the root process does not bring any state change and thus leads only to an increase in convergence time. Hence, if the probability of activating a non-root process in the scope of the example algorithm is increased, then the probability associated with such suboptimal paths can be decreased. We varied the probability of activating the root process in the example system to see the effect on $A_{\mathcal{U}}(k)$. As Figure 6 shows, unconditional instantaneous availability increases as the probability of activating the root is decreased. This is because one write operation of the root process alone corrects its local state; further activations are time-consuming only. But once the root process has performed a computation step, any activation of a non-root process corrects its local state. The paths in the counterexample can be analyzed in order to identify those processes whose activations lead to void transitions. The respective process activation probabilities of the scheduler can then be fine-tuned to increase $A_{\mathcal{U}}(k)$.

4.4 Abstraction Schemes for Silent Self-Stabilization

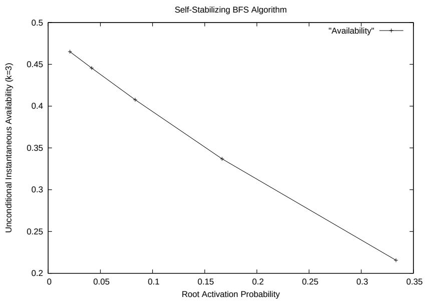


Fig. 6. Variation of unconditional instantaneous availability

Probabilistic model checking of self-stabilizing systems suffers from the state space explosion problem even for a small number of processes. This is due to the fact that the set of initial states of a self-stabilizing system is equal to the entire state space. As we intend to quantify the dependability of a self-stabilizing algorithm in an implementation scenario, we may be confronted with systems having a large number of processes. This necessitates a method to reduce the size of the model before giving it to the model checker. Often, data abstraction is used to reduce the size of large systems while preserving the property under investigation [13]. We next evaluate existing abstraction schemes and identify a suitable abstraction scheme for silent self-stabilizing systems.

Data abstraction constructs an abstract system by defining a finite set of abstract variables and a set of expressions which maps variables of the concrete system to the domain of the abstract variables. A form of data abstraction is predicate abstraction where a set of Boolean predicates is used to partition the concrete system state space [14]. Doing so results in an abstract system whose states are tuples of boolean variables. However, predicate abstraction can only be used to verify safety properties as it does not preserve liveness properties [15]. Since convergence is a liveness property, predicate abstraction cannot be used to derive smaller models of self-stabilizing systems.[4]

Ranking abstraction overcomes the deficiency of predicate abstraction by adding a *non-constraining progress monitor* to a system [15]. A progress monitor keeps track of the execution of the system with the help of a ranking function. The resulting augmented system can then be abstracted using predicate abstraction.

An important step in abstracting a system using a ranking abstraction is the identification of a so called *ranking function core*. This need not be a single ranking function – parts of it suffice to begin the verification of a liveness property. The fact that we are trying to evaluate a silent self-stabilizing system makes the search of a ranking function core easier. The proof of the convergence property of a self-stabilizing system is drawn using either a ranking function [17], for instance a Lyapunov function [18], or some other form of well-foundedness argument [19]. Thus, one already has an explicit ranking function (core) and, if that is not the case, then the ranking function core can be "culled" from the proof of a silent self-stabilizing system. Further, we can derive an abstracted self-stabilizing system with the help of usual predicate abstraction techniques once the system has been augmented with a ranking function.

5 Conclusion and Future Work

We defined a set of metrics, namely unconditional instantaneous availability, generic instantaneous availability, MTTF, and MTTR, to quantify the dependability of self-stabilizing algorithms. These metrics can also be used to compare different self-stabilizing solutions to a problem. We also showed how to model a self-stabilizing system as a MDP or as a MC to derive these metrics. Further, heuristic-based methods were presented to exploit counterexamples of probabilistic model checking and to re-engineer silent self-stabilizing systems.

There are still open challenges with respect to dependability engineering of self-stabilizing systems. An abstraction scheme suitable for non-silent self-stabilizing algorithms is required to make their dependability analysis scalable.

[4] However, for the properties considered here, which are step-bounded properties, this reasoning does not apply. In fact, we experimented with the predicate-abstraction-based probabilistic model checker PASS [16] that also supports automatic refinement. This was not successful because PASS seemingly was unable to handle the many distinct guards appearing in the initial state abstraction.

As discussed, there are multiple ways to refine a system which in turn leads to the challenge of finding the most viable alternative. We would also like to increase the tool support for dependability engineering of self-stabilizing systems. We believe that the identification of optimal schedulers and the determination of feasible transitions are the most promising candidates for solving the problem.

References

1. Dhama, A., Theel, O., Warns, T.: Reliability and Availability Analysis of Self-Stabilizing Systems. In: Stabilization, Safety, and Security of Distributed Systems. Volume 4280 of LNCS., Springer (2006) 244–261
2. Arora, A., Kulkarni, S.S.: Component Based Design of Multitolerant Systems. IEEE Trans. Software Eng. **24**(1) (1998) 63–78
3. Arora, A., Kulkarni, S.S.: Designing Masking Fault-Tolerance via Nonmasking Fault-Tolerance. IEEE Trans. Software Eng. **24**(6) (1998) 435–450
4. Ghosh, S., Gupta, A., Herman, T., Pemmaraju, S.V.: Fault-Containing Self-Stabilizing Algorithms. In: PODC. (1996) 45–54
5. Ghosh, S., Pemmaraju, S.V.: Tradeoffs in fault-containing self-stabilization. In: WSS. (1997) 157–169
6. Beauquier, J., Gradinariu, M., Johnen, C.: Randomized Self-Stabilizing and Space Optimal Leader Election under Arbitrary Scheduler on Rings. Distributed Computing **20**(1) (2007) 75–93
7. Aljazzar, H., Leue, S.: Debugging of Dependability Models Using Interactive Visualization of Counterexamples. In: QEST, IEEE Computer Society (2008) 189–198
8. Dolev, S., Israeli, A., Moran, S.: Self-Stabilization of Dynamic Systems Assuming Only Read/Write Atomicity. Distributed Computing **7**(1) (1993) 3–16
9. Trivedi, K.S.: Probability and Statistics with Reliability, Queuing, and Computer Science Applications. John Wiley and Sons (2001)
10. Hinton, A., Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM: A Tool for Automatic Verification of Probabilistic Systems. In: TACAS. Volume 3920 of LNCS. (2006) 441–444
11. Wimmer, R., Braitling, B., Becker, B.: Counterexample Generation for Discrete-Time Markov Chains Using Bounded Model Checking. In: VMCAI. Volume 5403 of LNCS. (2009) 366–380
12. Baier, C., Katoen, J.P.: Principles of Model Checking. The MIT Press (2008)
13. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: POPL. (1977) 238–252
14. Graf, S., Saïdi, H.: Construction of Abstract State Graphs with PVS. In: CAV. Volume 1254 of LNCS. (1997) 72–83
15. Balaban, I., Pnueli, A., Zuck, L.: Modular Ranking Abstraction. Int. J. Found. Comput. Sci. **18**(1) (2007) 5–44
16. Hermanns, H., Wachter, B., Zhang, L.: Probabilistic CEGAR. In: CAV. Volume 5123 of LNCS. (2008) 162–175
17. Kessels, J.L.W.: An Exercise in Proving Self-Stabilization with a Variant Function. Inf. Process. Lett. **29**(1) (1988) 39–42
18. Oehlerking, J., Dhama, A., Theel, O.: Towards Automatic Convergence Verification of Self-Stabilizing Algorithms. In: Self-Stabilizing Systems. Volume 3764 of LNCS., Springer (2005) 198–213
19. Gouda, M.G., Multari, N.J.: Stabilizing Communication Protocols. IEEE Trans. Computers **40**(4) (1991) 448–458