Noname manuscript No.
(will be inserted by the editor)

The Complexity of Automated Addition of Fault-tolerance Without Explicit Legitimate States

Fuad Abujarad · Yiyan Lin · Borzoo Bonakdarpour · Sandeep S. Kulkarni

the date of receipt and acceptance should be inserted later

Abstract Existing algorithms for automated *model repair* for adding fault-tolerance to fault-intolerant models incur an impediment that designers have to identify the set of *legitimate states* of the original model. This set determines states from where the original model meets its specification in the absence of faults. Experience suggests that of the inputs required for model repair, identifying such legitimate states is the most difficult. In this paper, we consider the problem of automated model repair for adding fault-tolerance where legitimate states are not explicitly given as input. We show that without this input, in some instances, the complexity of model repair increases substantially (from polynomial-time to NP-complete). In spite of this increase, we find that this formulation is relatively complete; i.e., if it was possible to perform model repair *with* explicit legitimate states, then it is also possible to do so *without* the explicit identification of the legitimate states. Finally, we show that if the problem of model repair can be solved with explicit legitimate states, then the increased cost of solving it without explicit legitimate states is very small.

In summary, the results in this paper identify instances of automated addition of fault-tolerance, where the explicit knowledge of legitimate state is beneficial and where it is not very crucial.

Keywords Model repair, program synthesis, fault-tolerance, automated formal methods.

This work was partially sponsored by Canada NSERC Discovery Grant 418396-2012 and NSERC Strategic Grant 430575-2012, US Air Force Contract FA9550-10-1-0178, and NSF CNS 0914913.

Fuad Abujarad
Yale University
Department of Emergency Medicine
464 Congress Ave, Ste 260
New Haven, CT 06519, USA

Borzoo Bonakdarpour
Department of Computing and Software
McMaster University
1280 Main Street West
Hamilton ON, Canada L8S 4L7
E-mail: borzoo@McMaster.ca

Yiyan Lin and Sandeep Kulkarni
Department of Computer Science and Engineering
3115 Engineering Building
East Lansing, MI, 48824, USA
E-mail: {linyiyan, sandeep}@cse.msu.edu

1 Introduction

In 2009, Toyota, one of the leading companies in the automotive industry, issued a recall for almost 3.8 million vehicles. The problem was a sticking gas pedal that can cause the vehicle to suddenly accelerate. To solve this problem Toyota added a shim to increase the tensions on the spring that would prevent the pedal from sticking. Essentially, in this case, Toyota attempted to solve the problem via *fault prevention*. Another suggested technique to handle this problem is by introducing brake override where pressing the gas pedal and the brake simultaneously will cause the car to ignore the gas pedal input. Essentially, this technique is an instance of *fault-tolerance*, where the system cannot prevent occurrence of faults but ensures appropriate behavior even if faults occur. Clearly, this solution requires modification to some software (specifically, the software for the Electronic Control Model). Thus, this is also an instance of *model/program repair* where an existing model/program needs to be repaired, so that it satisfies additional properties (e.g., fault-tolerance).

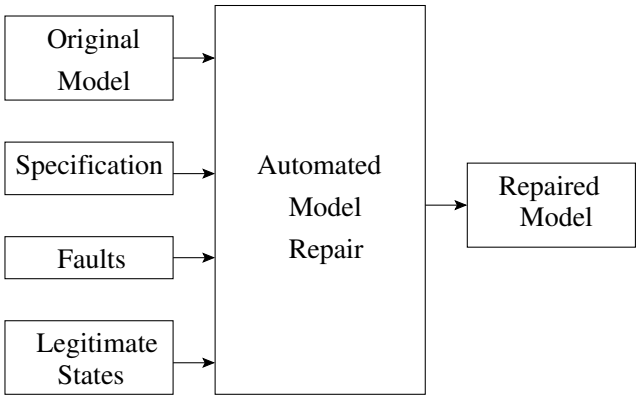

Fig. 1 Model Repair with Explicit Legitimate States.

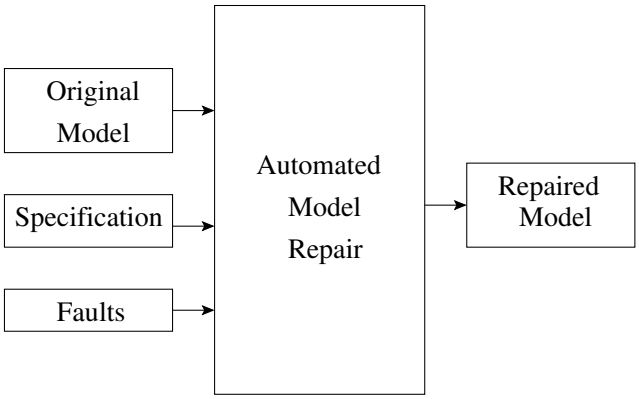

Fig. 2 Model Repair *without* Explicit Legitimate States.

As one can imagine there are several instances where one needs to repair an existing model. In particular, model repair is required to account for bug fixes, newly discovered faults, or changes in the environment. Quite often, model repair is done manually and, hence, create several concerns. For one, it requires a significant effort and resources. Also, the repaired model may violate other properties that the original model provided. Consequently, this approach entails that the correctness of the new model must be re-verified and re-tested to ensure that it still preserves the old properties in addition to the new properties.

Another approach for revising existing models/programs is through *automated model repair*. The goal of the automated model repair is to automatically repair an existing model to generate a new model that is correct by construction [10,29]. Such model will also preserve the existing model properties and satisfy new properties. In its basic form, the problem of automated model repair (also known as incremental synthesis) focuses on modifying an existing model, say p, into a new model, say p'. It is required that p' satisfies the new property of interest. Additionally, p' continues to satisfy existing properties of p using the same transitions that p used.

One specific property of interest where model repair can be applied is *fault-tolerance*. Roughly speaking, fault-tolerance is referred to the property where a model respects its specification in the absence and presence of faults. Thus, applying model repair in order to *add* fault-tolerance to a fault-intolerant model is referred to the process of transforming the intolerant model into a fault-tolerant model that preserves the intolerant model's specification as well as ensuring meeting the specification in the presence of faults.

Current approaches for automated model repair for adding fault-tolerance [16,15,30,36] describe the model as an abstract program. They require the designer to specify (1) the existing abstract program that is correct in the absence of faults, (2) the specification, (3) the set of faults that have to be tolerated, and (4) the set of *legitimate states*, from where the existing program satisfies its specification (cf. Figure 1). We refer to this problem as *the problem of model repair with explicit legitimate states*.

Of the aforementioned four inputs, the first three are easy to identify and/or unavoidable. For example, one is expected to utilize model repair only if they have an existing model that fails to satisfy a required property. Thus, if model repair is applied in the context of newly identified faults, original model and faults are already available. Likewise, specification identifies what the model is supposed to do. Clearly, requiring it is unavoidable.

Our experience in this context shows that while identifying the other three arguments is often straightforward, identifying precise legitimate states requires significant effort. With this motivation, in this paper, we focus on the problem of model repair in the context of addition of fault-tolerance where the input only consists of the fault-intolerant program, faults and the specification; i.e., the legitimate states are not given as input. We call this problem as *the problem of model repair without explicit legitimate states* (cf. Figure 2).

In this context, the following questions need to be addressed for such a new formulation:

Q. 1 Is the new formulation relatively complete? (i.e., if it is possible to perform model repair using the problem formulation in Figure 1, is it guaranteed that it could be solved using the formulation in Figure 2.)
An affirmative answer to this question will indicate that reduction of designers' burden does not affect the solvability of the corresponding problem.

Q. 2 Is the complexity of both formulations in the same class? (By same class, we mean polynomial-time re-

ducibility, where complexity is computed in the size of state space.)
An affirmative answer to this question will indicate that the reduction in the designers' burden does not significantly affect the complexity.

Q. 3 Is the increased time cost, if any, small comparable to the overall cost of program repair?
While Question 2 focuses on qualitative complexity, assuming that the answer is affirmative, Question 3 will address the quantitative change in complexity.

The contributions of the paper are as follows:

- We show that the answer to Q. 1 is affirmative (cf. Theorem 1).
- Regarding Q.2, we introduce two versions of the problem (*partial* repair and *total repair*)[1]:
  - We show that the answer to Q.2 is negative for partial repair. Specifically,
    - We show that there are several instances where the problem of partial repair is NP-complete when legitimate states are not specified although it is in polynomial-time when they are.
    - We show that if the specification of the input program is a safety specification and *one* liveness property[2], then the problem can be solved in polynomial-time.
    - We show that the conditions for polynomial solvability are tight; i.e., permitting two leads-to properties makes the problem NP-complete. And, generalizing the safety specification slightly to include conditional safety specification also makes the problem NP-complete.
  - We show that the answer to Q.2 is affirmative for total repair.
  - We show that there is a class of problems where partial repair can be achieved in polynomial time. This class includes all instances where it is possible to successfully repair a program based on the formulation in Figure 1.
- Regarding Q. 3, we show that for instances where the answer to the question in Figure 1 is affirmative, the extra computation cost of solving the problem using an approach in Figure 2 is small.

Organization The rest of the paper is organized as follows: We define the notions of programs, specifications, and fault-tolerance in Section 2. In Section 3, we formally state the problem of automated model repair for adding fault-tolerance. In Sections, 4 – 7, we answer the above three questions respectively. We discuss related work in Section 8 and conclude in Section 9.

[1] In total repair, the repaired model is expected to behave the same as the original model in the absence of faults. In partial repair, the set of behaviors of the repaired model is a subset of the set of behaviors of the original model in the absence of faults.

[2] We focus on *leads-to* properties, where holding a predicate must eventually be followed meeting another predicate.

2 Programs, Specifications, and Fault-tolerance

In this section, we formally define programs, specifications, faults and fault-tolerance. A subset of these definitions is based on the ones given by Arora and Gouda [4].

2.1 Programs and Specifications

Since we focus on the design of programs, we specify the finite state space of the program in terms of its variables. Each variable is associated with its domain. A *state* of the program is obtained by assigning each of its variables a value from the respective finite domain. The state space of the program is the finite set of all states. Thus, a *program* p is a tuple $\langle S_p, \delta_p \rangle$ where S_p is the program state space, and δ_p is a subset of $S_p \times S_p$ which identifies the program *transitions*.

A *state predicate*, say S, of $p(= \langle S_p, \delta_p \rangle)$ is any subset of S_p. Since a state predicate can be characterized by the set of all states in which its Boolean expression is true, we use sets of states and state predicates interchangeably. Thus, intersection, union, and complementation of sets are the same as the conjunction, disjunction, and negation of the respective state predicates. We say that the state predicate S is *closed* in p iff $(\forall s_0, s_1 :: s_0 \in S \wedge (s_0, s_1) \in \delta_p \;\; \Rightarrow \;\; s_1 \in S)$. In other words, S is closed in p iff every transition of p that begins in S also ends in S. We also use the term $p|S$ to denote the set of transitions $\{(s_0, s_1) \mid s_0 \in S \wedge s_1 \in S \wedge (s_0, s_1) \in \delta_p\}$. In other words, $p|S$ denotes the set of transitions of p that begin and end in S.

Let $\sigma = \langle s_0, s_1, ... \rangle$ be a sequence of states, then σ is a *computation* of $p(= \langle S_p, \delta_p \rangle)$ iff the following two conditions are met:

- $\forall j : 0 < j < length(\sigma) : (s_{j-1}, s_j) \in \delta_p$, where $length(\sigma)$ is the number of states in σ,
- if σ is finite and the last state in σ is s_l then there does not exist state s such that $(s_l, s) \in \delta_p$.

Notation. We call δ_p the transitions of p. When it is clear from context, we use p and δ_p interchangeably.

The *safety* specification, Sf_p, for program p is specified in terms of a set of bad states, $SPEC_{bs}$, and a set

of bad transitions $SPEC_{bt}$. Thus, σ=$\langle s_0, s_1, ...\rangle$ *satisfies* the safety specification of p iff the following two conditions are met.

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

Let $\mathcal{F}$ and $\mathcal{T}$ be state predicates, then the *liveness* specification, Lv_p, of program p is specified in terms of one or more *leads-to* properties of the form $\mathcal{F} \rightsquigarrow \mathcal{T}$. A sequence $\sigma = \langle s_0, s_1, ...\rangle$ satisfies $\mathcal{F} \rightsquigarrow \mathcal{T}$ iff $\forall j : (\mathcal{F}$ is true in $s_j \Rightarrow \exists k : j \leq k < length(\sigma) : \mathcal{T}$ is true in $s_k)$. We assume that $\mathcal{F} \cap \mathcal{T} = \{\}$. If not, we can replace the property by $((\mathcal{F} - \mathcal{T}) \rightsquigarrow \mathcal{T})$.

A *specification*, say $SPEC$ is a tuple $\langle Sf_p\ ,\ Lv_p\rangle$, where Sf_p is a safety specification and Lv_p is a liveness specification. A sequence σ *satisfies* $SPEC$ iff it satisfies Sf_p and Lv_p. Hence, for brevity, we say that the program specification is an intersection of a safety specification and a liveness specification.

Let $I \subseteq S_p$ and $I \neq \{\}$. We say that a program p satisfies $SPEC$ from I iff the following two conditions are satisfied:

- I is closed in p, and
- every computation of p that starts from a state in I satisfies $SPEC$.

If p satisfies $SPEC$ from I, then we say that I is a *legitimate state predicate* of p for $SPEC$. We use the term "legitimate state predicate" and the corresponding "set of legitimate states" interchangeably.

Assumption 2.1 : For simplicity of subsequent definitions, if p satisfies $SPEC$ from I, we assume that p includes at least one transition from every state in I. If p does not include a transition from state s then we add the transition (s, s) to p. Note that this assumption is not restrictive in any way. It simplifies subsequent definitions, as one does not have to model terminating computations explicitly.

2.2 Faults and Fault-tolerance

The *faults*, say f, that a program is subject to are systematically represented by transitions. Based on the classification of faults from [8], this representation suffices for physical faults, process faults, message faults, and improper initialization. It is not intended for program bugs (e.g., buffer overflow). However, if such bugs exhibit behavior such as component crash, it can be modeled using this approach. As an example, for the case considered in the Introduction, a sticky gas pedal can be modeled by a variable *pedal.stuck*; when this variable is false the gas pedal behaves normally. But a fault can set it to true thereby preventing the gas pedal from changing its status. Thus, a fault for $p(= \langle S_p, \delta_p\rangle)$ is a subset of $S_p \times S_p$.

We use '$p[]f$' to mean 'p in the presence of f'. The transitions of $p[]f$ are obtained by taking the union of the transitions of p and the transitions of f. Just as we defined computations of a program in Section 2.1, we define the notion of program computations in the presence of faults. In particular, a sequence of states $\sigma = \langle s_0, s_1, ...\rangle$ is a computation of $p[]f$ (i.e., a computation of $p(= \langle S_p, \delta_p\rangle)$ in the presence of f) iff the following three conditions are satisfied:

- $\forall j : 0 < j < length(\sigma) : (s_{j-1}, s_j) \in (\delta_p \cup f)$, and
- if $\langle s_0, s_1, ...\rangle$ is finite and terminates in state s_l then there does not exist state s such that $(s_l, s) \in \delta_p$,
- if σ is infinite then $\exists n : \forall j > n : (s_{j-1}, s_j) \in \delta_p$

Thus, if σ is a computation of p in the presence of f then in each step of σ, either a transition of p occurs or a transition of f occurs. Additionally, σ is finite only if it reaches a state from where the program has no outgoing transition. And, if σ is infinite then σ has a suffix where only program transitions execute. We note that the last requirement can be relaxed to require that σ has a sufficiently long subsequence where only program transitions execute. However, to avoid details such as the length of the subsequence, we require that σ has a suffix where only program transitions execute. (This assumption is not required for failsafe fault-tolerance described later in this section.)

We use f-span (*fault-span*) to identify the set of states reachable by $p[]f$. In particular, a predicate T is an f-span of p from I iff $I \Rightarrow T$ and $(\forall(s_0, s_1) : (s_0, s_1) \in p[]f\ :\ (s_0 \in T\ \Rightarrow\ s_1 \in T))$. Thus, at each state where I of p is true, f-span T of p from I is also true. Also, T, like I, is also closed in p. Moreover, if any action in f is executed in a state where T is true, the resulting state is also one where T is true. It follows that for all computations of p that start at states where I is true, T is a boundary in the state space of p up to which (but not beyond which) the state of p may be perturbed by the occurrence of the actions in f.

Fault-Tolerance. In the absence of faults, a program, p, satisfies its specification and remains in its legitimate states. In the presence of faults, it may be perturbed to a state outside its legitimate states. By definition, when the program is perturbed by faults, its state will be one in the corresponding f-span. From such a state, it is desired that p does not violate its safety specification. Furthermore, p recovers to its legitimate states from where p subsequently satisfies both its safety and liveness specification.

Based on this intuition, we now define what it means for a program to be *masking* fault-tolerant. Let Sf_p and Lv_p be the safety and liveness specifications for program p. We say that p is masking fault-tolerant to Sf_p and Lv_p from I iff the following two conditions hold:

1. p satisfies Sf_p and Lv_p from I.
2. $\exists\, T$ ::
   (a) T is an f-span of p from I.
   (b) $p[]f$ satisfies Sf_p from T.
   (c) Every computation of $p[]f$ that starts from a state in T has a state in I.

While masking fault-tolerance is ideal, for reasons of costs and feasibility, a weaker level of fault-tolerance is often required. Two commonly considered weaker levels of fault-tolerance include *failsafe* and *nonmasking*. In particular, we say that p is failsafe fault-tolerant [25] if the conditions 1, 2a, and 2b are satisfied in the above definition. And, we say that p is nonmasking fault-tolerant [24] if the conditions 1, 2a, and 2c are satisfied in the above definition.

3 Problem Statement

In this section, we identify the repair problem for automated addition of fault-tolerance with and without explicit legitimate states.

Model Repair *with* Explicit Legitimate States (Approach in Figure 1). We formally specify the problem of deriving a fault-tolerant program p' from a fault-intolerant program p with explicit legitimate states I. The goal of the model repair is to modify p to p' by *only adding fault-tolerance*, i.e., without adding new behaviors in the absence of faults. Since the correctness of p is known only from its legitimate states, I, it is required that the legitimate states of p', say I', cannot include any states that are not in I. Additionally, inside the legitimate states, it cannot include transitions that were not transitions of p. Also, by Assumption 2.1, p' cannot include new terminating states that were not terminating states of p. Finally, p' must be fault-tolerant. Thus, the problem statement (from [30]) for the case where the legitimate states are specified explicitly is as follows.

Problem Statement 3.1:
Repair for Fault-Tolerance *with* Explicit Legitimate States.
Given p, I, Sf_p, Lv_p and f such that p satisfies Sf_p and Lv_p from I,
Identify p' and I' such that: (Respectively, does there exist p' and I' such that)
A1: $I' \Rightarrow I$.
A2: $s_0 \in I' \;\Rightarrow\; \forall s_1 : s_1 \in I' : ((s_0, s_1) \in p' \Rightarrow (s_0, s_1) \in p)$.
A3: p' is f-tolerant to Sf_p and Lv_p from I'.

Note that this definition can be instantiated for each level of fault-tolerance (i.e., masking, failsafe, and nonmasking). Also, the above problem statement can be used as a repair problem or a decision problem (with the comments inside parenthesis).

We call the above problem the problem of 'partial repair' because the transitions of p' that begin in I' are a subset of the transitions of p that begin in I'. An alternative formulation is that of 'total repair' where the transitions of p' that begin in I' are *equal to* the transitions of p that begin in I'. In other words, the problem of *total repair* is identical to the problem statement 3.1 except that A2 is changed to A2′ described next:

A2′: $(s_0 \in I' \Rightarrow \forall s_1 \in I' : ((s_0, s_1) \in p' \iff (s_0, s_1) \in p)$

Modeling Repair *without* Explicit Legitimate States (Approach in Figure 2) Now, we formally define the new problem of model repair without explicit legitimate states. The goal in this problem is to find a fault-tolerant program, say p_r. It is, also, required that there is some set of legitimate states for p, say I, such that p_r does not introduce new behaviors in I. Thus, the problem statement for partial repair for the case where the legitimate states are not specified explicitly is as follows.

Problem Statement 3.2
Repair for Fault-Tolerance *without* Explicit Legitimate States.

Given p, Sf_p and Lv_p, and f
Identify p_r such that: (Respectively, does there exist p_r such that)
($\exists I$ ::
B1: $s_0 \in I \;\Rightarrow\; \forall s_1 : s_1 \in I :\; ((s_0, s_1) \in p_r \Rightarrow (s_0, s_1) \in p)$
B2: p_r is a f-tolerant to Sf_p and Lv_p from I.)

Just like problem statement 3.1, the problem of total repair is obtained from problem statement 3.2 by changing B1 with B1′ described next:

B1′: $s_0 \in I \;\Rightarrow\; \forall s_1 : s_1 \in I :\; ((s_0, s_1) \in p_r \iff (s_0, s_1) \in p)$

Existing algorithms for model repair [15,30,12,36, 16] are based on Problem Statement 3.1. Also, the tool SYCRAFT [15] utilizes Problem Statement 3.1 for the addition of fault-tolerance. However, as stated in Section 1, this requires the users of *SYCRAFT* to identify the legitimate states explicitly. The goal of this paper is to evaluate the effect of simplifying the task of the designers by permitting them to omit explicit identification of legitimate states.

4 Relative Completeness (Q. 1)

In this section, we show that if the problem of model repair can be solved with explicit legitimate states (Problem Statement 3.1) then it can also be solved without explicit legitimate states (Problem Statement 3.2). Since each problem statement can be instantiated with partial or total repair, this requires us to consider four combinations. We prove this result in Theorem 1.

Theorem 1 *If the answer to the decision problem 3.1 is affirmative (i.e., $\exists$ p' and I' that satisfy the constraints of the Problem 3.1) with input p, Sf_p, Lv_p, f, and I, then the answer to the decision problem 3.2 is affirmative (i.e., $\exists$ p_r that satisfies the constraints of the Problem 3.2) with input p, Sf_p, Lv_p, and f.*

Proof Since the answer to the decision problem 3.1 is affirmative, there exists program p' and I' that satisfy constraints in Problem Statement 3.1. To show that the answer to the decision problem 3.2 is affirmative, we need to find p_r such that constraints of Problem Statement 3.2 are satisfied. We let transitions of p_r to be

$\{(s_0, s_1) \mid s_0 \in I' \wedge s_1 \in I' \wedge (s_0, s_1) \in p \vee ((s_0 \notin I' \wedge (s_0, s_1) \in p')\ \}$

Next, we show that p_r satisfies the constraints of Problem Statement 3.2. Towards this end, we instantiate I to be I' and show that constraints $B1$ and $B2$ are satisfied

- **Constraint B1**: By construction of transitions of p_r, this constraint is satisfied for the case where we consider partial repair and for the case where we consider total repair.
- **Constraint B2**: By construction, I' is closed in p_r. Also, since $I' \Rightarrow I$ and p satisfies Sf_p and Lv_p from I, it is straightforward to observe that p_r satisfies Sf_p and Lv_p from I'.
  Also, transitions of p_r that begin outside I' are identical to that of p'. The second constraint "$(\exists T :: \ldots)$" from the definition of fault-tolerance is also satisfied. Thus, p_r is fault-tolerant to Sf_p and Lv_p from I'. □

Implication of Theorem 1 for Q. 1: From Theorem 1, it follows that answer to Q. 1 from Introduction is affirmative for both partial and total repair. Hence, the new formulation (cf Figure 2) is relatively complete.

5 Complexity Analysis for Partial Repair (Q. 2)

In this section, we focus on the complexity issue for partial repair (Problem 3.2). In particular, we first consider the complexity of the problem in the general case. In Section 5.1, we show that the problem of partial repair is NP-complete. Moreover, we show that this result is valid even if the specification is suffix-closed and fusion closed (defined later in Section 5.1). The importance of these conditions is that these constraints have been found to be useful in demonstrating existence of legitimate state predicates in the context of adding recovery [5]. Subsequently, in Section 5.2, we show that if we consider a restricted specification where the specification consists of a safety specification and one leads-to property then the problem of partial repair can be solved in polynomial-time. We show (in Section 5.2) that this result for polynomial time solution is tight in that the problem is NP-complete if we let specification to be a safety specification and two leads-to properties. And, in Section 5.3, we argue that it is also NP-complete if we consider a slightly more generalized form of safety specification, namely *conditional safety* specification, and no explicit liveness specification (i.e., other than deadlock freedom). We also revisit this issue in Section 6.1 to develop a heuristic for partial repair.

5.1 Complexity for Partial Repair

In this section, we utilize proof by reduction to show that the problem of partial repair of fault-tolerant program is NP-complete in the size of the state space. Towards this end, we define the decision problem precisely.

Instance. A program $p = \langle S_p, \delta_p \rangle$, a set of faults f, safety specification Sf_p, and liveness Lv_p.

The decision problem(FTwLS). Is the answer to the decision Problem 3.2 affirmative when instantiated with masking fault-tolerance?

Next, we utilize the fact that the 2-path problem is NP-complete [9] to show that the FTwLS problem is NP-complete by reducing the 2-path program to FTwLS.

The simplified 2-path problem(2PP). Given a digraph $G = \langle V, A\rangle$, where V is a set of vertices and A is a set of arcs, and three distinct vertices $v_1, v_2, v_3 \in V$. Decide whether G has a simple (v_1, v_3)-path that also contains vertex v_2.

Theorem 2 *FTwLS is NP-complete in the size of the state space of the input program.*

Mapping. Since the membership of FTwLS to NP is trivial, we focus on reduction from the 2PP instance to an instance of FTwLS. In particular, we first present a polynomial-time mapping from an arbitrary instance of the 2PP problem (i.e., $G = \langle V, A\rangle$, v_1, v_2 and v_3), to an instance of the FTwLS problem (i.e., p, f, Sf_p and Lv_p) as follows:

- (state space) $S_p = \{s_v | v \in V\}$
- (program transition) $\delta_p = (\{s_u \to s_v | (u, v) \in A\} - \{s_{v_3} \to s_u | (v_3, u) \in A\}) \cup \{s_{v_3} \to s_{v_1}\}$
- $f = \{\}$
- (safety specification) $Sf_p = S_p \times S_p - (\delta_p \cup f)$
- (liveness specification) $Lv_p = \{true \rightsquigarrow \{s_{v_2}\} \wedge true \rightsquigarrow \{s_{v_3}\}\}$

Here is an intuitive description of the above mapping. The state space of the program is obtained by including all the vertices in digraph G, and one state in the program corresponds to one vertex in the digraph. To construct the program's transitions, at first step each arc (e.g. (u, v)) in G is mapped as the corresponding transition. Then, all the transitions originating from s_{v_3} are removed. Finally we include the transition (s_{v_3}, s_{v_1}).

All transitions except those in p or f violate safety. (Note that there are no bad states in this instance.) The liveness specification requires that starting from an arbitrary state, the program reaches state s_{v_2} and state s_{v_3}.

Proof Given the above mapping, we now show that there is a solution to the 2PP problem, iff the answer to FTwLS is affirmative.

- ($\Rightarrow$) Let the answer to 2PP problem be a simple path Π that starts at vertex v_1, ends at v_3 and contains v_2. We show that the set of program transitions obtained from the arcs along Π and the transition (s_{v_3}, s_{v_1}) satisfies the constraints of FTwLS. We prove our claim as follows.
  To show this, we instantiate the set of legitimate states I to be the set of states obtained by considering states in Π. Specifically, $I = \{s_u | u \in \Pi\}$. Since transitions in any computation are obtained by arcs corresponding to the original graph, **B1** is trivially satisfied.
  Since the program satisfies $true \rightsquigarrow \{s_{v_3}\}$, state s_{v_3} must be reachable. From this state, there is only one outgoing transition to s_{v_1}. From this state, the program will continue executing in the loop included in the states in Π. Since Π is a simple path that visits both v_2 and v_3, it follows that this computation, say σ, satisfies the liveness specification. Moreover, since σ only uses transitions corresponding to the arcs in the original graph, safety specification is also satisfied. Additionally, since $f = \{\}$, the program always remains in I. Hence, constraint **B2** is also satisfied.
- ($\Leftarrow$) Let the answer to the FTwLS be $p' = \langle S_p, \delta_{p'}\rangle$. Observe that based on the liveness specification Lv_p, the legitimate state predicate I used to show that p' is fault-tolerant must include s_{v_2} and s_{v_3}. Moreover, since s_{v_3} has only one outgoing transition I must also include s_{v_1}.
  Now we show the existence of a simple path from v_1 to v_3 that includes v_2. This is a two-step proof, (1) First, we show that there exists a path Π from v_1 to v_3 that contains v_2, and (2) Then, we show that the path Π is acyclic.
  By the definition of Lv_p, p' reaches state s_{v_2}. Let α be the shortest computation prefix that reaches s_{v_2}. Likewise, let β be the shortest computation prefix originating at s_{v_2} and reaching s_{v_3}. Now, we can extend α with β to obtain a computation prefix $\alpha\beta$. Let Π be the path obtained by arcs corresponding to transitions in $\alpha\beta$. By construction, Π begins in v_1 and reaches v_2 first and then v_3. If $\alpha\beta$ contains any state more than once, then there is a computation of p' that can repeat that loop and never reach s_{v_3}. Since p' is satisfies Lv_p, this is not possible. Hence, $\alpha\beta$ does not contain any state more than once. In other words, Π is a simple path. □

Next, we point out that the above NP-completeness result is valid even if the specification of the program is suffix closed and fusion closed, where:

- **Suffix Closure:** A specification $SPEC$ is *suffix closed* iff for any sequence σ if σ satisfies $SPEC$ then every suffix of σ also satisfies $SPEC$.

– **Fusion Closure:** A specification $SPEC$ is *fusion closed* iff for any state x and any two finite state sequences α and γ and any two state sequences β and δ, the following condition is satisfied:

$\langle \alpha, x, \beta \rangle$ satisfies $SPEC$ and $\langle \gamma, x, \delta \rangle$ satisfies $SPEC$

$\Rightarrow$

$\langle \alpha, x, \delta \rangle$ satisfies $SPEC$ and $\langle \gamma, x, \beta \rangle$ satisfies $SPEC$.

This result follows from the observation that the instance of the FTwLS problem identified in Theorem 2 contains specification that is suffix closed and fusion closed. Thus, we have

Corollary 1 *FTwLS is NP-complete in the size of the state space of the input program even if the input specification is suffix closed and fusion closed.*

The reason for presenting this result is that in [5], it has been shown that if the specification is suffix closed and fusion closed then one can identify the set of legitimate states that can be used to provide recovery with the use of correctors. However, the result in this paper shows that even if this condition were satisfied, the complexity of adding partial repair can be high.

Also, the specification in the instance of the FTwLS problem constructed in the proof of Theorem 2 consists of a safety specification and a liveness specification that consists of two leads-to properties. Hence, we make the following observation.

Corollary 2 *FTwLS is NP-complete in the size of the state space of the input program even if the liveness specification Lv_p is restricted to two leads-to properties.*

The importance of this result lies in showing the tightness of the polynomial-time algorithm developed in Section 5.2 where Lv_p consists of one leads-to property. In particular, the above corollary shows that the assumption that Lv_p consists of at most one leads-to property is essential for the problem to be in polynomial-time.

Finally, we also observe that the proof of Theorem 2 can also be extended to the case where we consider failsafe or nonmasking fault-tolerance. Thus, we have

Corollary 3 *Problem 3.2 is NP-complete when instantiated with partial repair and failsafe fault-tolerance (respectively, nonmasking fault-tolerance)*

ALGORITHM 1: Add_masking_sf_lv_fr: Addition of Masking Fault-Tolerance

Input: program transitions p, fault transitions f, safety specification Sf_p (consisting of $SPEC_{bs}$ and $SPEC_{bt}$), liveness specification Lv_p (consisting of one $\mathcal{F} \rightsquigarrow \mathcal{T}$ property)

Output: masking fault-tolerant program p_r.

1: $ms := \{s_0 : \exists s_1, ..., s_n : (\forall j : 0 \leq j < n : (s_j, s_{j+1}) \in f) \wedge (s_{n-1}, s_n) \in SPEC_{bt}\}$
2: $mt := \{(s_0, s_1) : ((s_1 \in ms \vee SPEC_{bs}) \vee (s_0, s_1) \in SPEC_{bt})\}$
3: $fs := S_p - ms - SPEC_{bs}$
4: $p_1 := p - mt$
5: $Inv := fs$
6: **repeat**
7: $\quad old\_Inv, old\_fs := Inv, fs$
8: $\quad Inv := Inv - deadlock(Inv, p_1)$
9: $\quad$ **repeat**
10: $\quad\quad S_1, T_1 := Inv, fs$
11: $\quad\quad p_1 := p|Inv \cup \{(s_0, s_1)|s_0 \notin Inv \wedge s_0 \in fs \wedge s_1 \in fs\} - mt$
12: $\quad\quad fs := fs - \{s|s \in fs \wedge rank(s, Inv, p_1) = \infty\}$
13: $\quad\quad$ **while**$(\{s_0|\exists s_1 : s_0 \in fs \wedge s_1 \notin fs \wedge (s_0, s_1) \in f\} \neq \emptyset)$
14: $\quad\quad\quad$ **do** $fs := fs - \{s_0|\exists s_1 : s_0 \in fs \wedge s_1 \notin fs \wedge (s_0, s_1) \in f\}$
15: $\quad\quad Inv := Inv \wedge fs$
16: $\quad\quad$ **while** $deadlock(Inv, p_1) \neq \emptyset$
17: $\quad\quad\quad$ **do** $Inv := Inv - deadlock(Inv, p_1)$
18: $\quad\quad$ **if** $Inv = \emptyset \vee fs = \emptyset$ **then**
19: $\quad\quad\quad$ declare no masking fault-tolerant program p' exists
20: $\quad\quad$ **end if**
21: $\quad$ **until** $(S_1 = Inv \wedge T_1 = fs)$
22: $\quad Inv := Inv - \{s|rank(s, \mathcal{T}, p_1) = \infty \wedge s \in \mathcal{F}\}$
23: **until** $old\_Inv = Inv \wedge old\_fs = fs$
24: $p_1 := p_1 - \{(s_0, s_1)|s_0 \in fs - Inv \wedge rank(s_0, Inv, p_1) \leq rank(s_1, Inv, p_1)\}$
25: $p_r := p_1 - \{(s_0, s_1)|s_0 \in Inv \wedge rank(s_0, \mathcal{T}, p_1) \leq rank(s_1, \mathcal{T}, p_1) \wedge rank(s_0, \mathcal{T}, p_1) \neq 0 \wedge rank(s_1, \mathcal{T}, p_1) \neq \infty\}$

Function: deadlock(S: state predicate , t: transition set)
1: **return** $\{s_0|s_0 \in S \wedge (\forall s_1 \in S : (s_0, s_1) \notin t)\}$

Function: rank(s: state , $\mathcal{T}$: state predicate , t: transition set)
1: **return** the shortest path length from s to one of the states in $\mathcal{T}$, if the path(consisting only transitions in t) exists; or ∞, otherwise.

5.2 Algorithm for Model Repair without Legitimate States

In this section, we present the algorithm Add_masking_sf_lv_fr to show that for some special cases; i.e., programs where the specification consists of a safety specification and at most one leads-to property, the problem of partial repair can be solved in polynomial time. We first show this result for masking fault-tolerance and then point out how it can be extended for failsafe and nonmasking fault-tolerance.

Given a program p, fault transitions f, and program specification $\langle Sf_p, Lv_p \rangle$, the goal of the algorithm is to compute a masking fault-tolerant program p_r that satisfies the constraints of Problem 3.2.

The algorithm starts with identifying states ms and transitions mt. In particular, ms denotes states from where faults alone can violate safety. And, mt includes transitions that either reach ms or reach $SPEC_{bs}$ or are in $SPEC_{bt}$ (Lines 1 and 2). Thus, it follows that p_r should not reach a state in ms and it should not execute a transition in mt. Variable fs denotes the current estimate of the fault-span of p_r. Likewise, p_1 denotes current estimate of transitions in p_r. Hence, on Line 3, we remove ms from fs. And, we remove mt from p_1. Finally, Inv, the current estimate of the legitimate state predicate is initialized to fs (Line 5).

The algorithm continues with a loop (Lines 6-23) to repair program p_1 until there is no change to Inv and fs. First of all, all the deadlock states in Inv are removed (Line 8). Then, the inner loop (Line 9-21) removes program states and transitions so as to obtain a masking fault-tolerant program. Program transitions p_1 are recalculated (Line 11) according to the new value of Inv. This includes all transitions that could be included while respecting constraints of Problem 3.2. Subsequently, fault-span is computed as follows: First, we remove states from where there is no chance of recovery to Inv (Line 12). Next we remove states from fs that violate its closure (Line 13-14) in the presence of faults. If any state is removed from fs, it is also removed from Inv (Line 15). If this creates any deadlock states in Inv, those states are removed (Line 16-17). If this results in removal of all states in Inv, the algorithm declares failure. The inner loop terminates when there is no change to Inv and fs.

After the inner loop terminates, we focus on the liveness specification. Specifically, we remove those states in $\mathcal{F}$ from where reaching a state in $\mathcal{T}$ is not possible (Line 22).

Upon termination of the outer loop, program p_1 (Line 11) ensures that every state in fs has a path to a state in Inv and every state in $\mathcal{F} \wedge Inv$ has a path to $\mathcal{T}$. However, reaching these states may be prevented by other transitions that create a cycle. Hence, we remove such cycles (Line 24 and 25). Thus, the algorithm is as shown in Algorithm 1.

The proof of correctness for Add_masking_sf_lv_fr contains two parts, proof for soundness and proof for completeness respectively. We prove these results in Theorems 3 and 4.

Theorem 3 Add_masking_sf_lv_fr *is sound.*

Add_masking_sf_lv_fr is sound if the repaired program satisfies the constraints of Problem 3.2 when instantiated with partial repair and masking fault-tolerance. To show this, we need to show the following constraints.

- p_r satisfies Sf_p and Lv_p from Inv

  Proof When Add_masking_sf_lv_fr terminates there is no change in the last iteration of the outer or the inner loop. With this observation, by Line 8, the program does not deadlock in any state in Inv. Furthermore, by Lines 3, 4 and 11, the program does not reach a state in ms from where faults violate safety. And, it does not execute a transition that violates safety either. Hence, safety is satisfied.
  Regarding liveness, by Line 22, $rank(s, \mathcal{T}, p_1)$ of all states in $Inv \cap \mathcal{F}$ is finite. In other words, there is a path from every state in $Inv \cap \mathcal{F}$ to some state in $\mathcal{T}$. Moreover, by Line 25, starting from any state in Inv with finite rank and executing a transition in p_r reduces the rank. Thus, eventually, starting from every state in $\mathcal{F}$, we reach a state where $rank(s, \mathcal{T}, p_1)$ is 0, i.e., we reach a state in $\mathcal{T}$. Hence, Lv_p is satisfied.

- $p_r[]f$ satisfies Sf_p from fs
  This proof is similar to the proof that p_r satisfies Sf_p from Inv.
- fs is f-span of p_r from Inv

  Proof From Line 22, it follows that Inv is subset of fs. Closure of fs in p follows from Line 11 and the fact that p_r is obtained by removing (but not adding any) transitions from p_1. Closure of fs in f follows from Lines 13 and 14 and the fact that no state is removed in the last iteration of loop 6-23 or 9-21.

- Any computation of p_r starting from fs reaches a state in Inv.

  Proof From Line 12, it follows that upon termination of loop 6 to 23 and 9 to 21, $rank(s, Inv, p_1)$ is finite, i.e., every state in fs has no path to some state in Inv. Moreover, by Line 24, any transition of p reduces the rank of a state in $fs - Inv$. Hence, it implies that every computation starting in fs reaches a state in Inv. □

Theorem 4 Add_masking_sf_lv_fr *is complete.*

Proof Add_masking_sf_lv_fr declares failure only if all states are removed from Inv or fs. Next, we show that any state that is removed from legitimate state predicate (respectively fault-span) cannot be in the Inv (respectively fs) of any solution that solves Problem 3.2. For example, Line 3 removes states from fs. The removed states are such that faults violate safety from those states. It follows that these states cannot be included in the fault-span of any solution that solves Problem 3.2. Also, Line 8 removes deadlock states from Inv. Again, these states cannot be included in the invariant of any solution. By a similar argument, we can

show that all the removed states and transitions must be removed to obtain a masking fault-tolerant program. □

5.2.1 Algorithm for Adding Fail-safe and Non-masking Fault-tolerance for Model Repair without Legitimate States

The algorithm Add_masking_sf_lv_fr we discussed so far is designed to add masking fault-tolerance to program. Now, we point out that with some modifications, Add_masking_sf_lv_fr can be used for adding failsafe or nonmasking fault-tolerance. Specifically, if we remove Line 12 from Add_masking_sf_lv_fr then the resulting algorithm is sound and complete for adding failsafe fault-tolerance. And, if we initialize ms and mt to be empty sets, then resulting algorithm is sound and complete for adding nonmasking fault-tolerance.

5.2.2 Role of the Requirements of Add_masking_sf_lv_fr.

The Add_masking_sf_lv_fr algorithm assumes that the input specification is a safety property and a liveness property that consists of one leads-to property. Corollary 2 shows that the assumption about one leads-to property is essential. In particular, if Lv_p consists of two leads-to properties then the problem is NP-complete. In Section 5.3, we show that the assumption about safety property is also essential. In particular, if we consider a slightly generalized version, called conditional safety specification, the problem of adding masking fault-tolerance becomes NP-complete.

5.3 Complexity for Partial Repair with Conditional Safety Specification

In Section 5.2, we showed that the problem of adding masking fault-tolerance is in polynomial-time if the specification is a safety specification and one leads-to property. Here, we show that the problem becomes NP-complete if we relax the safety specification requirement with a conditional safety specification. This result is valid even if the program does not need to satisfy any explicit liveness specification(except deadlock freedom) in the absence of faults. This demonstrates that the condition for polynomial-time algorithm in Section 5.2 is tight.

The conditional safety specificaiton considered in this paper is motivated by the difference between the safety specificaiton considered previously in this paper and the general safety specification from [2]. Specifically, the general safety specification in [2] states that the safety specification is specified by a set of *prefixes*. A computation violates this generalized safety specification if any prefix of that computation is ruled out by the generalized safety specification. Our notion of safety specification in Section 2 considers the case where it is possible to decide whether a given prefix violates the safety specificaiton by simply evaluating the last two states in that prefix. The conditional safety specificaiton considered in this section generalizes this to the case where deciding whether a given prefix violates the safety specification is decided by the last three states. The NP-completeness result for conditional safety specification implies a similar hardness result for generalized safety specificaiton in [2].

Additionally, we note that in this section, the specification of interest is only a conditional safety specification and does not include any liveness requirement. In other words, we show that the problem of partial repair with conditional safety specification is NP-complete even if the input does not contain any liveness requirements.

To show this result, we first define conditional safety specification, that is a generalization of safety specification considered elsewhere in the paper. A conditional safety specification, Csf_p, for program p is specified in terms of bad states, $SPEC_{bs}$, bad transitions $SPEC_{bt}$, and bad transition triples $SPEC_{btp}$. Thus, $\sigma{=}\langle s_0, s_1, ...\rangle$ satisfies the conditional safety specification iff the following three conditions are satisfied.

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

In this section, we consider a generalized version of Problem 3.2 for the case where the input safety specification, Sf_p, is replaced by conditional safety specification Csf_p. We show that the problem of partial repair for adding masking fault-tolerance is NP-complete if the specification is a conditional safety specification and without any liveness specification.

Instance. A program $p = \langle S_p, \delta_p\rangle$, a set of faults f, and conditional safety specification Csf_p.

The decision problem (CSS). Is the answer to the decision Problem 3.2 (with extension to conditional safety specification) affirmative when instantiated with masking fault-tolerance?

We show that CSS is NP-complete by a reduction from the 3-SAT problem, defined next.

3-SAT Instance. Let $x_1, x_2, ..., x_n$ be propositional variables. Given is a Boolean formula

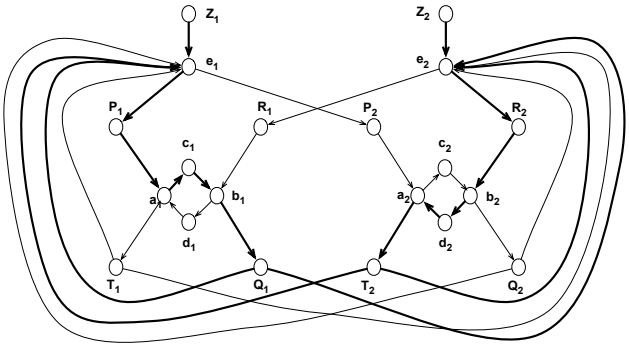

Fig. 3 Mapping of $(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$ into corresponding program transitions. The transitions in bold show the repaired program where $x_1 = true$ and $x_2 = false$

$y = y_1 \ \wedge \ y_2 \ \cdots \ \wedge \ y_M$, where each y_j $(1 \leq j \leq M)$ is a disjunction of exactly three literals. Does there exist an assignment of truth values to $x_1, x_2, ..., x_n$ such that y is satisfiable?

Theorem 5 *CSS is NP-complete in the size of the state space of the input program.*

Since showing the membership of partial repair without legitimate states with conditional safety specification in NP is straightforward, we focus on showing that it is NP-hard. Hence, we first present the mapping from the 3-SAT instance to the problem of partial repair without explicit legitimate states. Then, we show that the given 3-SAT instance is satisfiable iff the answer to the corresponding instance of partial repair is affirmative.

5.3.1 Mapping 3-SAT to Partial Repair without Explicit Legitimate States

We now present the mapping of an instance of the 3-SAT problem to an instance of the partial repair problem without explicit legitimate states with conditional safety specification. Recall that this instance consists of the program (specified in terms of its state space and transitions), conditional safety specification and faults. We begin with identifying the input program. Then, we identify faults and finally we identify the conditional safety specification.

The state space of the input program. Corresponding to each variable x_i of the given 3-SAT instance, we introduce eight states $P_i, Q_i, R_i, T_i, a_i, b_i, c_i$, and d_i where $1 \leq i \leq n$ (cf. Figure 3). For each disjunction y_j, we introduce states Z_j and e_j, where $1 \leq j \leq M$, in the state space. Thus, state space of the input program is $S_p = \{P_i, Q_i, R_i, T_i, a_i, b_i, c_i, d_i \mid 1 \leq i \leq n\} \cup \{Z_j, e_j \mid 1 \leq j \leq M\}$.

Transitions of the input program. Corresponding to each variable x_i, we include the following transitions in δ_p: $(P_i, a_i), (a_i, c_i), (c_i, b_i), (b_i, Q_i)$, $(R_i, b_i), (b_i, d_i)$, (d_i, a_i), (a_i, T_i), (Q_i, e_j) and (T_i, e_j) where $1 \leq j \leq M$. Moreover, corresponding to each disjunction y_j, we include the following transitions:

- (Z_j, e_j),
- If x_i is a literal in y_j then we include the transition (e_j, P_i) in δ_p, and
- If $\neg x_i$ is a literal in y_j then we include the transition (e_j, R_i) in δ_p.

Fault transitions. Fault transitions perturb the program from T and Q states to Z states, specifically, the fault transitions are $f = \{(T_i, Z_j), (Q_i, Z_j) | 1 \leq i \leq n, \ 1 \leq j \leq M\}$.

Conditional safety specification Csf_p**.** Conditional safety specifications states that no transition other than the original program or fault transition can be included in the fault-tolerant program. Furthermore, it states that transition (P_i, a_i) cannot be followed by (a_i, T_i). Likewise,(R_i, b_i) cannot be followed by (b_i, Q_i). And, (c_i, b_i) cannot be followed by (b_i, d_i). Finally, (d_i, a_i) cannot be followed by (a_i, c_i). Thus, we have

- $SPEC_{bt} = S_p \times S_p - (f \cup \delta_p)$,
- $SPEC_{btp} = \{(P_i, a_i, T_i)\} \cup \{(R_i, b_i, Q_i)\} \cup \{(c_i, b_i, d_i)\} \cup \{(d_i, a_i, c_i) | 1 \leq i \leq n\}$.

Proof – ($\Rightarrow$) If the given instance of the 3-SAT problem is satisfiable then we construct the transitions of repaired program by including the following transitions:

  - (Z_j, e_j), $1 \leq j \leq M$,
  - If y_j contains x_i and x_i is assigned truth value *true*, then (e_j, P_i),
  - If y_j contains $\neg x_i$ and x_i is assigned truth value *false*, then (e_j, R_i),
  - If x_i is assigned truth value *true* ,then $(P_i, a_i), (a_i, c_i), (c_i, b_i), (b_i, Q_i)$, and (Q_i, e_j), $1 \leq i \leq n$,
  - If x_i is assigned truth value *false*, then $(R_i, b_i), (b_i, d_i), (d_i, a_i)$, (a_i, T_i), and (T_i, e_j), $1 \leq i \leq n$.

  The predicate, I', used to show that this program satisfies Csf_p includes all reachable states except $\{ Z_j | 1 \leq j \leq M \}$. It is straightforward to show that the constraints **B1** and **B2** are satisfied.
- ($\Leftarrow$) The legitimate state predicate of the repaired program contains at least one state. Our first step is to show that for some i, Q_i or T_i is included in the legitimate state predicate of the repaired program. To show this, we observe that if Z_j, $1 \leq j \leq M$, is included in the legitimate state predicate for some

j then the corresponding state e_j must also be included in the legitimate state predicate. Hence, the repaired program must include at least one transition that begins in e_j. It follows that either P_i or R_i, $1 \leq i \leq n$, must also be included the legitimate state predicate. If P_i (respectively R_i) is included in the legitimate state predicate, then a_i (respectively b_i) must also be included so that program does not deadlock. Also, if a_i(respectively b_i) is included in the legitimate state predicate then it must reach c_i(respectively d_i) as next state, so that Csf_p is not violated. Likewise, the program must reach b_i and Q_i (respectively a_i and T_i) to avoid deadlock and violation of safety specification. From the above discussion, it follows that for some i, Q_i or T_i is included in the legitimate state predicate of the repaired program. Now, based on the definition of faults, all states in $\{Z_j | 1 \leq j \leq M \}$ are reachable in the presence of faults. Hence, transition (Z_j, e_j) must be included for $1 \leq j \leq M$ in the repaired program.

Furthermore, some transition originating from e_j must also be included. Transitions from e_j correspond to literals in disjunction y_j. If a transition of the form (e_j, P_i) is included then we set x_i to *true*. If a transition of the form (e_j, R_l) is included then we set x_l to *false*.

Observe that if P_i is reachable in the repaired program then it must also include (P_i, a_i), (a_i, c_i), (c_i, b_i), and (b_i, Q_i) so that Csf_p is satisfied. And, if R_i is reachable in the repaired program then it must also include (R_i, b_i), (b_i, d_i), (d_i, a_i), and (a_i, T_i). However, if all these transitions are included then Csf_p will not be satisfied. Therefore, for any i, repaired program cannot reach both P_i and R_i. This implies that the truth value assigned to x_i by any disjunction is the same; i.e., x_i cannot be both *false* and *true* simultaneously. Moreover, based on the construction of the instance of the program of partial repair, the truth assignments to literals make each clause to be satisfied, i.e., the assignment of truth values to literals causes the given 3-SAT formula to be satisfiable.

□

6 Complexity Comparison for Total Repair (Q. 2 (continued))

Although the complexity of partial repair increases substantially when legitimate states are not available explicitly, we find that complexity of total repair effectively remains unchanged. We note that this is the first instance where complexity difference between partial and total repair has been identified. To show this result, we show that in the context of total repair Problem 3.2 is polynomial time reducible to Problem 3.1. Since the results in this section require the notion of weakest legitimate state predicate, we define it next. Recall that, we use the term legitimate state predicate and the corresponding set of legitimate states interchangeably. Hence, weakest legitimate state predicate corresponds to the largest set of legitimate states.

Definition. Let $I_w = wLsp(p, Sf_p, Lv_p)$ be the *weakest legitimate state predicate* of p for $SPEC (= \langle Sf_p \ , \ Lv_p \rangle)$ iff:
1: p satisfies $SPEC$ from I_w, and
2: $\forall \ I :: \ (\ p$ satisfies $SPEC$ from $I) \Rightarrow (I \ \Rightarrow \ I_w)$. □

Claim Given p, Sf_p, and Lv_p, it is possible to compute $wLsp(p, Sf_p, Lv_p)$ in polynomial time in the state space of p.

This claim was proved in [1] where we have identified an algorithm to compute weakest legitimate state predicate. The algorithm utilizes state exploration to identify states from where the specification may be violated in the presence of faults. Thus, it first identifies the weakest state predicate from where occurence of faults cannot lead a computation to a state a bad transition is enabled. Then, it removes the states from where leads-to properties cannot be satisfied. We note that the rest of the paper only relies on existence of such an algorithm, i.e., it does not rely on its details.

Theorem 6 *If the answer to the decision problem 3.2 (***with total repair***) is affirmative (i.e., $\exists$ p_r that satisfies the constraints of the Problem 3.2) with input p, Sf_p, Lv_p, and f, then the answer to the decision problem 3.1 (with total or partial repair) is affirmative (i.e., $\exists$ p' and I' that satisfy the constraints of the Problem 3.1) with input p, Sf_p, Lv_p, f, and $wLsp(p, Sf_p, Lv_p)$.*

Proof Intuitively, the program p_r obtained for solving problem statement 3.2 can be used to show that problem 3.1 is satisfied. Specifically, let I_2 be the predicate used to show that p_r satisfies constraints of Problem 3.2. Since $wLsp(p, Sf_p, Lv_p)$ is the weakest legitimate state predicate, $I_2 \ \Rightarrow \ wLsp(p, Sf_p, Lv_p)$. Now, it is starightforward to observe that the constraints of Problem statement 3.1 are satisfied. Then, let $p' = p_r$ and $I' = I_2$. □

Remark: Note that if the phrase 'with total repair' shown in bold in Theorem 6 is replaced by 'with partial repair' then the corresponding theorem is not valid.

Theorem 7 *For total repair, the repair problem 3.2 is polynomial time reducible to the repair problem 3.1.*

Proof Given an instance, say X, of the decision problem 3.2 that consists of p, Sf_p, Lv_p, and f, the corresponding instance, say Y, for the decision problem 3.1 is p, Sf_p, Lv_p, f and $wLsp(p, Sf_p, Lv_p)$. From Theorems 1 and 6 it follows that answer to X is affirmative iff answer to Y is affirmative. □

In this section, we first summarize the results about complexity of model repair with and without explicit legitimate states. As shown earlier in this section, the complexity of total repair remains unchanged when legitimate states are not explicitly identified. However, as shown in Section 5, the problem of partial repair is NP-complete. Both these results are valid for high atomicity programs where each process can read and write all variables atomically.

For distributed programs, it is shown (in [30]) that repairing the program for adding failsafe and masking fault-tolerance is NP-complete when the set of legitimate states is specified explicitly. A variation of that proof also works for model repair without explicit legitimate states. Also, for partial repair, the complexity of adding nonmasking fault-tolerance can be shown to be NP-complete based on the corresponding proof of NP-completeness of partial repair from Section 5. However, for total repair with or without explicit legitimate states, the problem of adding nonmasking fault-tolerance is in NP. However, it is not known whether it is NP-complete or whether it is in polynomial-time.

In summary, the results for complexity comparison are as shown in Table 1. Results marked with † follow from NP-completeness results from Section 5. Results marked ‡ follow from this section. Results marked ? indicate that the complexity of the corresponding problem is open. The results marked ∗ are from [30]. And, finally, the results marked △ follow from the corresponding proof in [30] and is stated here without proof.

6.1 Heuristic for Polynomial Time Solution for Partial Repair

When comparing the complexity of total repair and partial repair, we observe that the problem of total repair could be solved in P with the use of weakest legitimate state predicate. In particular, Theorem 6 utilizes the weakest legitimate state predicate to solve the problem of total repair without explicit legitimate states. In this section, we show that a similar approach can be utilized to develop a heuristic for solving the problem of partial repair in polynomial time. Moreover, if there is an affirmative answer to the repair problem with explicit legitimate states then this heuristic is guaranteed to find a repaired program that satisfies constraints of Problem 3.2. Towards this end, we present Theorem 8.

Theorem 8 *For partial repair, the repair problem 3.2 consisting of (p, Sf_p, Lv_p, f) is polynomial time reducible to the repair problem 3.1 provided there exists a legitimate states predicate I such that the answer to the decision problem 3.1 for instance (p, I, Sf_p, Lv_p, f) is affirmative.*

Proof If an instance of Problem 3.1 has an affirmative answer then from Theorem 1, the corresponding instance of Problem 3.2 has an affirmative answer. Similar to the proof of Theorem 7, we map the instance of Problem 3.2 to an instance of Problem 3.1 where we use the weakest legitimate state predicate. Now, from Theorem 1 it follows that the answer to this repaired instance of Problem 3.1 is also affirmative. □

What the above theorem shows is that even for partial repair, if it were possible to obtain a fault-tolerant program with explicit legitimate states then it is possible to do so in the same complexity class without explicit legitimate states. However, there may be instances where the answer to the decision problem 3.1 may be negative and the answer to the corresponding decision problem 3.2 is affirmative. For these instances, for partial repair, the complexity can be high.

7 Relative Computation Cost (Q. 3)

As mentioned in Section 1, the increased cost of model repair in the absence of explicit legitimate states needs to be studied in two parts: complexity class and relative increase in the execution time. We considered the former in Section 5 and 6. In this section, we consider the latter. As we can see from Section 6, if the legitimate states are not specified explicitly, the increased cost of model repair is essentially that of computing $wLsp(p, Sf_p, Lv_p)$. Hence, we analyze the cost of computing $wLsp(p, Sf_p, Lv_p)$ in the context of three examples, Byzantine agreement, token ring and mutual exclusion. Of these we describe the first example in detail and only provide a summary of results for other two. These examples show that reducing the burden of the designer in terms of not requiring the explicit legitimate states increases the computation cost by approximately 1% or less.

Throughout this section, the experiments are run on a MacBook Pro with 2.6 Ghz Intel Core 2 Duo processor and 4 GB RAM. The OBDD representation of the Boolean formula has been done using the C++ interface to the CUDD package developed at the University of Colorado [43].

		Repair *Without* Explicit Legitimate States		Repair *With* Explicit Legitimate States	
		Partial	**Total**	**Partial**	**Total**
High Atomicity	Failsafe	$NP - C^{\dagger}$	$P^{\ddagger}$	P^{*}	P^{*}
	nonmasking	$NP - C^{\dagger}$	$P^{\ddagger}$	P^{*}	P^{*}
	masking	$NP - C^{\dagger}$	$P^{\ddagger}$	P^{*}	P^{*}
Distributed Programs	Failsafe	$NP - C^{\dagger}$	$NP - C^{\triangle}$	$NP - C^{*}$	$NP - C^{*}$
	nonmasking	$NP - C^{\dagger}$	?	?	?
	masking	$NP - C^{\dagger}$	$NP - C^{\triangle}$	$NP - C^{*}$	$NP - C^{*}$

Table 1 The complexity of different types of automated repair (NP-C = NP-Complete).

7.1 Case Study 1: Byzantine agreement program

Now, we illustrate our algorithm in the context of the Byzantine agreement program. We start by specifying the fault-intolerant program. Then we provide the program specification. Finally we describe the weakest legitimate state predicate generated by our algorithm.

Fault-intolerant program. The Byzantine agreement program consists of a "general" and three or more non-general processes. Each process copies the decision of the general and finalizes (outputs) that decision. The general process maintains two variables: the decision $d.g$ with domain $\{0, 1\}$ and the Byzantine $b.g$ with domain $\{true, false\}$, to indicate whether or not the general is Byzantine. Moreover, a byzantine process can change its decision arbitrarily. Each of the non-general processes has the following three variables: the decision d with domain $\{0, 1, \bot\}$, where $\bot$ denotes that the process did not yet receive any decision from the general, the Byzantine b with domain $\{true, false\}$, and the finalize f with the domain $\{true, false\}$ to denote whether or not the process has finalized (outputted) its decision. The following are the actions of the Byzantine agreement program. Of these, the first action allows a non-general to receive a decision from the general if it has not received it already. The second action allows the non-general to finalize its decision after it receives it. The third and fourth actions allow a Byzantine process to change its decision and finalized status. The last two actions are environment actions.

$$
\begin{array}{lrcl}
1 :: & (d.j = \bot) \wedge (f.j = false) & \longrightarrow & d.j := d.g; \\
2 :: & (d.j \neq \bot) \wedge (f.j = false) & \longrightarrow & f.j := true; \\
3 :: & (b.j = true) & \longrightarrow & \\
 & d.j := 1|0, & f.j := false|true; & \\
4 :: & (b.g = true) & \longrightarrow & d.g := 1|0;
\end{array}
$$

Where $j \in \{1 \ldots n\}$ and n is the number of non-general processes.

Specification. The safety specification of the Byzantine agreement requires *validity* and *agreement*:

- *Validity* requires that if the general is non-Byzantine, then the final decision of a non-Byzantine process must be the same as that of the general. Thus, $validity(j)$ is defined as follows.
  $validity(j) = ((\neg b.j \wedge \neg b.g \wedge f.j) \Rightarrow (d.j = d.g))$
- *Agreement* means that the final decision of any two non-Byzantine processes must be equal. Thus, $agreement(j, k)$ is defined as follows.
  $agreement(j, k) = ((\neg b.j \wedge \neg b.k \wedge f.j \wedge f.k) \Rightarrow (d.j = d.k))$
- The final decision of a process must be either 0 or 1. Thus, $final(j)$ is defined as follows.
  $final(j) = f.j \Rightarrow (d.j = 0 \vee d.j = 1)$

We formally identify safety specification of the Byzantine agreement in the following set of bad states:

$SPEC_{BA_{bs}} = (\exists j, k \in \{1..n\} :: (\neg(validity(j) \wedge agreement(j, k) \wedge final(j)))$

Observe that $SPEC_{BA_{bs}}$ can be easily derived based on the specification of the Byzantine Agreement problem. The liveness specification of the Byzantine agreement requires that eventually every non-Byzantine process finalizes its decision. Thus, the liveness specification is $\neg b.j \rightsquigarrow (f.j)$.

Application of our algorithm. The weakest legitimate state predicate computed (for 3 non-general processes) is as follows. If the general is non-Byzantine then it is necessary that $d.j$, where j is also a non-Byzantine, be either $d.g$ or $\bot$. Furthermore, a non-Byzantine process cannot finalize its decision if its decision equals $\bot$. Now, we consider the set of states where the general

is Byzantine. In this case, the general can change its decision arbitrarily. Also, the predicate includes states where other processes are non-Byzantine and have the same value that is different from $\perp$. Thus, the generated weakest legitimate state predicate is as follows:

$$
\begin{array}{l}
I_{\mathcal{BA}} = \\
(\ \neg b.g \wedge (\ \forall p \in \{1..n\} :: ((\neg b.p \wedge f.p) \Rightarrow d.p \neq \perp) \wedge \\
(\neg b.p \Rightarrow (d.p = \perp \vee d.p = d.g)))\ ) \vee \\
(\ b.g \wedge (\ \forall\, j,k \in \{1..n\} : j \neq k :: \\
\qquad\qquad (d.j = d.k)\ \wedge\ (d.j \neq \perp)\ )\ )
\end{array}
$$

Observe that $I_{\mathcal{BA}}$ cannot be easily derived based on the specification of the Byzantine Agreement problem. More specifically, the set of states where the general is Byzantine, are not reachable from the initial states of the program.

The amount of time required for performing the automated model repair and computing the set of legitimate states for a different number of processes is as shown in Table 2. Thus, the extra time required when legitimate states are unavailable is small (about 1%).

Finally, we use this case study to show that a typical algorithm for computing legitimate states to be reachable states from some initial state(s) does not work. In particular, we make the following claim:

Claim An automated model repair approach where one uses legitimate states to be the set of states reached from the initial states is not relatively complete.

To validate this claim, observe that the initial states of the Byzantine Agreement program equal the states where all processes are non-byzantine and the decision of all non-general processes is $\perp$. States reached by the fault-intolerant program from these states do not include the states where the general is byzantine. Although, an agreement can be reached among non-general processes even though the general is Byzantine. And, utilizing these reachable states as the set of legitimate states is insufficient to obtain the fault-tolerant program.

7.2 Case Study 2: Token Ring

The second case study focuses on the problem of adding fault-tolerance to a token ring. Specifically, in this example, there are $n+1$ processes organized in a ring protocol. Each process has a variable x that is either 0 or 1. Initially, all x values are equal to 0. In such a state (where $x.0$ is equal to $x.n$) process 0 has the token and can execute. When process 0 executes, it changes $x.0$ to 1. In the resulting state (where $x.0$ differs from $x.1$), process 1 has the token and process 1 can execute and change $x.1$ to 1. Continuing thus, eventually, one reaches a state where all x values are 1 and thereby the token is with process 0. At this point, $x.0$ is changed to 0 and the process repeats itself. Thus, the actions of the program are as shown below.

$$
\begin{array}{lll}
1\ ::\ x.j \neq x.(j-1) & \longrightarrow & x.j\ :=\ x.(j-1); \\
2\ ::\ x.0 = x.n & \longrightarrow & x.0\ :=\ x.n +_2 1;
\end{array}
$$

where $+_2$ denotes modulo 2 addition.

The fault perturbs all except one process. Upon occurrence of faults at process j, $x.j$ is lost and is replaced by $\perp$. Adding fault-tolerance to this example requires that only legal values of x are used by other processes and that the program eventually returns to a state from where the token circulation will resume correctly. This example has been studied in [12] for its application in automated model repair with explicit legitimate states and in [1] for automatically generating weakest legitimate state predicate. Table 3 shows the comparison of the cost for both. As one can see, the cost of generating legitimate state predicate is very small.

7.3 Case Study 3: Mutual Exclusion

The third case study focuses on the problem of mutual exclusion where the fault-intolerant program is Raymond's tree-based mutual exclusion program [41]. In this example, processes are organized in a ring and the root process has the token. To request access to critical section, each process sends a request to its holder (the process closer to the root in the tree). This request is forwarded towards the root. When the root decides that it can send the token (e.g., after it is done using it), it sends it to one of its children that requested it. Moreover, the tree is reorganized so that the new process that has the token is the root of the tree. We only focus on this aspect of the program and model it using the following action: (The role of maintaining request queue affects both generation of legitimate states and model repair in a similar manner).

$$
1 :: (h.k = k\ \wedge\ j \in Adj.k)\ \wedge\ (h.j = k) \longrightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad h.k := j,\ \ h.j := j;
$$

where $Adj.k$ denotes one of the neighbors of k.

The fault perturbs the holder relation arbitrarily. Hence, the holder relation does not form a tree thereby preventing some processes from getting the token as well as creating multiple tokens. The goal in this case is to generate a stabilizing mutual exclusion program that will ensure that eventually the program recovers to a state where the holder relation forms a tree. This example has been studied in [12] for its application in automated model repair with explicit legitimate states

No.of Process	Reachable States	Leg. States Generation Time(Sec)	Total Repair Time(Sec)
10	10^9	0.57	6
20	10^{15}	1.34	199
30	10^{22}	4.38	1836
40	10^{30}	9.25	9366
50	10^{36}	26.34	>10000
100	10^{71}	267.30	>10000

Table 2 Comparison time for generating weakest legitimate state predicate vs time for model repair (Byzantine Agreement).

No. of Process	**Reachable States**	**Leg. States Generation Time(Sec)**	**Time for Model Repair**
10	10^4	0.1	0.6
20	10^9	0.2	4
30	10^{14}	0.3	493
40	10^{19}	0.4	>10000
50	10^{23}	0.6	>10000
100	10^{47}	0.19	>10000

Table 3 Comparison time for generating weakest legitimate state predicate vs time for model repair (Token Ring).

and in [1] for automatically generating weakest legitimate state predicate. The subsequent results (cf. Table 4) show the comparison of the cost for both. As one can see, the cost of generating legitimate state predicate is very small.

8 Related Work

The concept of model repair is a branch of program synthesis. This topic has been studied from different perspectives ranging from synthesis from temporal logic specifications to controller synthesis and synthesizing winning strategies in game theory. In this section, we present the work in the literature that is relevant to our work in this paper.

8.1 Controller Synthesis

Synthesis of discrete-event systems has mostly been studied in the context of controller synthesis and game theory. The seminal work in the area of controller synthesis is due to Ramadge and Wonham [40]. The discrete controller synthesis (DCS) problem is as follows: starting from two languages $\mathcal{U}$ and $\mathcal{D}$, identify a third language $\mathcal{C}$ such that $\mathcal{U} \cap \mathcal{C} \subseteq \mathcal{D}$. In the DCS terminology, the three languages $\mathcal{U}$, $\mathcal{D}$, and $\mathcal{C}$ are called the *plant*, the *desired system*, and the *controller*, respectively. $\mathcal{U} \cap \mathcal{C}$ is called the *controlled system*. Finally, the set $\mathcal{A}$ of alphabets represents events that can occur. Obviously, the languages $\mathcal{U}$ and $\mathcal{D}$ may represent the set of computations of a given program and a safety and/or reachability specification. Moreover, $\mathcal{C}$ identifies the computations that do not violate $\mathcal{D}$ in the presence of uncontrollable transitions.

The idea of transforming a fault-intolerant system into a fault-tolerant system using controller synthesis was first developed by Chao and Lim [21]. Similar to the model in this paper, Chao and Lim consider faults as a system malfunction and failures as something that should not occur in any execution. Their control objective is a set of states that should be reachable by controllable actions or what they define as *recurrent events*. Also, Girault and Rutten [26], demonstrate the application of discrete controller synthesis in automated addition of fault-tolerance in the context of untimed systems. They model different types of faults (e.g., processor crash, Byzantine faults, value corruption) by uncontrollable actions in a labeled transition system (LTS). They show that given a fault-intolerant program as a plant, discrete controller synthesis can automatically add fault-tolerance to the synchronous product of the plant and the fault model LTS with respect to invariance and reachability constraints.

One can notice that our work in this paper is in spirit close to DCS. Specifically, an input program and fault transitions may be modelled as controllable and uncontrollable actions. In fact, in both problems, the objective is to *restrict* the program actions at each state through synthesizing a controller such that the behavior of the entire system is always *desirable* according to safety and reachability conditions, in the presence

No. of Process	**Reachable States**	**Leg. States Generation Time(Sec)**	Time for Model Repair
10	10^9	0.01	0.7
20	10^{26}	0.1	11
30	10^{44}	0.2	65
40	10^{64}	0.5	260
50	10^{84}	0.9	725
100	10^{200}	0.43	$>$ 10000

Table 4 Comparison time for generating weakest legitimate state predicate vs time for model repair (Mutual Exclusion).

of an adversary. As mentioned in Section 3, conditions **A1** and **A2** of the problem statement precisely express this notion of restriction. Furthermore, the conjunction of all conditions expresses the notion of *language inclusion*, where the synthesized program in the absence of faults is supposed to exhibit a subset of behaviors of the input intolerant program. Although the complexity of controller synthesis for centralized programs is comparable to ours if we eliminate distribution from our formulation [27], our work differs from synthesizing discrete-event controllers in that:

1. The computation model for synthesizing controllers is based on prioritized synchronization[3], whereas ours is based on interleaving.
2. Our synthesis algorithm is concerned with properties typically used in specifying fault-tolerance requirements rather than any arbitrary specification. Hence, our algorithm tends to synthesize programs more efficiently.
3. The notion of dependability and in particular fault-tolerance involves features beyond just invariance and reachability. One such feature is *recovery*, where a program returns to its normal behavior when its state is perturbed by the occurrence of faults. Recovery is an essential building block in fault-tolerant systems and it is the focus of this paper. In controller synthesis [26,19,22,6,7,21,35,40,47,33,42,32], and in particular in [26], which is in spirit close to our work, the recovery mechanism must be given as input to the DCS algorithm. Thus, one key difference between our work in this paper and the methods in is the fact that we automatically synthesize recovery paths. We also analyze the complexity and cost of dealing with livelocks (i.e., cycles outside the invariant predicate) in this paper. Finally, note that adding recovery transitions to a given fault-intolerant program is not trivial due to the distributed nature of programs and read/write restrictions.
4. Finally, we model *distribution* by specifying read/write restrictions, whereas in controller synthesis, decentralized plants are modelled through *partial observability* [33,42]. As mentioned earlier, the issue of distribution drastically increases the complexity of synthesis [14,30]. This results is also known in the context of controller synthesis, but to the best of our knowledge, this paper introduces the first instance where synthesis of distributed fault-tolerant programs scales up and moderate-sized programs beyond toy examples are successfully synthesized. We are also not aware of any work in controller synthesis that analyzes efficiency and effectiveness of respective algorithms using different types of programs in terms of structure and size. Unlike existing controller synthesis tools such as Sigali [17], our synthesis method and corresponding tool Sycraft [15] is capable of synthesizing distributed programs.

[3] In prioritized synchronization, different processes have synchronized transitions and some have higher priority. Hence, not all interleavings are possible.

8.2 Game Theory

Game-theoretic approaches for synthesizing controllers and reactive programs [38] are generally based on the model of two-player games [45]. In such games a program makes moves in response to the moves of its environment. The program and its environment interact through a set of interface variables and, hence, the environment can only update the interface variables. In our model, however, faults can perturb all program variables. Moreover, in a two-player game model, players take turns and the set of states from where the first player can make a move is disjoint from the set of states from where the second player can move [48]. To the contrary, in our work, fault-tolerance should be provided against faults that can execute from any state.

Some theoretic methods are based on the theory of tree automata [44]. Such an automaton represents the specification of a system. A synthesis algorithm checks the non-emptiness of the automaton, i.e., whether there exists a tree acceptable by the tree automaton. If the tree automaton is indeed nonempty, then the specifica-

tion is called *realizable* and there exists a model of the synthesized program.

Pnueli and Rosner address the problem of synthesizing synchronous open reactive modules in [38]. They generalize their method in [39], by proposing a technique for synthesizing asynchronous reactive modules. In particular, they investigate the problem of synthesizing an asynchronous reactive module that include only one process and interacts with a non-deterministic environment through Boolean variables.

While symbolic model checking has been studied extensively (e.g., [20,37,28]), little work has been done on symbolic synthesis and especially on performance analysis of synthesis methods. Wallmeier, Hütten, and Thomas [48] introduce an algorithm for synthesizing finite state controllers by solving infinite games over finite state spaces. They model the winning constraint by safety conditions and a set of request-response properties as liveness conditions. They transform this game into a Büchi game which inevitably involves an exponential blow-up. Moreover, the approach in [48] does not address the issue of distribution. The reported maximum number of variables in their experiments is 23, which is far less than the number of variables that we have handled using our symbolic algorithms.

Although there exist approaches for games that are not turn-base, similar to discrete controller synthesis, such game-theoretic approaches do not address the issue of addition of recovery. That is, in order to use algorithms that synthesize game strategies (e.g., the well-known algorithm by Liu and Smolka [34]), one has to first augment the input model with *all* possible recovery transitions and then solve the game. Such augmentation can be extremely costly. Our approach synthesizes recovery path in an on-the-fly fashion. Other frameworks such as multi-agent games (e.g., alternating-time temporal logic (ATL) [3]) are also dffierent from our work. In particular, ATL is a branching-time temporal logic, whereas our notion of specifications (based on Alpern-Schneider framework [2]) is semantically similar to LTL. In fact, in our framework preserving the existing specification during repair is in part possible due to the universal nature of properties. On the contrary, existential path properties in branching-time temporal logics make it impossible to, for instance, remove a transition without having the full specification at hand to test whether it violates the existing specification.

8.3 Automated Addition of Fault-Tolerance

Algorithms for automatic addition of fault-tolerance [30,31,11,13] add fault-tolerance concerns to existing untimed or real-time programs in the presence of faults, and guarantee the addition of no new behaviors to the original program in the absence of faults. While the previous work in this area assumes the set of legitimate states is given as input, this paper studies the impact of this assumption on repair techniques to synthesize fault-tolerant models. In the seminal work in this area, Kulkarni and Arora [30] introduce synthesis methods for automated addition of fault-tolerance to untimed centralized and distributed programs. In particular, they introduce polynomial-time sound and complete algorithms for adding all levels of fault-tolerance (failsafe, nonmasking, and masking) to centralized programs. The input to these algorithms is a fault-intolerant centralized program, safety specification, and a set of fault transitions. The algorithms generate a fault-tolerant program along with an invariant predicate. The authors also show that the problem of adding masking fault-tolerance to distributed programs is NP-complete in the size of the input program's state space.

In [31], Kulkarni and Ebnenasir address the problem of automated synthesis of *untimed multitolerant* programs, i.e., programs that tolerate multiple classes of faults and provide a (possibly) different level of fault-tolerance to each class. They show that if one needs to add failsafe (respectively, nonmasking) fault-tolerance with respect to one class of faults and masking fault-tolerance with respect to another class of faults, then such addition can be done in polynomial-time in the size of the state space of the fault-intolerant program. They, however, show that if one needs to add failsafe fault-tolerance with respect to one class of faults and nonmasking fault-tolerance with respect to another class of faults, then the problem is NP-complete.

Ebnenasir [23] develops a divide-and-conquer method synthesis problem that scales up. In an application of this approach for safety properties, Ebnenasir develops an algorithm that statically analyzes (and possibly repairs) program instructions on separate machines in a parallel/distributed platform. Based on this method, the author implements a distributed framework that exploits the computational resources of wide area networks for program synthesis. Using this approach, it is possible to synthesize failsafe Byzantine agreement with 40 processes on a cluster of three machines in 353 seconds. Using our BDD-based approach, the same program can be synthesized in 22 seconds using one machine only.

The problem of online fault *detection* in timed automata is studied by Tripakis [46]. The author proposes a polynomial-space online algorithm for designing a diagnoser that detects faults in behaviors of a given timed automaton after they occur. In this work, it is assumed that (1) the given system is in the synchronous model,

and (2) faults and failures are identical events. Thus, this model does not capture situations where the occurrence of faults (although undesirable) is common and expected, but may *lead* to a system with failures. Bouyer, Chevalier, and D'Souza [18] address the same problem where the diagnoser is realizable as a deterministic timed automaton or an event record automaton.

9 Conclusion and Future Work

The goal of this work is to simplify the task of model repair and, thereby, make it easier to apply in practice. Of the inputs required for model repair, existing model is clearly a necessity. Moreover, the model repair task is expected to be straightforward, as model repair is expected to be used in contexts where designers already have an existing model. Another input, namely specification, is also already available to the designer when model repair is used in contexts where the existing model fails to satisfy the desired specification. Yet another input is the new property that is to be added to the existing model. In the context of fault-tolerance, this requires the designers to identify the faults that need to be tolerated. Once again, identifying the fault is straightforward especially in contexts where the model needs to be repaired due to newly identified faults (e.g., stuck-at gas pedal). While representing these faults may be somewhat difficult, it is possible to represent them easily for faults such as stuck-at faults, crash faults, Byzantine faults, transient faults, message faults, etc.

However, based on our experience, the most difficult input to identify is the set of legitimate states from where the original model satisfies its specification. In part, the difficulty is due to the fact that identifying these legitimate states explicitly is often not required during the evaluation of the original model. Hence, our goal in this paper is to focus on the problem of model repair when these legitimate states are not specified explicitly. Moreover, as shown by the example in Section 7 typical algorithms for computing legitimate states based on initial states do not work.

We considered three questions in this context: (1) relative completeness, (2) qualitative complexity class comparison and (3) quantitative change the time for model repair. We illustrated that our approach for model repair without explicit legitimate states is relatively complete; i.e., if model repair can be solved with explicit legitimate states then it could also be solved without explicit legitimate states. This is important since it implies that the reduction in the human effort required for model repair does not reduce the class of the problems that could be solved.

Regarding the second question, we found some surprising and counterintuitive results. Specifically, for total repair (where the behavior of the repaired model in the absence of faults is identical to the behavior of the original model), we found that the complexity class remains unchanged. However, for partial repair, the complexity class changes substantially. In particular, we showed that problems that could be solved in polynomial-time when legitimate states are available explicitly become NP-complete if explicit legitimate states are unavailable. This result is especially surprising since this is the first instance where complexity levels for total and partial repair have been found to be different. Even though the general problem of partial repair becomes NP-complete without the explicit legitimate states, we found a subset of these problems that can be solved in polynomial-time. Specifically, this subset included all instances where model repair was possible when legitimate states are specified explicitly.

Regarding the third question, we showed that the extra computation cost obtained by reducing the human effort for specifying the legitimate states is negligible (less than 1%) in our case studies obtained from popular distributed fault-tolerant protocols.

Also, we have integrated our algorithms for automated repair without explicit legitimate states in our synthesis tool *SYCRAFT* [15]. We note that this result can also be extended to other problems in model repair where one adds safety and liveness properties as well as timing constraints.

References

1. F. Abujarad and S. Kulkarni. Weakest Invariant Generation for Automated Addition of Fault-Tolerance. *Electronic Notes in Theoretical Computer Science*, 258(2):3–15, 2009. Available as Technical Report MSU-CSE-09-29 at http://www.cse.msu.edu/cgi-user/web/tech/reports?Year=2009.
2. B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.
3. R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 49(5):672–713, 2002.
4. A. Arora and M. G. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19(11):1015–1027, 1993.
5. A. Arora and S. S. Kulkarni. Detectors and correctors: A theory of fault-tolerance components. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 436–443, 1998.
6. E. Asarin and O. Maler. As soon as possible: Time optimal control for timed automata. In *Hybrid Systems: Computation and Control (HSCC)*, pages 19–30, 1999.
7. E. Asarin, O. Maler, A. Pnueli, and J. Sifakis. Controller synthesis for timed automata. In *IFAC Symposium on System Structure and Control*, pages 469–474, 1998.

8. A. Avižienis, J. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, pages 11–33, 2004.
9. J. Bang-Jensen and G. Gutin. *Digraphs: Theory, Algorithms and Applications*, chapter 9, pages 477–478. Springer, 2002.
10. B. Bonakdarpour, A. Ebnenasir, and S. S. Kulkarni. Complexity results in revising UNITY programs. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4(1):1–28, January 2009.
11. B. Bonakdarpour and S. S. Kulkarni. Incremental synthesis of fault-tolerant real-time programs. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, LNCS 4280, pages 122–136, 2006.
12. B. Bonakdarpour and S. S. Kulkarni. Exploiting symbolic techniques in automated synthesis of distributed programs with large state space. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 3–10, 2007.
13. B. Bonakdarpour and S. S. Kulkarni. Masking faults while providing bounded time phased recovery. In *International Symposium on Formal Methods (FM)*, 2008.
14. B. Bonakdarpour and S. S. Kulkarni. Revising distributed UNITY programs is NP-complete. In *Principles of Distributed Systems (OPODIS)*, pages 408–427, 2008.
15. B. Bonakdarpour and S. S. Kulkarni. Sycraft: A tool for synthesizing distributed fault-tolerant programs. In *CONCUR '08: Proceedings of the 19th international conference on Concurrency Theory*, pages 167–171, Berlin, Heidelberg, 2008. Springer-Verlag.
16. B. Bonakdarpour, S. S. Kulkarni, and F. Abujarad. Symbolic synthesis of masking fault-tolerant programs. *Springer Journal on Distributed Computing (DC)*, 25(1):83–108, March 2012.
17. P. Bournai, M. L. Borgne, and P. L. Guernic. Synthesis of discrete-event controllers based on the signal environment. In *Discrete Event Dynamic System: Theory and Applications*, pages 325–346, 2000.
18. P. Bouyer, F. Chevalier, and D. D'Souza. Fault diagnosis using timed automata. In *Foundations of Software Science and Computation Structure*, pages 219–233, 2005.
19. P. Bouyer, D. D'Souza, P. Madhusudan, and A. Petit. Timed control with partial observability. In *Computer Aided Verification (CAV)*, pages 180–192, 2003.
20. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.
21. K. H. Cho and J. T. Lim. Synthesis of fault-tolerant supervisor for automated manufacturing systems: A case study on photolithography process. *IEEE Transactions on Robotics and Automation*, 14(2):348–351, 1998.
22. D. D'Souza and P. Madhusudan. Timed control synthesis for external specifications. In *Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 571–582, 2002.
23. A. Ebnenasir. DiConic addition of failsafe fault-tolerance. In *Automated Software Engineering (ASE)*, pages 44–53, 2007.
24. F. Gartner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys (CSUR)*, 31(1):1–26, 1999.
25. F. C. Gärtner and A. Jhumka. Automating the addition of fail-safe fault-tolerance: Beyond fusion-closed specifications. In *FORMATS/FTRTFT*, pages 183–198, 2004.
26. A. Girault and É. Rutten. Automating the addition of fault tolerance with discrete controller synthesis. *Formal Methods in System Design (FMSD)*, 35(2):190–225, 2009.
27. P. Gohari and W. M. Wonham. On the complexity of supervisory control design in the RW framework. *IEEE Transactions on Systems, Man, and Cybernetics*, 30(5):643–652, 2000.
28. T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.
29. B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *Computer Aided Verification (CAV)*, pages 226–238, 2005.
30. S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. In *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*, pages 82–93, 2000.
31. S. S. Kulkarni and A. Ebnenasir. Automated synthesis of multitolerance. In *International Conference on Dependable Systems and Networks (DSN)*, pages 209–219, 2004.
32. R. Kumar and V. K. Garg. Optimal supervisory control of discrete event dynamicalsystems. *SIAM Journal on Control and Optimization*, 33(2):419–439, 1995.
33. F. Lin and W. M. Wonham. Decentralized control and coordination of discrete-event systems with partial observation. *IEEE Transactions On Automatic Control*, 35(12), December 1990.
34. X. Liu and S. A. Smolka. Simple linear-time algorithms for minimal fixed points (extended abstract). In *Proceedings of the 25th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 53–66, 1998.
35. O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In *12th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 229–242, 1995.
36. H. Mantel and F. C.Gärtner. A case study in the mechanical verification of fault-tolerance. Technical Report TUD-BS-1999-08, Department of Computer Science, Darmstadt University of Technology, 1999.
37. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
38. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Principles of Programming Languages (POPL)*, pages 179–190, 1989.
39. A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *International Colloqium on Automata, Languages, and Programming*, number 372 in Lecture Notes in Computer Science, pages 652–671. Springer-Verlag, 1989.
40. P. J. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, 1989.
41. K. Raymond. A tree based algorithm for mutual exclusion. *ACM Transactions on Computer Systems*, 7:61–77, 1989.
42. K. Rudie and W. M. Wonham. Think globally, act locally: Decentralized supervisory control. *IEEE Transactions On Automatic Control*, 37(11):1692–1708, 1992.
43. F. Somenzi. CUDD: Colorado University Decision Diagram Package. `http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html`.
44. W. Thomas. *Handbook of Theoretical Computer Science: Chapter 4, Automata on Infinite Objects*. Elsevier Science Publishers B. V., 1990.

45. W. Thomas. On the synthesis of strategies in infinite games. In *Theoretical Aspects of Computer Science (STACS)*, pages 1–13, 1995.
46. S. Tripakis. Fault diagnosis for timed automata. In *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*, pages 205–224, 2002.
47. S. Tripakis and K. Altisen. On-the-fly controller synthesis for discrete and dense time systems. In *Formal Methods 1999 (FM)*, pages 233–252, 1999.
48. N. Wallmeier, P. Hütten, and W. Thomas. Symbolic synthesis of finite-state controllers for request-response specifications. In *Implementation and Application of Automata (CIAA)*, pages 11–22, 2003.