Ensuring Average Recovery with Adversarial Scheduler

Jingshu Chen[1], Mohammad Roohitavaf[2], and Sandeep S. Kulkarni[3]

1 **Michigan State University, East Lansing, USA**
`chenji15@cse.msu.edu`
2 **Michigan State University, East Lansing, USA**
`roohitav@cse.msu.edu`
3 **Michigan State University, East Lansing, USA**
`sandeep@cse.msu.edu`

Abstract

In this paper, we focus on revising a given program so that the average recovery time in the presence of an adversarial scheduler is bounded by a given threshold λ. Specifically, we consider the scenario where the fault (or other unexpected action) perturbs the program to a state that is outside its set of legitimate states. Starting from this state, the program executes its actions/transitions to recover to legitimate states. However, the adversarial scheduler can force the program to reach one illegitimate state that requires a longer recovery time.

To ensure that the average recovery time is less than λ, we need to remove certain transitions/behaviors. We show that achieving this average response time while removing minimum transitions is NP-hard. In other words, there is a tradeoff between the time taken to synthesize the program and the transitions preserved to reduce the average convergence time. We present six different heuristics and evaluate this tradeoff with case studies. Finally, we note that the average convergence time considered here requires formalization of hyperproperties. Hence, this work also demonstrates feasibility of adding (certain) hyperproperties to an existing program.

1998 ACM Subject Classification F.4 Mathematical logic and formal languages

Keywords and phrases Average Recovery Time, Hyper-liveness, Program Repair

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.23

1 Introduction

The problem of model repair focuses on revising a given program so that it satisfies new properties while preserving its existing properties. Such model repair is highly desirable when program requirements change (especially when new requirements are added) or bugs are identified in an existing program. The problem of model repair has been studied in the context of revising a program to add safety properties (e.g., to ensure that program never reaches an undesired state), liveness properties (e.g., if the program reaches a state where predicate X is true, then it will reach a state where some predicate Y is true), fault-tolerance properties (e.g., ensuring that safety and/or liveness is preserved in the presence of faults), and timing constraints [7, 12, 19, 6, 14, 18, 20, 22, 16, 13, 15].

All of the properties considered in [7, 12, 19, 6, 14, 18, 20, 22, 16, 13, 15] are expressed in terms of the framework in [2] that shows that any specification can be decomposed into a safety specification and a liveness specification. An important characteristic of the properties in [2] is that *Whether a program computation satisfies the specification is independent of other computations produced by that program.* So, if we consider a safety requirement of the

form: *the value of a variable x is never* 0 then we can evaluate a given program computation to decide whether x ever reaches 0. If yes, it implies that the computation violates the specification. It does not depend upon all other computations that the program can produce. Likewise, if the specification requires that: *if the program reaches a state where X is true then it will reach a state where predicate Y is true* and the given program computation satisfies this requirement then this observation remains true irrespective of other computations produced by that program. This implies that if we want to verify whether a given program is correct, we can evaluate each of its computations separately to determine if it satisfies the specification. If all of them satisfy the specification, we can identify that the program satisfies the specification. Otherwise, the program violates the specification.

Certain requirements however do not satisfy this constraint. Examples of this include some security properties (e.g., information flow [4], noninterference [17]) and some performance properties (e.g., average response time). To illustrate this, consider the requirement *if the program reaches a state where X is true then the average number of transitions required to reach a state in Y is* 5 *or less.* If a program has a computation where the response time (i.e., the number of steps required from X to Y) is 6 that does not imply that the specification is violated. In particular, if the program has several other computations with response time of 4 or less, then including the computation with response time of 6 is perfectly acceptable. In other words, properties such as average response time require analysis of all program computations simultaneously to decide whether program satisfies the specification or not. In [9], authors have introduced the notion of hyperproperties to characterize such requirements. They have also shown that hyperproperties are strictly more general than the (simpler) properties identified in [2].

Existing work in [7, 12, 19, 6, 14, 18, 20, 22, 16, 13, 15] is designed for addition of properties from [2] and does not address performing model repair to add hyperproperties. With this motivation, in this paper, we focus on developing complexity results and algorithms for the addition of one type of hyperproperty, namely average response time.

To motivate the requirement considered in this paper, consider a typical requirement in the context of fault-tolerant and/or stabilizing programs: In these programs, it is required that after faults stop occurring, the program recovers to a legitimate state. An important attribute for this recovery is the time taken for it. There are several ways –worst case, average case etc – to compute the recovery time. The recovery time is also affected by assumptions made about any non-deterministic choices the program may face. In our work, we consider the following approach to compute the average time for recovery (convergence). We focus on the case where the fault perturbs the program to an illegitimate state and the program recovers from there to a legitimate state. Since faults are typically random in nature, there is a probability distribution associated with illegitimate states. (For sake of simplicity, in our case studies, we assume that all illegitimate states are reached equally likely. However, our approach can handle any probability distribution.) Subsequently, during program recovery, there is often a non-deterministic choice given to the program. When faced with such a choice, we consider the case where we use adversarial scheduler that attempts to force the program down on a path that increases the convergence time. This enables one to account for an implementation where the designer considers the non-deterministic choices in any arbitrary order.

Based on the choices considered above, we focus on ensuring that the average recovery time in the presence of an adversarial scheduler (denoted as average convergence time for brevity) is less than λ_e. Furthermore, during this repair, we want to preserve existing safety and liveness properties. Hence, during repair, we focus on removing existing behaviors so that the average convergence time is less than λ_e.

Contributions of the paper. The main contributions of the paper are as follows:

- Since repair for satisfying the average response time constraint requires removal of behaviors/transitions that are responsible for increasing the convergence time, we evaluate the complexity of revision for satisfying the average convergence time requirement while removing a minimum number of transitions. We show that this problem is NP-hard.
- We also show that if we omit the requirement about removing only a minimum transitions then the problem can be solved in P. We present an approach (denoted as SCP) to evaluate this approach. While it is very fast, we find that it removes a large number of transitions (in some cases $> 99\%$).
- To overcome the limitations of SCP and the NP-hardness of maximizing the number of transitions, we present five additional heuristics namely ELP, KBP, RIA, RIAD and SSP. Of these, RIA and RIAD take into account possible distribution constraints that prevent a process from reading or writing all program variables. We show that these approaches provide a tradeoff between the time required to find the desired program and the number of transitions that are removed to guarantee average convergence time.

Organization of the paper. The rest of the paper is organized as follows: In Section 2, we define the notion of programs and average convergence time. In Section 3, we define the problem of adding average convergence time. The complexity analysis of this problem is discussed in Section 4. In Section 5, we present our six approaches that identify the tradeoff between the time for repair and the non-determinism preserved in the repaired program. We discuss our experimental results for two case studies in Section 6. Finally, we discuss related work in Section 7 and conclude in Section 8.

2 Preliminaries

In this section, we formally introduce the notions of program and other related definitions. Our definitions are based on those given by Arora and Gouda [3].

▶ **Definition 1** (Program). A program $\mathcal{P}$ is a tuple $\langle S_{\mathcal{P}}, \delta_{\mathcal{P}} \rangle$, where:

- $S_{\mathcal{P}}$ is the (finite) state space, i.e., the set of all states of $\mathcal{P}$.
- $\delta_{\mathcal{P}}$ is a set of transitions. Specifically, $\delta_{\mathcal{P}} \subseteq S_{\mathcal{P}} \times S_{\mathcal{P}}$.

For simplicity of presentation, we assume that there is at least one outgoing transition of $\mathcal{P}$ from each state. If there is no transition from state s, we can simply add transition (s, s). We would like to note that this is not a restriction in any sense. However, it avoids having to consider terminating states in subsequent definitions.

▶ **Definition 2** (State Predicate). Given a program $\mathcal{P}$ ($\langle S_{\mathcal{P}}, \delta_{\mathcal{P}} \rangle$), a state predicate $\mathcal{P}$ is a subset of $S_{\mathcal{P}}$.

▶ **Definition 3** (Computation). A computation of $\mathcal{P}$ is an infinite sequence of states, $\rho = \langle s_0, s_1, \ldots \rangle$, where

- $\forall j, j > 0$: $(s_{j-1}, s_j) \in \delta_{\mathcal{P}}$.

▶ **Definition 4** (Distance of a state predicate in a computation). Let $\rho = \langle s_0, s_1, \ldots \rangle$ be a computation of $\mathcal{P}$. Let S be a state predicate of $\mathcal{P}$. We say that the distance of ρ to S (denoted by $compdist(\mathcal{P}, \rho, S)$) is w iff $\forall j : (j < w) \Rightarrow s_j \notin S$ and $s_w \in S$.

In the above definition, if ρ does not contain a state in S, we say that $compdist(\mathcal{P}, \rho, S) = \infty$. Next, we overload the definition of distance to define the notion of a distance of a state

predicate from a given state, say s. There may be several computations that start from s. Since we focus on an adversarial scheduler, distance of a state s from state predicate S is described by considering the maximum number of steps required from s in some computation of $\mathcal{P}$. In other words, this definition captures the maximum distance from state s to state predicate S.

▶ **Definition 5** (Distance of a state predicate from a state)**.** Distance of state s to a state predicate S in program $\mathcal{P}$, denoted by $statedist(\mathcal{P}, s, S)$, is $max(compdist(\mathcal{P}, \rho, S)|\rho$ is a computation of $\mathcal{P}$ that starts in $s)$.

Using the above definition, we can define the notion of average time to recover from some state predicate T to another state predicate S as follows:

▶ **Definition 6** (Distance between two state predicates)**.** Let S and T be state predicates of $\mathcal{P}$. The average convergence time from T to S in program $\mathcal{P}$, denoted by $predist(\mathcal{P}, T, S)$ is $average(statedist(\mathcal{P}, s, S)|s \in T - S)$.

For sake of simplicity, we define $predist(\mathcal{P}, S, S)$ to be 0.

▶ **Definition 7** (Average convergence time)**.** Let S and T be state predicates of $\mathcal{P}$. Let λ be a real number. We say that T converges to S within λ iff

- $predist(\mathcal{P}, T, S) \leq \lambda$.

Observe that if some computation of $\mathcal{P}$ starts from a state in T and never reaches a state in S then $predist(\mathcal{P}, T, S) \geq \lambda$ from any number λ.

3 Problem Formulation

In this section, we formally state our program repair problem with respect to average convergence time requirements. The goal of this problem is to revise a given program $\mathcal{P}$ to $\mathcal{P}'$ that uses a subset of behaviors of $\mathcal{P}$ to reduce the convergence time to the set of legitimate states. Since $\mathcal{P}'$ only uses a subset of behaviors of $\mathcal{P}$, it follows that if $\mathcal{P}$ satisfied any safety or liveness property (that is described using the framework [2]) then $\mathcal{P}'$ satisfies that property as well.

The input to the repair program consists of program $\mathcal{P}$ with state space $S_{\mathcal{P}}$ and transitions $\delta_{\mathcal{P}}$. It also includes the state predicate denoting the legitimate states, S. Finally, it includes the desired average convergence time λ. The goal of the program is to identify program $\mathcal{P}'$ that uses the behaviors of $\mathcal{P}$, and provides convergence to S with average time λ. Thus, the problem statement is as follows:

▶ **Definition 8** (The Program Repair Problem)**.** Given a program $\mathcal{P} = \langle S_{\mathcal{P}}, \delta_{\mathcal{P}} \rangle$, the set of legitimate states S, and the required average convergence time λ, identify $\mathcal{P}' = \langle S_{\mathcal{P}'}, \delta_{\mathcal{P}'} \rangle$ such that

- $S_{\mathcal{P}'} \subseteq S_{\mathcal{P}}$
- $\delta_{\mathcal{P}'} \subseteq \delta_{\mathcal{P}}$
- $\mathcal{P}'$ converges to S from $S_{\mathcal{P}'}$ within λ', $\lambda' \leq \lambda$.

In order to characterize the complexity of the above problem, we identify a corresponding decision problem. The first attempt to find this decision problem is as follows:

▶ **Definition 9** (The Decision Problem (Attempt 1) (Dec_1).)**.** Given a program $\mathcal{P} = \langle S_{\mathcal{P}}, \delta_{\mathcal{P}} \rangle$, the set of legitimate states S, and the required average convergence time λ: Does there exist a program $\mathcal{P}' = \langle S_{\mathcal{P}'}, \delta_{\mathcal{P}'} \rangle$ that satisfies the requirements specified for the program repair problem in Definition 8.

The decision problem Dec_1 can be trivially answered by setting $S_{\mathcal{P}'}$ to S. In this case, it is straightforward that $\mathcal{P}$ converges to S from S within 0. To avoid such trivial answer, we require that recovery from all states in $S_{\mathcal{P}}$ be preserved. Hence, we require that $S_{\mathcal{P}'} = S_{\mathcal{P}}$. Hence, the second attempt at defining the decision problem is as follows:

▶ **Definition 10** (The Decision Problem (Attempt 2) (Dec_2))**.** Given a program $\mathcal{P} = \langle S_{\mathcal{P}}, \delta_{\mathcal{P}} \rangle$, the set of legitimate states S, and the required average convergence time λ: Does there exist a program $\mathcal{P}' = \langle S_{\mathcal{P}'}, \delta_{\mathcal{P}'} \rangle$, such that $S_{\mathcal{P}'} = S_{\mathcal{P}}$ and $\mathcal{P}'$ satisfies the requirements specified for the program repair problem in Definition 8.

The decision problem Dec_2 can also be solved in P (in the state space of the program) as follows: We start with $\mathcal{P}'$ that contains no transitions in $S_{\mathcal{P}} - S$. Then, from each state in $S_{\mathcal{P}} - S$, we add the shortest path (least recovery in terms of number of transitions) to some state in S. If there are several shortest paths, we can choose any one of them. We argue (in Section 4) that the resulting program guarantees that starting from any state in $S_{\mathcal{P}}$, the program reaches a state in S. Also, the resulting program provides the least average convergence time. Hence, if the average convergence time is larger than λ then it is impossible to find $\mathcal{P}'$ that satisfies the problem statement Dec_2.

In both decision problems Dec_1 and Dec_2, we require that $\delta_{\mathcal{P}'} \subseteq \delta_{\mathcal{P}}$. Requiring $\delta_{\mathcal{P}'} = \delta_{\mathcal{P}}$ is meaningless since it would require $\mathcal{P}$ and $\mathcal{P}'$ to be identical. However, we can focus on finding $\mathcal{P}'$ that preserves the maximum behavior of $\mathcal{P}$. Having $\mathcal{P}'$ with more non-determinism is desirable, as it provides the designer with a maximum choice in terms of implementation. It is also known to increase the ability to add new properties in the future. Thus, we define the decision problem as follows:

▶ **Definition 11** (The Decision Problem (Final) (Dec_3))**.** Given a program $\mathcal{P} = \langle S_{\mathcal{P}}, \delta_{\mathcal{P}} \rangle$, the set of legitimate states S, the required average convergence time λ, **and integer** k: Does there exist a program $\mathcal{P}' = \langle S_{\mathcal{P}'}, \delta_{\mathcal{P}'} \rangle$, such that $S_{\mathcal{P}'} = S_{\mathcal{P}}$, $|\delta_{\mathcal{P}'}| \geq k$, and $\mathcal{P}'$ satisfies the requirements specified for the program repair problem in Definition 8.

In Section 4, we show that Dec_3 is NP-complete in the state space $S_{\mathcal{P}}$. Given that Dec_2 is in P but Dec_3 is NP-complete, it follows that there is a tradeoff between the time required to find $\mathcal{P}'$ and the fraction of transitions/behaviors removed by $\mathcal{P}'$. In particular, it is efficient to find $\mathcal{P}'$ that preserves only a small subset of behaviors. However, it is significantly more complex to design $\mathcal{P}'$ that designs the maximum possible behaviors.

4 Complexity Analysis

In this section, we show that Dec_2 can be solved in polynomial time in the state space of the program, using a straightforward approach, but Dec_3 is NP-complete.

Regarding Dec_2, we construct transitions of $\mathcal{P}_{minpath}$ as follows: For each state, $s \notin S$, we include the transitions corresponding to the path L_s which is the shortest path from s to some state in S. Next, in Lemmas 12 and 13, we show that the resulting program provides the least average convergence time for any program that solves the problem in Definition 8. Hence, if this program does not provide the desired average recovery time then the answer to Dec_2 is false.

▶ **Lemma 12.** *$\mathcal{P}_{minpath}$ guarantees that starting from any state in S_p, the program reaches a state in S.*

▶ **Lemma 13.** *$\mathcal{P}_{minpath}$ provides the least average recovery time for any program that solves the problem in Definition 8.*

▶ **Theorem 14.** *Dec_2 can be solved in polynomial time in the state space of the input program $\mathcal{P}$.*

Regarding Dec_3, we can reduce the problem of adding liveness constraints in [5] to Dec_3. Specifically, in [5], it is shown that the following problem is NP-complete.

Given a program $\mathcal{P}$, two state predicates S and T and a positive integer k, does there exist a $\mathcal{P}'$ such that $S_{\mathcal{P}'} = S_{\mathcal{P}}$, $\delta_{\mathcal{P}'} \subseteq \delta_{\mathcal{P}}$, every computation of $\mathcal{P}'$ that starts in a state predicate T reaches a state in state predicate S and $\delta_{\mathcal{P}'} \geq k$.

Showing that Dec_3 is in NP is straightforward. To reduce the above problem to an instance of Dec_3, we essentially need to set the value of λ, the average convergence time of $\mathcal{P}'$ to $|S_{\mathcal{P}}|$. It is straightforward to observe that if every computation of $\mathcal{P}'$ reaches the state predicate S then the average convergence time is less than $|S_{\mathcal{P}}|$. Thus, we have

▶ **Theorem 15.** *Dec_3 is NP-complete in the state space of the input program $\mathcal{P}$.*

5 Repair Approaches

In this section, we consider the problem of repairing a given program $\mathcal{P}$ to meet the average convergence time λ requirement. As shown in Theorem 15, this problem is NP-hard under the constraint that the revised program must preserve a given number of transitions. By contrast, if we solve this problem minimally (Dec_2) without the above constraint, then the problem can be solved in P (cf. Theorem 14). However, the solution for this approach (discussed in Section 5.1) is likely to include only a small number of transitions in the repaired program. In other words, there is a tradeoff between the time required to design the repaired program and the level of non-determinism (choices) available to that repaired program. Hence, we evaluate this tradeoff with several heuristics. At one extreme, we consider the approach that is expected to be the fastest but provides least non-determinism. This approach is based on the algorithm that solves Dec_2. The other extreme, i.e., the solution with maximum choices, requires exponential time (unless $P = NP$) and, hence, is infeasible. We also consider several intermediate heuristics as well.

We develop the following six approaches. Of these, the first approach, Shortest Convergence Path (SCP), focuses on adding those transitions which lead to the shortest convergence paths. The second approach, Eliminate Longest Path (ELP), focuses on removing offending behaviors that cause an increase in the delay of convergence. The third approach, Keep Best Path (KBP), repairs the given program by only preserving transitions that lead to the shortest convergence paths when several outgoing transitions are available for states.

These first three approaches view the program purely in terms of its transitions. They ignore the structure of the program. In the next two heuristics, we focus on the structure of the program in terms of guarded commands [11]. Specifically, the forth approach, Restrict Individual Actions (RIA), uses the guarded commands of the input program $\mathcal{P}$ and constructs $\mathcal{P}'$ whose guards are a combination of guards involved in $\mathcal{P}$. The fifth approach, Restrict Individual Actions with Distribution Restrictions (RIAD), extends RIA to deal with restrictions imposed by distributed systems. In particular, it restricts the actions whose guards can be combined. This allows one to ensure that the repaired program can be implemented in low atomicity where each process can read or write only a subset of variables. Hence, RIAD provides a mechanism to guarantee average convergence time to distributed systems where each process can read/write only a subset of program variables.

Finally, the sixth approach, Solve Similar Problem (SSP) partitions the problem into two subproblems. Of these, in the first step, we focus on guaranteeing worst case convergence

time with value that is larger than the desired average convergence time. And, in the second step, we apply ELP on the resulting program.

In all these approaches, our algorithm takes as input the program $\mathcal{P}$, its set of legitimate states S and the desired average convergence time λ_e. The algorithm returns the desired program $\mathcal{P}'$ if a solution is found that solves the problem in Definition 8. Otherwise, it returns ϕ.

5.1 Approach 1 (SCP): A Refinement Procedure via Including Shortest Convergence Paths

In this section, we present SCP (Shortest Convergence Path) – a fast heuristic that focuses on reducing convergence time without considering the number of preserved transitions. Based on the idea of Dec_2, SCP repairs a given program by preserving only those transitions that lead a program to shortest convergence paths. This reduces/eliminates choices that the scheduler can play. However, it is anticipated that it would eliminate a large number of transitions/behaviors. To identify transitions that lead to the shortest convergence path, our approach SCP performs a backward computation from legitimate states.

Figure 1 gives pseudocode for the overall refinement algorithm of SCP. We initialize the revised program to $\emptyset$ and set the current reachable state set to the set of legitimate states. In each iteration of the RepairBySCP loop, starting from current reachable states, we perform a backward computation to identify transitions that lead to the shortest convergence path. Based on such one-step backward search per iteration, we identify newly reachable state S_{next} (Line 8) and calculate the transitions that lead program from $S_{next}, S_{current}$ (Line 9). Then, we update the current reachable state set in Line 10. In this step, our implementation simply performs $S_{current} \cup S_{next}$. Now, we use the transitions computed in Line 9 to update the current revised program (Line 11). In particular, our implementation simply performs $\delta_{current} \cup \delta_{tmp}$. Based on the updated program transitions, we re-calculate the average convergence time of $\mathcal{P}^c$. There are two scenarios for our algorithm to stop the *while* loop. One is that if we reach a program $\mathcal{P}^c$ whose average convergence time is larger than λ_e, the loop will stop. As in Line 7, $\mathcal{P}'$ records the revised program that fits the average convergence time requirement. The other one is that our refinement procedure has included all the transitions in the shortest convergence path and the generated program fits the average convergence time requirement, that is, $\lambda_r < \lambda_e$. In this case, our refinement process will break the loop.

Hence, after executing such an iterative refinement procedure, our refinement algorithm generates a program $\mathcal{P}'$ that holds the maximum $\lambda_{\mathcal{P}'}$ and $\lambda_{\mathcal{P}'} \leq \lambda_e$.

5.2 Approach 2 (ELP): A Refinement Procedure via Eliminating Maximal Transitions

In this section, we present ELP (Eliminate Maximal Path) –a heuristic that focuses on reducing convergence time while preserving maximum non-determinism.

The key idea of ELP is to find the set S_{max}, the set of states from where the (worst case) convergence path is the longest. Let λ_{worst} be this worst case convergence path. Let S_{next} be the set of states from where the worst case convergence path is $\lambda_{worst} - 1$. After finding S_{max} and S_{next}, we remove transitions $\{(s_1, s_2) | s_1 \in S_{max} \wedge s_2 \in S_{next}\}$. This process is repeated until we find the desired program or conclude that realizing such a program is impossible. As an illustration, consider the Figure 3. This figure shows four states s_1, s_2, s_3 and s_4. Assume that the worst case convergence path from s_2, s_3 and s_4 are $10, 7$ and 5

```
RepairBySCP(P, λ_e):
Input    λ_e: the expected average convergence time.
         P: transitions δ_P and invariant S_P.
Output   P': λ_P' ≤ λ_e
1        S_current = invariant;
2        δ_current = empty;
3        P' = ∅;
4        P^c = ∅;
5        do
6        {
7            P' = P^c;
8            S_next ← BackwardOneStepCompute(S_current);
9            δ_tmp ← CalculateTransition(S_next, S_current);
10           S_current ← UpdateCurrentState(S_current, S_next);
11           P^c ← RefineProgramTransitions(δ_current, δ_tmp);
12           λ_r ← CalculateAvgConvTime(P^c);
13           if (P^c = ∅)
14               break ;
15       }
16       while (λ_r > λ_e);// λ_r is the current average convergence time after refinement.
17       Return P';
```

Figure 1 SCP: program repair by preserving shortest convergence path.

```
RepairByELP(P, λ_e):
Input    λ_e: the expected average convergence time.
         P: transitions δ_P and invariant S_P.
Output   P': λ_P' ≤ λ_e
1        P' = P;
2        do
3        {
4            S_max ← CalculateMaximalState(P');
5            S_next ← CalculateNextMaxState(P');
6            δ_maxGroup ← CalculateMaxTrans(S_max, S_next);
7            P' ← RefineProgramTransitions(P^c, δ_maxGroup);
8            λ_r ← CalculateAvgConvTime(P^c);
9            if (P' = ∅)
10               break ;
11       }
12       while (λ_r > λ_e);// λ_r is the current average convergence time after refinement.
13       Return P'
```

Figure 2 ELP: A Refinement Procedure via Eliminating Maximal Transitions.

respectively. In that case, worst case path from s_1 is 11. Also, assume that $s_1 \in S_{max}$. In this case, we remove the transition (s_1, s_2).

Figure 2 gives the pseudo code for the overall refinement algorithm. The procedure RepairByELP repairs the given program $\mathcal{P}$ top-down, starting with original program transition $\delta_{\mathcal{P}}$. In each iteration, we calculate the maximal state set S_{max} (Line 4) and the next-maximal state set S_{next} (Line 5) for the current revised program $\mathcal{P}'$. With S_{max} and S_{next}, we calculate a group of maximal transitions. As in Line 6, we calculate maximal transitions (denoted as $\delta_{maxGroup}$). Then in Line 7, we repair program by eliminating $\delta_{maxGroup}$ from current program transitions set. Line 8 calculates the current average (maximal) convergence time for $\mathcal{P}'$. Line 9 describes a possible case where our algorithm reaches an empty program. If this case occurs, Line 10 would break the computation loop. Otherwise, the whole RepairByELP loop will stop when it reaches a point where current average convergence time is less than λ_e. The resulting program $\mathcal{P}'$ fits the average convergence time requirement, that is $\lambda_{\mathcal{P}'} \leq \lambda_e$.

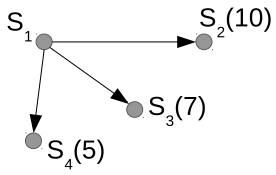

Figure 3 Four states s_1, s_2, s_3 and s_4.

```
RepairByKBP(P, λ_e):
Input    λ_e: the expected average convergence time.
         P: transitions δ_P and invariant S_P.
Output   P': λ_P' ≤ λ_e
1        P' = P;
2        do
3        {
4            S_max ← CalculateMaxState(P');
5            δ_nonMin ← CalculateNotMinTrans(S_max);
6            P' ← RefineProgram(P', δ_nonMin);
7            λ_r ← CalculateAvgConvTime(P');
8            if  (P^c = ∅)
9                break ;
10       }
11       while (λ_r > λ_e);// λ_r is the current average convergence time after refinement.
12       Return P';
```

Figure 4 KBP: A Refinement Procedure via Removing NonMinimum Transitions.

5.3 Approach 3 (KBP): A Refinement Procedure via Eliminating NonMinimum Transitions

In this section, we present KBP (Keep Best Path) – a heuristic that focuses on reducing convergence time with considering preservation of least non-determinism for those maximal states. Similar to the approach ELP, during the refinement procedure, our approach KBP iteratively removes a group of transitions from current program transition set until we reach a point where the revised program fits the average convergence time requirement. The difference between KBP and ELP is that we remove not only the maximal transitions from S_{max} but also other transitions except those that provide the best recovery time.

Once again, consider the transitions in Figure 3. In this figure, assuming that $s_1 \in S_{max}$, we keep the transition (s_1, s_4) and remove (s_1, s_2) and (s_1, s_3). Observe that in this case, we are removing more transitions while making a bigger impact on the average convergence time. Compared with ELP, we expect that KBP will reduce the time for repair but it will result in more transitions being removed.

The procedure RepairByKBP in Figure 4 repairs the given program top-down, starting with original transitions. Specifically, we calculate S_{max} in Line 4 and corresponding nonminimum transition set δ_{nonmin} for each state in S_{max}. Then, we repair program by eliminating δ_{nonmin} from current program transitions (Line 6). Line 7 calculates the current average (maximal) convergence time for $\mathcal{P}'$. Line 8 describes a possible case where our algorithm reaches an empty program. If this case occurs, Line 9 would break the computation loop. The resulting program $\mathcal{P}'$ is one solution that fits the average convergence time requirement, that is, $\lambda_{\mathcal{P}'} \leq \lambda_e$.

5.4 Approach 4 (RIA): A Refinement Procedure via Revising Maximal *Actions* with Minimal *Actions*

While the previous three approaches focused on repair at transition level, in this approach, we focus on additional structure in the given program to perform the repair. This allows one to take into consideration problems that arise in distributed systems as well as possible limitations on how programs are evaluated.

Before we describe our approach, we consider the case where the state space is more compactly represented by variables and program transitions are compactly represented using guarded commands of the form $g \longrightarrow st$. In particular, in this case, the state space is obtained by assigning each variable value from its respective domain. And, transitions corresponding to an action $g \longrightarrow st$, where g is a Boolean expression involving program variables and st is a statement that updates those program variables, are represented by the set $\{(s_0, s_1)|g$ evaluates to true in s_0 and s_1 is obtained by updating those variables as prescribed by $s_1\}$ [1]

Specifically, in this approach, we focus on revising the given program so that the guards and statements in the repaired program are *comparable* to that in the original program. To illustrate our approach, consider Figure 3. In this figure, let the transition (s_1, s_a) be executed by the action $g_a \longrightarrow st_a$, where $2 \leq a \leq 4$. In this figure, in approach ELP, we removed the transition (s_1, s_2). In RIA, we achieve the same by restricting the corresponding action $g_2 \longrightarrow st_2$ to be executed only if the action corresponding to (s_1, s_4) is not enabled. In other words, we change the action to $g_2 \wedge (\neg g_4) \longrightarrow st_2$. Observe that this change causes removal of additional transitions that start from a state where g_2 is true and g_4 is false. This approach is based on the heuristic that this overall change will result in reduction in the average convergence time. Observe that with this change, the guards involved in the repaired program are a combination of the guards involved in the original program. And, the statements in the repaired program are same as that in the original program. Since the guards of the original program represent predicates that could be checked in the original program, this allows the user to control the types of actions that can appear in the repaired program.

Figure 5 gives the pseudo code for the overall refinement algorithm. Specifically, Line 4 computes S_{max}, the state set in which each state has a possibility to reach maximal convergence path. Line 5 calculate the maximal actions for each state in S_{max}. Line 6 calculate the minimal actions for each state in S_{max}. Line 7 revise the maximal actions for each state in S_{max} using the corresponding minimal actions. Then Line 8 refines program by repairing those maximal actions from current program transition set. Line 9 calculates the current average (maximal) convergence time for $\mathcal{P}'$. Line 10 describes a possible case where our algorithm reaches an empty program. If this situation occurs, Line 11 would break the computation loop. Otherwise, the resulting program $\mathcal{P}'$ satisfies the average convergence time, that is $\lambda_{\mathcal{P}'} \leq \lambda_e$.

After executing such an interatively refinement procedure, our refinement algorithm either reaches an empty program or returns a program that fits the average convergence time requirement.

[1] As an illustration, if the program had two variables x and y with domain $\{0, 1\}$ and $\{0, 1, 2\}$ respectively then the state space contains 6 possible states $00, 01, 02, 10, 11$ and 12 where the first value denotes the value of x and the second denotes the value of y. And, action $x = y \longrightarrow x = 0$ corresponds to transitions $(00, 00), (11, 01)$.

RepairByRIA($\mathcal{P}, \lambda_e$):
Input λ_e: the expected average convergence time.
$\mathcal{P}$: transitions $\delta_{\mathcal{P}}$ and invariant $S_{\mathcal{P}}$.
Output $\mathcal{P}'$: $\lambda_{\mathcal{P}'} \leq \lambda_e$

```
1   P' = P;
2   do
3   {
4       S_max ← CalculateMaxState(P');
5       act_max ← CalculateMaxAct(S_max, P');
6       act_min ← CalculateMinAct(S_min, P');
7       act'_max ← RepairActions(act_max, act_min);
8       P' ← RefineProgram(act'_max, P');
9       λ_r ← CalculateAvgConTime(P');
10      if (P' = ∅)
11          break ;
12  }
13  while (λ_r > λ_e);// λ_r is the actual average convergence time of P.
14  Return P';
```

Figure 5 RIA: A Refinement Procedure via Revising Maximal *Actions* with Minimal *Actions*.

5.5 Approach 5 (RIAD): A Refinement Procedure via Revising Maximal Actions with Distribution Consideration

In this section, we extend RIA to take into account what guards could be used in repairing a given action. In turn, this allows to fully capture the requirements of a distributed system. To illustrate this, consider the case where the nature of distributed systems prevents a process from accessing all program variables. Rather, each process is only allowed to read and write a subset of variables.

Recall that in RIA, we restricted the guard of one action by negation of the guard of another action. Given a guard, it is straightforward to identify the variables that it is allowed to read. Hence, for each action, we identify *neighborhood* actions that can be used to restrict it while preserving the read/write restrictions. By only selecting this subset of actions, we can ensure that the synthesized program satisfies the read/write restrictions of the given system. Since RIAD is similar to RIA (except for this neighborhood restriction), we do not provide a detailed algorithm for RIAD.

5.6 Approach 6 (SSP): A Refinement Procedure via Eliminating Maximal Transitions from a Reduced Program

In this section, we propose SSP (Solve Simpler Problem). The main idea of this algorithm is to use the same repair technique as ELP (from Section 5.2) on a program $\mathcal{P}_r$ (described next) rather than the original program $\mathcal{P}$.

Let λ_e be the desired average convergence time. Let λ_{worst} be the worst case convergence time of $\mathcal{P}$. The goal of $\mathcal{P}_r$ is to ensure that the worst case convergence time is bounded by $\lambda_{\mathcal{P}_r}$ where $\lambda_{worst} \geq \lambda_{\mathcal{P}_r} \geq \lambda_e$. Subsequently, we utilize ELP to remove any behaviors from $\mathcal{P}_r$ to ensure that the average convergence requirements are satisfied.

The motivation behind SSP is that each step involved could be implemented efficiently. Specifically, the first step, which involves ensuring worst case behavior, is simpler. This is due to the fact that worst case analysis of repaired programs is substantially easier than average case behavior. Also, the transitions removed in the first step are good candidates for

removal from the desired program $\mathcal{P}'$. Also, it is anticipated that $\mathcal{P}_r$ is close to the desired program $\mathcal{P}'$. Hence, the amount of time involved in the second step would be small as well.

Observe that SSP provides a continuum of possible values for $\lambda_{\mathcal{P}_r}$. At one extreme, choosing $\lambda_{\mathcal{P}_r} = \lambda_{worst}$ will result in SSP to be equivalent to ELP. At another extreme, choosing $\lambda_{\mathcal{P}_r} = \lambda$ will result in unnecessary removal of transitions in the first step and obviate the need for the second step. For the sake of analysis, we choose $\lambda_{\mathcal{P}_r}$ to be the average of λ_{worst} and λ_e. Since the construction of $\mathcal{P}_r$ is straightforward and the remaining algorithm is same as ELP, we do not provide detailed algorithm for SSP.

6 Case Study & Experiment Results

We have developed a tool *Rtime* that implements the six approaches described previously. *Rtime* takes as input the following parameters:

- the input program $\mathcal{P}$,
- the set of legitimate states, S
- the desired convergence time, λ, and
- approach to be used for adding average convergence time

It identifies the program that satisfies the requirements of Problem 8. For the sake of analysis, we allow *Rtime* to output a program even if it removes all transitions from some state $s \in S_{\mathcal{P}}$. When this happens, the fraction of transitions that are preserved will be lower as well. Since our goal is to compare the level of non-determinism left in the program and the time taken for synthesis, this allows us to compare the different approaches directly.

Observe that all our approaches are sound by construction, i.e., when they output a program, we have already validated that the average convergence time of that program is less than the given parameter λ. Also, since these programs only use polynomial time, the number of transitions they preserve is not necessarily maximum. Also, if any of these approaches remove all transitions from some state s, making s be a deadlock state then they cannot satisfy $S_{\mathcal{P}'} = S_{\mathcal{P}}$. However, instead of declaring failure in this case, we report the number of transitions still preserved in the program. This allows us to compare all approaches in all examples. Note that the worst case is that all states outside S become deadlock states. In this case, the fraction of preserved transitions will be 0.

We now demonstrate our approaches on a classic stabilizing algorithm, which is K-state token ring program [10] and the Stabilizing Tree based algorithm [21] that is obtained adding stabilization to the classic mutual exclusion algorithm by Raymond [21]. All the experiments are performed on an Intel Core i7 machine 2.90GHz with 8GB memory. Also, the reachability analysis required for the different approaches is performed with the BDD package [8].

6.1 K-state Token Ring Program

We give a brief description of the K-state token ring program from [10]. The program $\mathcal{P}_{tk}$ consists of n processes, $0..(n-1)$, that are arranged in a unidirectional ring. For each process p_i, it has one variable x_i with domain $\{0, 1, \ldots \text{K-1}\}$.

$$\begin{array}{lllll} Action_0: & x_0 == x_{n-1} & \longrightarrow & x_0 = (x_0 + 1) \bmod K; \\ Action_1: & x_i \neq x_{i-1} & \longrightarrow & x_i = x_{i-1}; \end{array}$$

In the above two actions, $Action_0$ is only for process p_0. When $x_0 == x_{n-1}$ is satisfied, this action is enabled for execution. If chosen for execution, process 0 increments x_0 by 1 in modulo K arithmetic. $Action_1$ is for all other processes p_i, $i \neq 0$. When x_i differs from x_{i-1}, $Action_1$ is enabled for execution. When p_i executes its action, it sets x_i to the value of x_{i-1}.

Legitimate states. The legitimate states of the program are those states where only one token is circulated along the ring. To calculate this set, we start from a state where all x values are 0. Then, we compute all the states reached by the execution of the above program.

▶ Remark. In subsequent analysis, we let $K = n$ to ensure that convergence to legitimate states is guaranteed.

We conduct our experiments for repairing the token-ring program $\mathcal{P}_{tk}$ with different average convergence requirements. Instead of using specific real number values for the desired average convergence time, we use a fraction of the existing worst case convergence time. This is due to the fact that the time required to obtain an average convergence time of 10 for 3 processes is not comparable to that for 4 processes. Hence, to obtain a valid comparison, we first identify the average convergence time for each of the programs. Subsequently, we use a fraction of this worst case requirement as the value of desired average convergence time. Specifically, we use three values: λ_1, λ_2 and λ_3, where λ_1 is 70% of the original average convergence time, λ_2 is 80% of the original convergence time and λ_3 is 90% of the original convergence time. We perform our experiments for $k \in \{3, 4, 5, 6, 7\}$, where k is the number of processes in the input program.

Our results are as shown in Tables 1 and 2. Specifically, Table 1 presents transition preservation percentage of original program for the revised program generated by our approaches. Table 2 presents revision time (in seconds) for generating the revised program that fits the λ requirements. In particular, we run each experiment for at most one hour. We set the running time as N/A when the experiment couldn't return a result within one hour. From these results, we find that SCP identifies the desired program most quickly. For example for 7 processes when requiring λ_3, SCP could find the desired program within 0.26 seconds. However, it eliminated most of the transitions. It only maintained 00.03 percentage of the original transitions. By contrast, KBP took significantly longer time. However, it kept 81.26 percentage of transitions. Observed from these results, for token-ring program, we find that RIAD provides the best approach for tradeoff between the time required to obtain the desired program and the number of transitions preserved in that program.

6.2 Stabilizing Algorithm Based on Raymond's Tree based Mutual Exclusion Program

This program, $\mathcal{P}_{rt}$, consists of n processes, numbered $0..(n-1)$. These processes are arranged in a fixed binary tree. For each process p_i, it has one variable h_i with domain$\{i, NBR_i\}$, where NBR_i denotes the neighbor processes of p_i. When $h_i = i$, then process p_i has the token. Otherwise, the holder of p_i points to one of its neighbors. In particular, $\mathcal{P}_{rt}$ provides three types of convergence actions as follows.

$$
\begin{array}{lllll}
Action_0: & h_i \neq NBR_i \cup \{i\} & \longrightarrow & h_i = PR_i | i | NBR_i; \\
Action_1: & h_i \neq PR_i \wedge h_{PR_i} \neq i & \longrightarrow & h_i = PR_i; \\
Action_2: & h_i = PR_i \wedge h_{PR_i} = i & \longrightarrow & h_{PR_i} = PR_i;
\end{array}
$$

In the above actions, PR_i denotes the parent process of p_i in the static tree and NBR_i denotes the neighbor processes of p_i. Specifically, the first action $Action_0$ ensures that the holder of a process points to its neighbors or itself. This action is executed by all processes. The second and third actions are exeted by all processes except the root process. Of these, the second action ensures that the holder of p_i is either PR_i or holder of PR_i is same as i. And, the third action ensures that the holder relation between p_i and PR_i is acyclic.

Table 1 Transition Coverage Percentage of Different Apporaches for K-state Token Ring Program.

λ	# proc	SCP	ELP	KBP	RIA	RIAD	SSP
0.7	3	46.67%	**80.00%**	51.00%	60.00%	48.57%	**80.00%**
0.7	4	16.25%	**80.00%**	60.16%	70.00%	62.50%	**80.00%**
0.7	5	02.96%	63.62%	53.36%	55.97%	**74.70%**	63.62%
0.7	6	00.37%	53.96%	33.48%	64.58%	**79.57%**	53.96%
0.7	7	00.03%	46.91%	26.50%	53.85%	**82.60%**	46.91%
0.8	3	46.67%	**93.33%**	51.11%	60.00%	48.57%	80.00%
0.8	4	16.25%	**88.75%**	68.60%	**92.50%**	**92.50%**	80.00%
0.8	5	02.96%	**81.84%**	65.29%	77.81%	74.70%	71.53%
0.8	6	00.37%	72.76%	60.01%	64.58%	**79.57%**	61.00%
0.8	7	00.03%	64.85%	55.70%	70.11%	**82.60%**	59.27%
0.9	3	46.67%	**93.33%**	82.22%	**94.29%**	**94.29%**	80.00%
0.9	4	16.25%	**95.63%**	85.47%	**92.50%**	**92.50%**	80.00%
0.9	5	02.96%	91.11%	84.69%	**99.57%**	**99.57%**	63.62%
0.9	6	00.37%	88.16%	83.07%	**99.44%**	**99.44%**	60.99%
0.9	7	00.03%	**83.93%**	**81.26%**	**84.67%**	**82.60%**	53.76%

Table 2 Revision Time (in seconds) of Different Approaches for K-state Token Ring Program.

λ	# proc	SCP	ELP	KBP	RIA	RIAD	SSP
0.7	3	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
0.7	4	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
0.7	5	<0.01	0.06	0.20	0.07	0.06	0.07
0.7	6	0.02	3.34	9.05	0.50	0.43	3.30
0.7	7	0.26	N/A	1,134.79	4.30	2.60	N/A
0.8	3	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
0.8	4	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
0.8	5	<0.01	0.04	0.16	0.05	0.06	0.06
0.8	6	0.024	1.93	6.34	0.50	0.43	2.80
0.8	7	0.26	N/A	905.66	3.83	2.58	N/A
0.9	3	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
0.9	4	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
0.9	5	<0.01	0.02	0.09	0.03	0.04	0.07
0.9	6	0.02	0.95	4.13	0.29	0.34	2.71
0.9	7	0.25	206.50	580.12	2.80	2.77	N/A

Table 3 Transition Coverage of Different Apporaches(Raymond Tree Based Mutual Exclusion Program).

λ	# proc	SCP	ELP	KBP	RIA	RIAD	SSP
0.7	4	3.13%	**86.03%**	**82.91%**	58.46%	58.46%	77.67 %
0.7	5	0.38%	**81.25%**	**83.73%**	43.02%	58.46%	**81.25 %**
0.7	6	00.03 %	**85.60%**	84.43%	38.97%	52.06 %	**81.96%**
0.7	7	<0.01%	**86.22%**	**85.60%**	33.91%	49.66%	**84.40%**
0.7	8	<0.01%	**87.20 %**	**87.81 %**	34.14%	39.00%	**86.25%**
0.8	4	3.13%	**77.67%**	**75.83%**	49.63%	53.66%	**77.67 %**
0.8	5	0.38%	**85.66%**	**83.73%**	55.12%	60.98%	**81.25 %**
0.8	6	00.03%	**85.60%**	**87.73%**	40.44%	57.35 %	81.96%
0.8	7	<0.01%	**88.39%**	87.37%	36.04%	52.70 %	**84.18%**
0.8	8	<0.01%	**88.37%**	**89.43%**	35.00%	49.90%	**86.23%**
0.9	4	3.13%	86.03%	**91.64%**	76.47%	76.47%	77.67 %
0.9	5	0.38%	**91.31%**	**89.13%**	55.12%	63.41%	78.61 %
0.9	6	0.03%	**89.22%**	**91.89%**	62.21%	58.82%	81.96%
0.9	7	<0.01%	**91.42%**	**93.58%**	61.92%	57.77%	84.18%
0.9	8	<0.01%	**92.88%**	**91.87%**	63.85%	53.94%	86.23%

Table 4 Revision Time (in seconds) of Different Approaches (Raymond Tree Based Mutual Exclusion Program).

λ	# proc	SCP	ELP	KBP	RIA	RIAD	SSP
0.7	4	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
0.7	5	<0.01	0.01	0.03	0.03	0.02	0.01
0.7	6	0.02	0.25	1.11	1.08	0.71	0.30
0.7	7	0.15	11.87	53.52	49.80	30.53	12.67
0.7	8	0.01	17.47	1,741.98	3,249.34	2,180.22	24.73
0.8	4	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
0.8	5	<0.01	0.01	0.03	0.02	0.02	0.01
0.8	6	0.02	0.24	1.14	1.00	0.69	0.30
0.8	7	0.16	9.45	50.18	47.77	32.58	12.67
0.8	8	0.01	12.59	1,706.08	3,170.72	1,905.08	24.54
0.9	4	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
0.9	5	<0.01	<0.01	0.02	0.02	0.01	0.01
0.9	6	0.02	0.19	0.95	0.67	0.58	0.30
0.9	7	0.16	6 .35	32.63	42.37	28.80	12.67
0.9	8	0.01	5.39	1,678.03	1,845.25	1,845.94	24.54

Tables 3 and 4 present our experiment results for Raymond Tree based mutual exclusion program. In particular, Table 3 illustrates the transition coverage percentage of the original program for the newly generated program with respect to our six approaches. Table 4 shows the performance of these six approach in revision time (in seconds). Similar to previous study, we run each experiment for at most one hour. If the revision time exceeds one hour, we will identify it as N/A in the table.

We perform our experiments for $n \in \{4, 5, 6, 7, 8\}$, where n is the number of processes in the input program. From these results, same as in the experiment for token-ring program, SCP identifies the desired program most quickly. For example for 7 processes when λ is set to 0.9, SCP could find the desired program within 0.25 seconds. However, it eliminated most of the transitions. It maintained less than 0.01 percentage of the original transitions. By contrast, SSP took significantly longer time. However, it kept 86.23 percentage of transitions. Observed from these results, we find that KBP provide the best approach for tradeoff between the time required to obtain the desired program and the number of transitions preserved in that program.

7 Related Work

In this work, we focused on the problem of adding average recovery in the presence of an adversarial scheduler. The closest comparable work to this is [1] where authors have considered the problem of synthesizing a program with given average recovery time. In this work, the authors omit the notion of an adversarial scheduler. Instead, they assume that each non-deterministic choice is resolved through randomization. Hence, if the program synthesized using these approaches is used to reduce the recovery time then its average convergence time in the presence of an adversary can be higher. By contrast, in our work, we have focused on the problem of guaranteeing average recovery time in the presence of an adversary. In other words, the solution provided by our approaches will ensure that even if the adversary puts arbitrary probabilities on different non-deterministic choices, the average recovery constraint will be satisfied.

The work in [7, 12, 19, 6, 14, 18, 20, 22, 16, 13, 15] has focused on the topic of adding safety properties, liveness properties and fault-tolerance properties. The properties considered in this work are represented using the framework of safety and liveness by Alpern and Schneider [2]. As discussed in Section 1, each program computation can be evaluated independently to determine whether it satisfies or violates the specification. By contrast, the average response time considered in this paper cannot be represented using the framework in [2]. It requires a more generalized framework of hyperproperties [9]. In this framework, satisfaction of a requirement is determined by *all* computations included by the program. In particular, the average convergence time is an instance of a hyperliveness property. While the work in this paper enables one to repair a given program to add one hyperliveness property, one future work in this area is to generalize to other hypersafety and hyperliveness properties.

8 Conclusion

We focused on the problem of revising a given program to add average recovery time in the presence of an adversarial scheduler who could force the program to choose the least desirable path during recovery. Adding average recovery time requires removal of some behaviors/transitions that cause the recovery to increase beyond acceptable limit. We showed that ensuring that only a minimum number of transitions are removed is NP-hard. Hence, we proposed six different heuristics.

We find that, as expected, the first heuristic, SCP, constructs the desired program in the least amount of time. However, it ends up removing a large number of transitions unnecessarily. For example, in case of the token ring program with 7 processes, it found the desired program in 0.2 seconds. However, it preserved only 00.03 percent of transitions. By contrast, RIAD preserved 82.60% of transitions but took around 2 seconds to obtain the desired program.

We presented the analysis of our six approaches in two case studies. Based on these case studies, we find that RIAD and KBP provide the best approaches for tradeoff between the time required to obtain the desired program and the number of transitions preserved in that program. We plan to conduct more case studies in the future so that we could identify the effect of specific program structure on program revision for the problem of average recovery time.

In our work, we focused on the problem of average recovery time in the presence of an adversarial scheduler. There are two aspects to the recovery in the presence of faults: (1) state to which the program is perturbed to when faults stop occurring, and (2) the non-deterministic choices made by the scheduler during recovery. Regarding the first aspect, we considered the case where the state to which the program is perturbed to is chosen with equal probability. However, it is straightforward to extend it to the case where each state is associated with a different probability distribution. This will only change the way average convergence time is computed. However, all our aprpaoches could still be used. Regarding the second aspect, we assumed that the scheduler can arbitrarily choose the execution order. Our work could also be extended to other choices of scheduler.

This work also demonstrates the feasibility of adding some hyperproperties [9]. Specifically, the requirement of average convergence time cannot be expressed in terms of the framework of safety and liveness by [2]. This is due to the fact that checking whether a given program computation is acceptable or not depends upon other computations involved in the program. A possible future work in this area is to pursue such repair for other hyperproperties.

References

1 Saba Aflaki, Fathiyeh Faghih, and Borzoo Bonakdarpour. Synthesizing self-stabilizing protocols under average recovery time constraints. In *35th IEEE International Conference on Distributed Computing Systems, ICDCS 2015, Columbus, OH, USA, June 29 – July 2, 2015*, pages 579–588, 2015.

2 Bowen Alpern and Fred B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985. doi:10.1016/0020-0190(85)90056-0.

3 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.

4 Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. *Mathematical Structures in Computer Science*, 21(6):1207–1252, 2011. doi:10.1017/S0960129511000193.

5 Borzoo Bonakdarpour. *Automated Revision of Distributed and Real-Time Programs*. PhD thesis, Michigan State University, 2009.

6 Borzoo Bonakdarpour, Ali Ebnenasir, and Sandeep S. Kulkarni. Complexity results in revising unity programs. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4(1):5:1–5:28, Feb. 2009.

7 Borzoo Bonakdarpour, Sandeep S. Kulkarni, and Fuad Abujarad. Symbolic synthesis of masking fault-tolerant distributed programs. *Distributed Computing*, 25(1):83–108, 2012. doi:10.1007/s00446-011-0139-3.

8 Buddy – a binary decision diagram package. http://buddy.sourceforge.net/manual/main.html.

9 Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010. doi:10.3233/JCS-2009-0393.

10 Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974. doi:10.1145/361179.361202.

11 Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

12 Ali Ebnenasir and Aly Farahat. Swarm synthesis of convergence for symmetric protocols. In Cristian Constantinescu and Miguel P. Correia, editors, *EDCC*, pages 13–24. IEEE, 2012. doi:10.1109/EDCC.2012.22.

13 Ali Ebnenasir and Aly Farahat. Swarm synthesis of convergence for symmetric protocols. In *Proceedings of the Ninth European Dependable Computing Conference*, pages 13–24, 2012.

14 Ali Ebnenasir and Sandeep S. Kulkarni. Feasibility of stepwise design of multitolerant programs. *ACM Trans. Softw. Eng. Methodol.*, 21(1):1, 2011. doi:10.1145/2063239.2063240.

15 Aly Farahat and Ali Ebnenasir. A lightweight method for automated design of convergence in network protocols. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 7(4):38:1–38:36, December 2012.

16 Aly Farahat and Ali Ebnenasir. Local reasoning for global convergence of parameterized rings. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 496–505, 2012.

17 Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

18 Alex Klinkhamer and Ali Ebnenasir. A software tool for swarm synthesis of self-stabilization. http://www.cs.mtu.edu/~apklinkh/protocon/index.html.

19 Alex Klinkhamer and Ali Ebnenasir. On the complexity of adding convergence. In Farhad Arbab and Marjan Sirjani, editors, *FSEN*, volume 8161 of *Lecture Notes in Computer Science*, pages 17–33. Springer, 2013. doi:10.1007/978-3-642-40213-5_2.

20 Alex Klinkhamer and Ali Ebnenasir. Synthesizing self-stabilization through superposition and backtracking. In *16th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 252–267. Springer, 2014.

21 K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 7(1):61–77, 1989.

22 Amer Tahat and Ali Ebnenasir. A hybrid method for the verification and synthesis of parameterized self-stabilizing protocols. In *24th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR)*, 2014.