Lazy Repair for Addition of Fault-tolerance to Distributed Programs

Mohammad Roohitavaf, Yiyan Lin, and Sandeep S. Kulkarni
Computer Science and Engineering Department
Michigan State University
East Lansing, MI, USA
{roohitav, linyiyan, sandeep}@cse.msu.edu

Abstract—We focus on the issue of realizability constraints in the context of model repair. Model repair focuses on revising a given program to satisfy new properties of interest while satisfying existing properties such as fault-tolerance. An important difficulty in using model repair is that the repaired model must be realizable in the constraints given by the underlying system. It is well-known that these realizability constraints cause an increase in the complexity of model repair (e.g., from P to NP-complete). Hence, existing approaches for adding fault-tolerance to distributed program focuses on cautious repair where in every step, the model being repaired satisfies the realizability constraints. They also utilize heuristics to reduce the complexity of repair.

In this work, we focus on using lazy repair while adding fault-tolerance. Specifically, in this work, we utilize a two-step approach. The first step ignores the realizability constraints and performs model repair to add fault-tolerance. This ensures that the resulting program satisfies the desired property of interest (namely, fault-tolerance) although it may not be realizable. The second step attempts to revise this program to ensure that realizability constraints are satisfied without creating new program behaviors. This ensures that the resulting program satisfies both the properties of interest and realizability constraints. We demonstrate that this approach is more efficient than the cautious repair algorithm in the literature. We also note that inherently the lazy repair approach is also applicable in other contexts such as synchronous systems, cyber-physical systems, etc.

Keywords—Model Repair; Program Repair; Distributed Systems; Fault-Tolerance,

I. INTRODUCTION

Model repair focuses on the problem of repairing a given model to add some new requirements such as safety, liveness, fault-tolerance and/or security. In model repair, we begin with a model that represents an existing system that satisfies certain properties. The goal is to revise this model so that it preserves the existing properties and also satisfies the new desired properties.

The broad usage of automated model repair is as follows (cf. Figure 1): We begin with an original program for the underlying system. Subsequently, we obtain a model of that program (using either manual or automated approach). For subsequent discussion, we assume that this model is a finite state automaton that incorporates suitable abstraction. Specifically, the states in this automaton correspond to different program states and transitions correspond to actions performed by the program. Subsequently, we utilize model repair to automatically obtain the model that satisfies new properties while preserving existing properties. Once again, the revised model is also a finite state automaton. Finally the revised model is then translated into a concrete program (using either manual or automated approach).

\[
p \xrightarrow{abstracts} M \models \phi \xrightarrow{revises} M' \models \phi' \xrightarrow{realizes} p'
\]

Figure 1. Model Repair

As the structure in Figure 1 illustrates, the repaired model needs to be translated into a program for the underlying system. It follows that the generated model must satisfy certain properties to ensure that it can be converted into a program in the underlying system. We illustrate some simple instances of such constraints below.

- Suppose that we are trying to repair a program that models byzantine faults. In this case, if the generated model consists of an action of the form if a neighbor is byzantine then ignore the value received from it then such a model cannot be translated into a valid program. This is based on the assumption that a byzantine process is difficult/impossible to detect.
- Suppose we are trying to repair a program that reads some variables provided by the environment. These variables are read-only variables as far as the program is concerned. Hence, if the repaired model includes some actions that change the values of these variables then the corresponding model cannot be translated into a valid program.
- Consider a program where simultaneously changing the state of multiple processes is impossible (e.g., asynchronous systems). If the repaired model includes actions that change the state of multiple processes at once, then it cannot be realized as a valid program.
- Consider a system with parallel execution. In such systems, we need to ensure that the time allocated to two parallel tasks is the maximum
time required for those tasks (as opposed to the sum of time required if they were to be performed sequentially).

As the above discussion illustrates, the problem of model repair includes two issues: (1) adding the desired property while preserving the existing properties, and (2) ensuring that the generated model can be realized in the underlying system. We denote the latter as realizability constraints.

The realizability constraints can affect the problem of model repair substantially. In particular, in [1], it is shown that the problem adding masking fault-tolerance to a distributed program is NP-complete. However, if the realizability constraints are omitted then the problem can be solved in polynomial time.

A. Previous Approach: Cautious Repair

In [2], authors introduced an approach for dealing with realizability constraints in distributed systems. In this work, we denote the approach in [2] as cautious repair. Intuitively, in this work, in each step, heuristics are used to either remove transitions (behaviors) (e.g., to remove safety violating behaviors) or to add transitions (behaviors) (e.g., to add recovery).

While we will describe the approach in [2] in Section IV, the reason we call this approach as cautious repair is that in every step where it adds or removes transitions, it takes additional steps to ensure that the new model still satisfies the realizability constraints. Although this increases the complexity of each step, it ensures that at any stage, the model being repaired can be realized, i.e., it can be translated into a distributed program.

B. Goals: Lazy Repair

In this paper, we focus on the problem of lazy repair. Intuitively, lazy repair separates the concerns of model repair and realizability constraints. In other words, it ignores (some or all) of realizability constraints during repair. Thus, in the first step, it obtains a model $M''$ that satisfies the desired properties but may be unrealizable. Subsequently, it revises the repaired (but unrealizable) model to ensure that realizability constraints are satisfied. Thus, the approach is as shown in Figure 2.

In Step 1, we perform model repair while ignoring some or all realizability constraints. In this step, the goal is to provide maximal behavior to the intermediate program. By maximal behavior, we mean that transitions that are removed in this step are transitions that must be removed (or highly likely to be removed) in the repair process. Examples of such transitions include those transitions that cause violation of safety or transitions that reach states from where safety can be violated. Likewise, it adds transitions that are potentially useful. In the second step, we focus only on removing transitions from the intermediate model so that realizability constraints are satisfied. Observe that the intermediate program already satisfies the desired property. However, it is not relizable. Since we focus on universal specifications (e.g., LTL or safety and liveness properties, fault-tolerance properties) that require that all computations satisfy the given safety and liveness requirements, we can remove non-determinism from the intermediate program and still preserve the specification as long as removal of transitions does not create deadlock states. This is due to the fact that removal of transitions that do not create deadlocks results in a program whose behaviors are a subset of the original program.

The reason for considering this approach is as follows: We anticipate that Step 1 can be made more efficient. This is due to the fact that ignoring realizability constraints reduces the complexity of model repair [2]. We also anticipate that Step 2 would also be made more efficient since it focuses only on removal of transitions.

Contributions of the paper.

- We present an algorithm for adding fault-tolerance to a distributed program using the lazy repair approach. We observe that a pure lazy repair approach does not improve the performance. However, by combining the lazy repair approach with some heuristics from [2] improves the performance beyond that provided by cautious repair.
- We demonstrate the applicability of our approach by considering the problem of addition of fault-tolerance in the context of byzantine agreement and stabilizing chain.
- We note that the lazy repair approach is applicable in several contexts such as synchronous systems, cyber-physical systems, etc.

Organization of the paper. The rest of the paper is organized as follows: In Section II, we provide preliminaries and define the notion of model repair in the absence of realizability constraints. In Section III, we identify the need for realizability constraints. In particular, we identify realizability constraints associated with distributed programs. Section IV reviews cautious repair approach. Section V describes our approach for lazy repair and the motivation behind this approach. In Section VI, we provide our experimental results. In Section VII, we review related work. Finally, in Section VIII, we conclude the paper.

II. PROGRAMS, SPECIFICATION, MODEL REPAIR

In this section, we present the notion of programs, specification, faults, and fault-tolerance. We adopt definitions from previous work in the literature such as [2].
[3], and [4]. Here, we define a program in terms of its states and transitions. Later in Section III-A we explain how we can map a distributed program into the state-transition model defined here.

**Definition 1** (Program). A program $P$ is of the form $\langle S_P, \delta_P \rangle$ where $S_P$ is the state space of program $P$, and $\delta_P \subseteq S_P \times S_P$ is the set of transitions of program $P$.

Next, we provide definitions in the context of such a program.

**Definition 2** (state predicate). A state predicate (denoted as $S$) is a subset of $S_P$.

**Definition 3** (transition predicate). A transition predicate (denoted as $\delta$) is a subset of $S_P \times S_P$.

**Definition 4** (closure). A state predicate $S$ is closed in a transition predicate $\delta$ iff

$\forall (s, s') \in \delta, s \in S \Rightarrow s' \in S$

**Definition 5** (computation). A sequence of states, $\sigma = \langle s_0, s_1, \cdots \rangle$ is a computation of program $P$ iff

- $\forall j, 0 < j < \text{length}(\sigma) : (s_{j-1}, s_j) \in \delta_P$, and
- if $\sigma$ is finite and terminates in state $s_1$ then there does not exist state $s$ such that $(s_1, s) \in \delta_P$.

**Definition 6** (projection). Let $P$ be a program and $S$ be a state predicate. The projection of $\delta_P$ on $S$ (denoted $\delta_P[S]$) is the set of transitions of $\delta_P$ that start in $S$ and end in $S$; i.e.,

$\delta_P[S] = \{(s_0, s_1) | (s_0, s_1) \in \delta_P \wedge (s_0 \in S) \wedge (s_1 \in S)\}$

Following Alpern and Schneider [3], we let the specification of program to consist of a safety specification and a liveness specification.

**Definition 7** (safety specification). A safety specification is a tuple $Sf = (Sf_{bs}, Sf_{bt})$. Thus, a sequence $\langle s_0, s_1, \cdots \rangle$ (denoted by $\sigma$) refines the safety specification iff the following conditions hold,

- $\forall j, 0 \leq j < \text{length}(\sigma) : s_j \notin Sf_{bs}$, and
- $\forall j, 0 < j < \text{length}(\sigma) : (s_{j-1}, s_j) \notin Sf_{bt}$.

The liveness specification, on the other hand, denotes “good thing” must eventually happen during program execution.

**Definition 8** (liveness specification). A liveness specification $Lv$ is defined in terms of lead-to property ($L \rightsquigarrow T$), where both $L$ and $T$ are state predicates. A sequence $\langle s_0, s_1, \cdots \rangle$ (denoted by $\sigma$) refines the liveness specification, $L \rightsquigarrow T$ iff the following condition hold,

- $\forall j, 0 \leq j < \text{length}(\sigma) : (s_j \in L \Rightarrow \exists k, j \leq k < \text{length}(\sigma) : s_k \in T)$.

**Definition 9** (specification). A specification, say $SPEC$ is a tuple $\langle Sf, Lv \rangle$. A sequence $\sigma$ refines $SPEC$ iff it refines $Sf$ and $Lv$.

**Definition 10** (refines). $P$ refines $SPEC$ from $S$ iff the following conditions hold:

- $S$ is closed in $\delta_P$, and
- Every computation of $P$ that starts from a state in $S$ refines $SPEC$.

**Definition 11** (invariant). Given a program $P$, a state predicate $S$, and specification $SPEC$, we say $S$ is an invariant of $P$ iff the following conditions hold,

- $S$ is closed in $\delta_P$;
- Every computation of $P$ that starts in a state, say $s \in S$, satisfies $SPEC$; and
- $S \neq \emptyset$.

**Definition 12** (faults). A class of faults $f$ for $P (= \langle S_P, \delta_P \rangle)$ is a subset of $S_P \times S_P$.

Representing faults as a set of transitions, as suggested by the above definition, is possible notwithstanding the type of the faults (be they stuck-at, crash, fail-stop, omission, timing or Byzantine), the nature of the faults (be they permanent, transient, or intermittent), or the ability of the program to observe the effects of the faults (be they detectable or undetectable).

**Definition 13** (computation in presence of faults). A sequence of states, $\sigma = \langle s_0, s_1, \cdots \rangle$ is a computation of program $P$ in presence of faults $f$ denoted as $P[f\|f]$ iff

- $\forall j, 0 < j < \text{length}(\sigma) : (s_{j-1}, s_j) \in \delta_P \cup f$, and
- $\exists n : n \geq 0 : (\forall j : j > n : (s_{j-1}, s_j) \in \delta_P \cup f$, and
- if $\sigma$ is finite and terminates in state $s_1$ then there does not exist state $s$ such that $(s_1, s) \in \delta_P$.

**Definition 14** (fault-span). $T$ is an $f$-span of $P[f\|f]$ from $S$ iff

- $S \subseteq T$, and
- $T$ is closed in $P[f\|f]$.

**Definition 15** (masking fault-tolerance). $P$ is masking $f$-tolerant to $SPEC (= \langle Sf, Lv \rangle)$ from $S$ iff the following three conditions hold:

- $P$ refines $SPEC$ from $S$, and
- every finite computation of $P[f\|f]$ that starts from $S$ refines $Sf$, and
- there exists $T$ such that (1) $T$ is an $f$-span of $P$ from $S$ and (2) every computation of $P[f\|f]$ that starts from $T$, has state in $S$.

**Problem Statement.** Our goal in the repair problem is to revise a program that satisfies a specification in absence of fault into one that satisfies the specification even in presence of faults. Our goal is to add fault-tolerance and not adding new behaviors to satisfy specification in the absence of faults. Thus, the invariant and the transitions inside it should be the subset of those of the original program. Using the definitions provided in

---

3
this section, we define the repair problem similar to that in [1]:

Given program \( P = (S_P, \delta_P) \), specification \( SPEC \), set of faults \( f \), and state predicate \( S \) such that \( P \) refines \( SPEC \) from \( S \), identify program \( P' = (S_P, \delta_P') \), and state predicate \( S' \) such that:

- \( S' \subseteq S \), and
- \( \delta_P'|S' \subseteq \delta_P|S' \), and
- \( P' \) is masking \( f \)-tolerant to \( SPEC \) from \( S' \).

III. REALIZABILITY CONSTRAINTS

In Section II, we defined the program in terms of its state space and transitions. Hence, a program can be viewed as a graph where each state corresponds to a vertex and each transition corresponds to an edge. The graph of states and transitions is not the only way to represent a program. For example, we can define a program in terms of its variables and actions, or as we will see later in this section, we may define a distributed program in terms of its set variables and set of processes.

The graph representation is feasible for any program that has a finite state space. In other words, we can map other representations into graph of states and transitions if the program has a finite state space. As an illustration consider a program with two variables \( v_1 \) and \( v_2 \) with domain \{0, 1\}. The state space of this program includes four states. We denote each state as \( ij \) where \( i \) is the value of \( v_1 \) and \( j \) is the value of \( v_2 \) in that state. We can also map any program action into a set of transitions between program states. For example, we can map action \((if \ v_0 = 0 \ then \ v_1 = 1)\) into \{(00,10), (01,11)\}.

In order to repair a program, we first map it into the corresponding graph of states and transitions. Then, we revise the graph by adding or removing transitions (i.e., edges). However, in the process of revising the graph, we should be careful, because not every graph can be mapped back into a valid program. In order to map a graph back to a valid program, the revised graph should satisfy a set of constraints that we call the realizability constraints. These constraints are different in different contexts (e.g., distributed programs, synchronous programs, embedded systems, cyber-physical systems). In the rest of this section, we first define distributed program, and then, focus on the realizability constraints in the context of distributed programs.

A. Modeling of Distributed Programs

A distributed program \( P \) consists of a set of variables, and a set of processes. Hence, we can view a distributed program as the tuple \((V_P, \mathcal{P}_P)\), where \( V_P \) is a finite set of variables \( \{v_1, \ldots, v_u\} \), and \( \mathcal{P}_P \) is a finite set of processes \( \{p_1, \ldots, p_n\} \), where \( u \) and \( n \) are positive integers. Each variable \( v_j \) has a finite domain denoted as \( D_j \). Using the domains of variables, we define the state space of a distributed program as follows:

Definition 16 (state space of distributed programs). The state space, \( S_P \) of \( P \) is obtained by assigning each variable in \( V_P \) a value from its respective domain. Hence, \( S_P = D_1 \times D_2 \times \ldots \times D_u \).

We define a process of a distributed program as follows:

Definition 17 (process). A process \( p_j \) of \( P \) is specified by the tuple \((R_j, W_j, \delta_j)\) where \( R_j \subseteq V_P \) is a set of variables \( p_j \) is allowed to read, \( W_j \subseteq R_j \) is a set of variables \( p_j \) is allowed to write, \( \delta_j \subseteq S_P \times S_P \) are transitions of process \( p_j \). Given any two processes \( j \) and \( k \), we require that \( W_j \) and \( W_k \) are disjoint.

In a distributed program, each process can execute asynchronously. Hence, the execution of the program is obtained by some interleaving execution of the processes themselves. Moreover, if the program ever reaches a state where no process can execute its transition then essentially the program stops in that state. We model this by stuttering that state forever. Hence, transitions \( \delta_P \) of \( P \) are defined as follows:

Definition 18 (Transitions of distributed programs). The transitions of distributed program \( P = (V_P, \mathcal{P}_P) \) are

\[
\delta_P = \{(s_0, s_1) \mid \exists p_j \in \mathcal{P}_P : (s_0, s_1) \in \delta_j \lor (s_0 = s_1) \land \forall p_j \in \mathcal{P}_P, s \in S_P : (s_0, s) \not\in \delta_j\}
\]

Using Definition 16 and 18, we can map any distributed program to its corresponding graph of states and transitions defined in Definition 1. Since, we have mapped the distributed program to Definition 1 we can use all definitions provided in Section II for any given distributed program.

B. Realizability Constraints for Distributed Programs

Realizability constraints are the constraints that a graph of states and transitions should satisfy in order to be mapped into a valid program. In the context of distributed programs, realizability constraints are about the read and write restrictions for processes. In particular, as specified in the Definition 17, each process in a distributed program is able to read/write only a subset of variables.

For example, consider a program \( P = (V_P, \mathcal{P}_P) \) where \( V_P = \{v_0, v_1, v_2\} \) with domain \{0, 1\}, and \( \mathcal{P}_P = \{p_j, p_k\} \). Process \( p_j \) can read \( v_0 \) and \( v_1 \) and process \( p_k \) can read \( v_0 \) and \( v_2 \), i.e., \( R_j = \{v_0, v_1\} \) and \( R_k = \{v_0, v_2\} \). Furthermore, process \( p_j \) can write variables \( v_1 \), and process \( p_k \) can write \( v_2 \), i.e., \( W_j = \{v_1\} \) and \( W_k = \{v_2\} \).

Now, consider three different graphs of states and transitions in Figures 3, 4 and 5. We want to see whether they can be mapped to a valid distributed program with restrictions identified above.
In Figure 3, the program consists of exactly one transition, namely \{\{(000,011)\}\}. In this transition, the value of \(v_0\) remains unchanged and the values of both \(v_1\) and \(v_2\) change from 0 to 1. The transition \{\{(000,011)\}\} cannot be a transition of process \(p_j\), because \(p_j\) cannot change \(v_1\). Similarly, it cannot be a transition of process \(p_k\), because \(p_k\) cannot change \(v_1\). Therefore, program in Figure 3 is not realizable. Note that, not only the program in Figure 3, but also any program that includes transition \{\{(000,011)\}\} is not realizable.

In Figure 4, the program consists of a single transition \{\{(000,010)\}\}. This transition does not violate the write restrictions considered above, as it only changes \(v_1\), and process \(p_j\) is allowed to write \(v_1\). This transition corresponds to program ‘if \((v_0 = v_1 = v_2 = 0)\) then change the value of \(v_1\) to 1’. It follows that the program in Figure 4 cannot be realized in the program under consideration.

If we add transition \{\{(001,011)\}\} to the graph in Figure 4, we have the graph in Figure 4 that represents a realizable program. Even though each of \{\{(000,010)\}\} or \{\{(001,011)\}\} by itself violates the read restrictions, as a group, they can be realized without violating the read restrictions. Specifically, we can view this program as a set of transitions obtained by ‘if \(v_0\) and \(v_1\) = 0 then change the value of \(v_1\) to 1’. Thus, a program consisting of these two transitions is realizable.

Now, we formalize the notions of realizability constraints for distributed programs in terms of write and read restrictions:

**Write Restrictions.** Let \(P = \langle V_P, P_P \rangle\) be a program, and \(p_j = \langle R_j, W_j, \delta_j \rangle\) be a process of program \(P\). Then \(\delta_j\) must exclude the following transitions due to inability to write variables not in \(W_j\).

\[
\text{write}(W_j) = \{(s_0, s_1)|\exists v \notin W_j, v(s_0) \neq v(s_1)\}
\]

**Read Restrictions.** Let \(P = \langle V_P, P_P \rangle\) be a program, and \(p_j = \langle R_j, W_j, \delta_j \rangle\) be a process of program \(P\). Let \((s_0, s_1), s_0 \neq s_1\) be a transition in \(\delta_j\). If variable \(v\) is not in \(R_j\), then \(\delta_j\) must include corresponding transitions from all states \(s'_0\) where \(s'_0\) and \(s_0\) differ only in the value of \(v\). Let \((s'_0, s'_1)\) be one such transition. Now, it must be the case that \(s_1\) and \(s'_1\) are identical except for the value of \(v\) and, the value of \(v\) must be identical in \(s'_0\) and \(s'_1\). Thus, each transition \((s_0, s_1)\) in \(\delta_j\) is associated with the following group predicate.

\[
group_j(s_0, s_1) = \{(s'_0, s'_1)|\forall v: v \in R_j : v(s_0) = v(s'_0) \land v(s_1) = v(s'_1)\} \land 
\forall v : v \notin R_j : v(s'_0) = v(s'_1) \land v(s_0) = v(s_1)\}
\]

As an example, given a transition, say \((s_0, s_1) = \{(000,010)\}\), of process \(p_j\) in the previous example, group predicate \(\group_j(s_0, s_1)\) would be \{\{(000,010), (001,011)\}\}.

We also define group of a set of transitions \(\delta\) as follows:

\[
group_j(\delta) = \{(s'_0, s'_1)|\exists (s_0, s_1) : (s_0, s_1) \in \delta: (s'_0, s'_1) \in \group_j(s_0, s_1)\}.
\]

Based on the above read and write restrictions, we can define the realizability of a set of transitions by some process \(p_j\) in program \(P\). Specifically,

**Definition 19** (transition predicate realizability by process). Let \(p_j\) be a process of program \(P\) that is specified by \(\langle R_j, W_j, \delta_j \rangle\). We say that \(\delta_j\) is realizable by process \(p_j\) iff the following conditions hold:

- \(\delta \cap \text{write}(W_j) = \emptyset\). (Does not violate write restriction.)
- \(\forall (s_0, s_1) \in \delta : \group_j(s_0, s_1) \subseteq \delta_j\). (Does not violate read restriction.)

Finally, a program is realizable iff it can be realized by the set of processes in it.

**Definition 20** (transition predicate realizability by program). Let \(P\) be a program specified in terms of its variables \(\langle V_P, P_P \rangle\), where \(P_P = \{p_1, \ldots, p_n\}\). We say that a set of transitions \(\delta_P\) is realizable by program \(P\) iff

- There exists \(\delta_1, \delta_2, \ldots, \delta_n\) such that \(\forall j \in \{1..n\} :: \delta_j\) can be realized by process \(p_j\), and
- The transitions \(\delta_P\) are obtained from \(\delta_1, \delta_2, \ldots, \delta_n\) according to Definition 18.

## IV. CAUTIOUS REPAIR

Before we describe our approach for lazy repair, we identify basic characteristics in the cautious repair approach considered in [2]. Observe that we view the program in terms of its state space and its transitions. Hence, during the repair process, we need to add or remove transitions. In particular, if faults cause the program to reach a state where safety is violated, we need to remove (one or more) transitions on the path that led to violation of the safety property. Alternatively, if the program reaches a state due to faults, it may be necessary to add recovery transitions that restore the program to its legitimate states. Since the problem of adding fault-tolerance with these realizability constraints is NP-complete, the approach in [2] uses heuristics to identify transitions that should be removed or added.
We note that the approach in [2] is cautious because it ensures that the model always satisfies the realizability constraints identified in Section III-B. Specifically, if it removes a transition, then it computes the corresponding group of that transition. If some transition in this group is important, e.g., if it is useful as a recovery transition, then it requires the heuristics to determine if (1) the group should still be removed (2) the group should not be removed but it should identify an alternate transition to be removed, or (3) the decision should be deferred until further analysis. Also, if it adds a transition (e.g., as a possible recovery transition) then we need to add its corresponding group. If some transition in this group is undesirable, heuristics need to identify if (1) the group should still be added, (2) an alternate recovery transition should be considered, or (3) the decision should be deferred until further analysis.

In [2], authors develop heuristics to identify which selection should be made from the above choices. While all the details of these heuristics are outside the scope of the paper, we illustrate one of the heuristics (used subsequently in our algorithm). To illustrate this heuristic, consider the transitions in Figure 5. In this figure, consider the case where the transition (000, 010) is being considered as possible recovery transition. As discussed earlier, to include this transition, we also need to include (001, 011) as these two transitions are grouped together due to read restrictions. Furthermore, assume that (001, 011) violates safety. Thus, in this situation, we need to (1) add this group of transitions and ensure that the state 001 is never reached during execution, (2) ignore the recovery provided by (000, 010), or (3) defer the decision. In this situation, the heuristic in [2] states that we can keep the group if state 001 is not reachable in the original program in the presence of faults. In other words, this heuristic is based on the idea that if a state was unreachable in the original program, it is likely to be unreachable in the repaired program as well.

Since the realizability constraints are always met in each step, this process ensures that at any time, the model being repaired is realizable. However, the cost of each decision can be quite high. The goal of lazy repair is to split this into two steps: In the first step, we allow repair to be performed without realizability constraints and the second step focuses on ensuring realizability constraints.

V. Lazy Repair Algorithm

In this section, we present our two-step approach for adding fault-tolerance via the lazy repair approach. The structure of our program is as shown in Algorithm 1. Section V-A presents the details of the first step (Line 3 in Algorithm 1) where we add fault-tolerance without realizability constraints. And, Section V-B presents the tasks involved in ensuring realizability constraints (Line 9 in Algorithm 1).

A. Step 1: Adding Fault-Tolerance without Realizability Constraints

As we discussed in Section III-B, the read/write restrictions are the realizability constraints for the distributed programs. If we ignore these realizability restrictions, we can effectively repair the program in the polynomial time in the size of program state space by the Add-masking algorithm provided in [1]. Although Add-masking requires only polynomial time, in this step, we improve its execution time further based on the heuristic considered in Section IV. In particular, we tailor Add-masking to only focus on the subset of the state space that is reached by the fault-intolerant program in the presence of faults. Next, we briefly describe this algorithm.

The Add-masking algorithm first calculates the set $ms$ that identifies states from where executing one or more fault transitions leads to the violation of the given safety specification $S_f$. Subsequently, it calculates set $mt$ that identifies transitions that cannot be executed by the desired fault-tolerant program. This set includes any transition that violates specification, or reaches a state in $ms$. In order to avoid violation of specification, the algorithm removes any state in $ms$ from the program invariant, and any transition in $mt$ from the set of transitions of the revised program. Note that removing states and transitions from the program can create deadlock states, i.e., states from where there are no outgoing transitions. Hence, this algorithm recursively removes the deadlock states from the invariant. The resulting predicate is the first guess, say $S_1$, for the invariant of the desired fault-tolerant program. Likewise, it constructs the first guess, say $T_1$, for the fault-span of the desired fault-tolerant program. To do so, it computes the states reachable in the execution of the fault-intolerant program in the presence of faults. It then removes states from $ms$ from this set.

Next, the algorithm goes into a loop to determine if any further states need to be removed from $S_1$ or $T_1$. In this loop, it first constructs a program that identifies all possible available transitions, i.e., transitions that preserve the closure property of invariant and fault-span and are not included in set $mt$, i.e., transitions that should not be included. Then, it iteratively

- removes states from $T_1$ if recovery to $S_1$ from those states is impossible,
- removes states from $T_1$ if occurrence of faults from those states can cause the program to reach a state outside $T_1$,
- removes states from $S_1$ if they have been removed from $T_1$ (to ensure that invariant is always a subset of fault-span),
- removes states from $S_1$ if they have become deadlock states.

In [1], authors have shown that any state removed
from $S_1$ and $T_1$ in this fashion must be removed. Hence, we can use this feature in Step 2 so that we do not cause new states to be reachable during the second step.

When this iterative computation terminates, i.e., no new state can be removed from $S_1$ and $T_1$, the resulting program guarantees closure properties of $S_1$ (in program transitions) and $T_1$ (in program and fault transitions). Furthermore, there is a path from every state in $T_1$ to a state in $S_1$. However, there may be some cycles in $T_1 - S_1$ that cause the program to stay outside the invariant forever. Hence, Add-masking breaks such cycles by removing transitions. Although there is a potential for removing unnecessary transitions in this step, we pursue this approach to avoid potential exponential complexity if one were to consider all possible approaches for breaking cycles.

B. Step 2: Enforcing Realizability Constraints

In the second step, we revise the program resulting from the Add-masking algorithm, to satisfy the realizability constraints. Regarding the write restrictions, we have to remove any transition that violates write restriction. Regarding the read restriction, as we explained in Section III-B, each transition is associated with a group of transitions. In order to satisfy the read restrictions, we guarantee that for each transition, all transitions in its group exist in the set of program transitions. When a transition of a group is missing, two cases are possible: 1) the missing transition starts from a state outside the fault-span, 2) the missing transitions starts from a state inside the fault-span. In the first case, we can simply add the transition to the program, as the starting state of that transition never reached. Thus, including it does not affect the correctness. In the second case, we cannot add the transition, as its removal in step one, was for a "good reason". Thus, to satisfy the read restriction, we remove the whole group from the program.

With the above explanation, the algorithm for the second step is shown in the Algorithm 2. This algorithm receives masking fault-tolerant program and fault-span $T$ calculated by the Add-masking algorithm. In Line 1, from any state outside $T$ we add transitions to all states in the state space. We add these transitions to include any possible missing transition outside the fault-span (case 1 mentioned in the previous paragraph). We assign the resulting set of transitions to $\delta$. In the second line, we initialize the set of transitions of the final revised program, $\delta_p$, to the empty set.

In the loop in Lines 3 - 24, we expand $\delta_p$ by adding transitions that according to realizability constraints we are all allowed to include them. In Line 4, we set $NW_j$ to the set of all variables that process $p_j$ is not allowed to write. In Line 5, we set $\Delta_j$ to the set of transitions of process $p_j$ as the set of all transitions inside $\delta$ that does not change any variable in $NW_j$. This excludes any transition that violates write restrictions. We set $\delta_j$ to an empty set, then, we expand $\delta_j$ in loop in Lines 7 - 22, by adding group of transitions from $\Delta_j$ to $\delta_j$.

In Line 8, we choose one of the transitions of $\Delta_j$, say $(s_0, s_1)$, and assign group $p_j(s_0, s_1)$ to $G$ in Line 9. In Line 10, we check if $\Delta_j$ includes all transitions in $G$. If some of the transitions of $G$ are missing, we remove transitions of $G$ from $\Delta_j$ in Line 11. Otherwise, we add $G$ to the set of transitions of the final program $\delta_j$. However, before adding $G$, we try to expand it to include more transitions, because in this way we can decrease the number of iterations needed in the loop in Lines 7 - 22.

In order to expand $G$, the algorithm invokes Expand-Group. We illustrate the role of ExpandGroup with a simple example, where the process under consideration can read 3 variables, say $v_1, v_2$ and $v_3$. Thus, the group of transitions identified on Line 8, corresponds to some action of the form ‘if $(v_1 = c_1 \land v_2 = c_2 \land v_3 = c_3)$ then update a subset of variables from $W_j$’, where $c_1, c_2$ and $c_3$ are some constants. The goal of ExpandGroup is to enlarge this group by removing one or more variables from the condition checked in the above if-statement. For example, if we remove $v_3$ then this would correspond to ‘if $(v_1 = c_1 \land v_2 = c_2)$ then update a subset of variables from $W_j$ in the same fashion as before’. Observe that this statement corresponds to additional set of transitions than the original set. If this expanded group of transitions is included in $\Delta_j$ then we include the larger group in $\delta_j$ and remove it from $\Delta_j$ (Line 19 and 20). In turn, this obviates the need to consider corresponding transitions of the form ‘if $(v_1 = c_1 \land v_2 = c_2)$ then update a subset of variables from $W_j$’, where $c_3$ is another value from domain of $v_3$. In either case, subsequently, we attempt to eliminate $v_3$ in a similar manner. When successful, this eliminates a large number (exponential in the number of variables where this strategy is successful) of transitions from $\Delta_j$. (Since we are using BDDs where the program transitions are represented in terms of Boolean formula, identifying this expansion can be achieved efficiently. Hence, even if this is approach is not successful for some variables, the amount spent in identifying the expanded group is very small.)

Finally, after finishing loop in Lines 7 - 22, we add $\delta_j$ to $\delta_p$, in Line 23.

Note that in the second step (Algorithm 2), we may remove some of the transitions of the program resulting from the first step (Add-masking algorithm). That may create deadlock states in the program. In this case, we make those state unreachable starting from the invariant. Therefore, we add any transition to a deadlock state to the set of bad transitions of the specification. We also add any transition to the outside of the fault-span resulting from Add-masking algorithm to the set of bad transitions. Then, we run both steps again. Thus, the overall algorithm for adding masking fault-tolerance to


<table>
<thead>
<tr>
<th>Algorithm 1 Adding Masking Fault-tolerance to a Distributed Program via Lazy Repair</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> A program $\mathbb{P} = \langle V_\mathbb{P}, P_\mathbb{P} \rangle$ set of legitimate states $S$, faults $f$, and safety specification $S_f$.</td>
</tr>
<tr>
<td><strong>Output:</strong> Realizable program $\mathbb{P}'$, and $S'$ such that $\mathbb{P}'$ is masking fault-tolerant from $S'$.</td>
</tr>
<tr>
<td>1: $S' = S$</td>
</tr>
<tr>
<td>2: <strong>repeat</strong></td>
</tr>
<tr>
<td>3: $\mathbb{P}', S', T' = \text{Add-Masking}(\mathbb{P}, S', S, f)$</td>
</tr>
<tr>
<td>4: // Add masking fault-tolerance</td>
</tr>
<tr>
<td>5: // without realizability constraints</td>
</tr>
<tr>
<td>6: <strong>if</strong> $S' = \phi$ <strong>then</strong></td>
</tr>
<tr>
<td>7: declare failure to add fault-tolerance</td>
</tr>
<tr>
<td>8: <strong>end if</strong></td>
</tr>
<tr>
<td>9: $\mathbb{P}' = \text{Algorithm 2}(\mathbb{P}', T')$</td>
</tr>
<tr>
<td>10: $DL = {s_0</td>
</tr>
<tr>
<td>11: $S_f = S_f \cup {(s_0,s_1)</td>
</tr>
<tr>
<td>12: <strong>until</strong> $DL \neq \phi$</td>
</tr>
<tr>
<td>13: <strong>return</strong> $\mathbb{P}', S'$</td>
</tr>
</tbody>
</table>

A distributed program using the lazy repair approach is as shown in Algorithm 1.

The soundness of the Algorithm 1 is resulted by following two theorems. For the reason of space, the proofs of these theorems are provided in the [5].

**Theorem 1.** The program $\mathbb{P}'$ resulting from the Algorithm 1 is a masking fault-tolerant program to the specification from $S'$.

**Theorem 2.** The set of transition $\delta_{\mathbb{P}'}$ resulting from Algorithm 1 is realizable by program $\mathbb{P}'$.


<table>
<thead>
<tr>
<th>Algorithm 2 Constructing Distributed Program</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> A program $\mathbb{P} = \langle V_\mathbb{P}, P_\mathbb{P} \rangle$ and state predicate $T$.</td>
</tr>
<tr>
<td><strong>Output:</strong> Realizable program $\mathbb{P}'$.</td>
</tr>
<tr>
<td>1: $\delta := \delta_{\mathbb{P}} \cup {(s,s)</td>
</tr>
<tr>
<td>2: $\delta_{\mathbb{P}'} := \emptyset$</td>
</tr>
<tr>
<td>3: <strong>for all</strong> $p_j \in P_\mathbb{P}$ <strong>do</strong></td>
</tr>
<tr>
<td>4: $NW_j := {v</td>
</tr>
<tr>
<td>5: $\Delta_j := \delta \cap {(s, s')</td>
</tr>
<tr>
<td>6: $\delta_j := \emptyset$</td>
</tr>
<tr>
<td>7: <strong>while</strong> $\text{group}<em>j(\Delta_j) \not\subseteq \delta</em>{\mathbb{P}'}$ <strong>do</strong></td>
</tr>
<tr>
<td>8: $(s_0, s_1) := \text{choose one transition from } \Delta_j$</td>
</tr>
<tr>
<td>9: $G := \text{group}_j(s_0, s_1)$</td>
</tr>
<tr>
<td>10: <strong>if</strong> $G \not\subseteq \Delta_j$ <strong>then</strong></td>
</tr>
<tr>
<td>11: $\Delta_j = \Delta_j - G$</td>
</tr>
<tr>
<td>12: <strong>else</strong></td>
</tr>
<tr>
<td>13: <strong>for all</strong> $v \in (R_j - W_j)$ <strong>do</strong></td>
</tr>
<tr>
<td>14: $G' := \text{ExpandGroup}(v, G)$</td>
</tr>
<tr>
<td>15: <strong>if</strong> $G' \subseteq \Delta_j$ <strong>then</strong></td>
</tr>
<tr>
<td>16: $G := G'$</td>
</tr>
<tr>
<td>17: <strong>end if</strong></td>
</tr>
<tr>
<td>18: <strong>end for</strong></td>
</tr>
<tr>
<td>19: $\delta_j := \delta_j \cup G$</td>
</tr>
<tr>
<td>20: $\Delta_j = \Delta_j - G$</td>
</tr>
<tr>
<td>21: <strong>end if</strong></td>
</tr>
<tr>
<td>22: <strong>end while</strong></td>
</tr>
<tr>
<td>23: $\delta_{\mathbb{P}'} = \delta_{\mathbb{P}'} \cup \delta_j$</td>
</tr>
<tr>
<td>24: <strong>end for</strong></td>
</tr>
<tr>
<td>25: <strong>return</strong> $\mathbb{P}'$</td>
</tr>
</tbody>
</table>

From each other to finalize their decision. The protocol is subjected to a byzantine fault that permits the affected process to send arbitrary decision to other processes. It is assumed that at most one process is byzantine. The goal of the protocol is to allow each non-general process to identify a decision subject to the requirements of validity, agreement and termination. Of these, validity requires that if the general is not byzantine (malicious) and sends the same decision to all non-generals then the final decision of every non-byzantine non-general is the same as that of the general. The agreement requires that the final decision of any two non-byzantine non-generals must be same. Finally, termination requires that each process finalizes its decision.

We model the program as follows: The program consists of three variables for each non-general: $b$ (domain $\{\text{true}, \text{false}\}$ to denote whether the process is byzantine), $d$ (domain $\{0, 1, \bot\}$ to denote the decision), and $f$ (domain $\{\text{true}, \text{false}\}$ to denote whether the decision is finalized). It also contains two variables for general: $b$ (domain $\{\text{true}, \text{false}\}$ to denote whether the general is byzantine) and $d$ (domain $\{0, 1\}$ to denote the decision). Thus, the set $V_\mathbb{P}$ defined in Section III-A for this system is instantiated to be $\{b, g, d, g, b, j, d, j, f, j, b, k, d, k, f, k, b, l, d, l, f, l\}$.
Given these variables, it is straightforward to identify the state space of the resulting program.

The read/write restrictions are as follows: each non-general is allowed to read all the decision variables and the $b$ and $f$ value of itself. Thus, variables $j$ is allowed to write are $\{b,j,d,j,f,j,d,g,d,k,d,l\}$. The variables $j$ is allowed to write are $\{d,j,f,j\}$.

The actions of the fault-intolerant program are modeled as follows: Initially, when $d,j$ equals $\bot$, $j$ copies the decision of the general. After copying the decision, it can finalize that decision. Thus, the transitions of process $j$ are represented by the following actions (Similar actions are added for processes $k$ and $l$.

$$d,j = \bot \land f,j = 0 \quad \rightarrow \quad d,j = d,g$$
$$d,j \neq \bot \land f,j = 0 \quad \rightarrow \quad f,j = 1$$

Finally, the faults allow one of the processes to become Byzantine. And, the Byzantine process can change its decision arbitrarily. Thus, the faults that can affect process $j$ are represented by transitions corresponding to the following actions (Similar actions are added for $k$, $l$ and $g$).

$$-b.j \land -b,k \land -b,l \land b,g \rightarrow b.j = 1$$
$$b.j \rightarrow d.j = 0|1$$

We use this model to compare the time required for synthesis with the algorithm in [2]. The resulting comparison is as shown in Table I. As shown in this table, the lazy repair algorithm was able to add tolerance to Byzantine fault in an expeditious manner.

### Table I
#### Experimental Results for Byzantine Agreement

<table>
<thead>
<tr>
<th></th>
<th>Reachable States</th>
<th>Cautious Repair</th>
<th>Time for Step 1</th>
<th>Time for Step 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$BAFS^4$</td>
<td>$10^3$</td>
<td>$6s$</td>
<td>$&lt; 1s$</td>
<td>$&lt; 1s$</td>
</tr>
<tr>
<td>$BAFS^{28}$</td>
<td>$10^8$</td>
<td>$128s$</td>
<td>$4s$</td>
<td>$1s$</td>
</tr>
<tr>
<td>$BAFS^{12}$</td>
<td>$10^9$</td>
<td>$1387s$</td>
<td>$167s$</td>
<td>$10s$</td>
</tr>
<tr>
<td>$BAFS^{20}$</td>
<td>$10^{16}$</td>
<td>$20348s$</td>
<td>$385s$</td>
<td>$25s$</td>
</tr>
</tbody>
</table>

Hence, we present the results for the lazy repair approach only.

### Table II
#### Experimental Results for Byzantine Agreement with Fault-Stop Faults

<table>
<thead>
<tr>
<th></th>
<th>Reachable States</th>
<th>Time for Step 1</th>
<th>Time for Step 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Sc^{20}$</td>
<td>$10^{19}$</td>
<td>$2s$</td>
<td>$&lt; 1s$</td>
</tr>
<tr>
<td>$Sc^{25}$</td>
<td>$10^{22}$</td>
<td>$11s$</td>
<td>$&lt; 1s$</td>
</tr>
<tr>
<td>$Sc^{26}$</td>
<td>$10^{26}$</td>
<td>$18s$</td>
<td>$1s$</td>
</tr>
<tr>
<td>$Sc^{27}$</td>
<td>$10^{30}$</td>
<td>$27s$</td>
<td>$1s$</td>
</tr>
<tr>
<td>$Sc^{28}$</td>
<td>$10^{30}$</td>
<td>$52s$</td>
<td>$1s$</td>
</tr>
<tr>
<td>$Sc^{29}$</td>
<td>$10^{30}$</td>
<td>$220s$</td>
<td>$1s$</td>
</tr>
<tr>
<td>$Sc^{30}$</td>
<td>$10^{30}$</td>
<td>$889s$</td>
<td>$1s$</td>
</tr>
</tbody>
</table>

From these case studies, it follows that the lazy repair approach with the heuristic associated with states reached in the presence of faults in the fault-intolerant program it was possible to improve the performance of lazy repair.

### VII. Related Work

This paper focuses on addition of fault-tolerance via lazy repair where we separate the concerns of adding fault-tolerance and enforcing realizability constraints. Our approach differs from [2] in that in [2], both these concerns are handled simultaneously. Model repair has been studied in the context CTL properties [8], [9]. A game theoretic approach for model repair for LTL properties has been considered in [10]. In the context of parallel and distributed systems, model repair for adding UNITY properties [11] has been considered in [12]. In some of these instances [10], [8], there are no relevant realizability constraints and in others they have been addressed via the approach of cautious repair where every repair to the program results in necessary changes to enforce realizability constraints. The effect of these realizability constraints is typically an increase in complexity (e.g., $P$ to $NP$-complete). This effect has been shown in the context of adding fault-tolerance, safety properties and liveness properties [1], [12].

A tool for automated addition of fault-tolerance to distributed programs is presented in [2]. This work utilizes BDD based techniques to enable synthesis of programs with state space exceeding $10^{100}$. This work
uses the approach for cautious repair. In this work, we demonstrated that the use of some heuristics from this work and lazy repair approach can be used to improve the time for repair substantially.

VIII. Conclusion

In this paper, we focused on the problem of adding fault-tolerance using the approach of lazy repair. The lazy repair approach was motivated by the observation that performing model repair requires us to (1) ensure that the repaired model satisfies the property of interest, and (2) ensure that the repaired model satisfies the realizability constraints so that it can be implemented in the underlying system.

The lazy repair approach focused on separating these two requirements. Specifically, in the first step, it focused on adding fault-tolerance while ignoring the realizability constraints. The advantage of this step was that this problem can be solved in polynomial time whereas adding fault-tolerance with realizability constraints is NP-complete. Another advantage of this step was that it prevented the program from reaching a state, say $s$, iff reaching that state would make it impossible to add fault-tolerance. And, if it prevented the program from using a transition then it was very likely that it should have been removed from the fault-tolerant program. These properties and a potential efficient implementation predicated an efficient implementation of the first step. The second step focused only on removing behaviors to enforce realizability constraints. This was due to the fact that transitions removed during first step were good candidates to do so.

We found that lazy repair improves the performance of adding fault-tolerance when combined with some heuristics. In particular, by limiting the search space to states that may be reached by executing the fault-intolerant program in the presence of faults, it was possible to find the solution efficiently.

An advantage of lazy repair approach is that it enables reuse of repair algorithms. Specifically, since the first step ignores realizability constraints, it could be used in systems with different realizability constraints. As an illustration, [13] considers execution in synchronous semantics. Such execution is motivated by barrier controlled computation where the program is executed as follows: First, all nodes read the relevant data and wait for a barrier. After, all nodes have read the data, they update the data that needs to be changed and wait for a barrier. Then, this process is repeated forever. Such synchronous execution is critical for several parallel computing applications. Lazy repair has been shown to be successful [13] for synchronous semantics. To the best of our knowledge, there is no known algorithm for cautious repair in synchronous semantics.

One future work in this area is to identify additional tradeoffs while adding properties. Specifically, in our work, we used a heuristic that was developed for cautious repair to improve the performance of lazy repair. Future work in this context involves development and evaluation of different heuristics for lazy repair.

References


