# Automated Model Repair for Distributed Programs

**Borzoo Bonakdarpour**
School of Computer Science
University of Waterloo
200 University Ave. West
Waterloo, ON, N2L 3G1, Canada
borzoo@cs.uwaterloo.ca

**Sandeep S. Kulkarni**
Dept. of Computer Science and Engineering
Michigan State University
3115 Engineering Building
East Lansing, MI 48824, USA
sandeep@cse.msu.edu

**Abstract**

*Model repair* is a formal method that aims at fixing bugs in models automatically. Typically, these models are finite state automata that can be compactly represented using guarded commands or variations thereof. The bugs in these models can be identified using traditional techniques, such as verification, testing, or runtime monitoring. However, these techniques do not assist in fixing bugs automatically. The goal in model repair is to automatically transform an input model into another model that satisfies additional properties (e.g., a property that the original model fails to satisfy). Moreover, such transformation should preserve the existing specification of the input model. In this article, we review the efforts in the past decade on developing model repair algorithms in different domains. These domains include distributed computing, fault-tolerance and self-stabilization, and real-time systems. We present the results on complexity analysis, techniques for tackling intractability of the problem and scalability, and related tools. The techniques and tools discussed in this article demonstrate the feasibility of automated synthesis of well-known protocols such as Byzantine agreement, token ring, fault-tolerant mutual exclusion, etc.

## 1 Introduction

Although distributed systems are widely used nowadays, their implementation and deployment are still a time-consuming, error-prone, and hardly predictable task. Ensuring system-wide *correctness* is generally a challenging task and this challenge is significantly amplified when dealing with distributed systems. This is due to the inherently concurrent and non-deterministic behavior of distributed systems, as well as the occurrence of unanticipated physical and computational events such as faults. Thus, it is highly advantageous for designers to have access to automated methods that *synthesize* models[1] of distributed systems that are correct by construction.

There are other benefits to synthesizing models of computing systems as well. Requirements of a system normally evolve during the system's life cycle due to different reasons such as *incomplete specification* and *change of environment*. While the former is usually a consequence of poor requirements engineering, the latter is a maintenance issue. This notion of maintenance turns out to be critical for systems where programs are integrated with large collections of sensors and actuators in hostile physical environments. In such systems, it is essential that programs react to physical events such as faults, delays, signals, attacks, etc., so that the system specification is not violated. Since it is impossible to anticipate all possible physical events of this kind at design time, it is highly desirable to have automated techniques that revise and repair models according to the system specification with respect to newly identified physical events.

Taking the paradigm of correct-by-construction to extreme leads us to automated synthesis from specification [33], where a model is constructed from scratch from a set of temporal logic properties. Alternatively, in *model repair*, an algorithm transforms an input model into another model that meets additional properties. This approach is beneficial when one has an initial model at hand that is *almost* correct. In this way, one can reuse the previous efforts made to develop the model and does not have to synthesize a model from scratch which tends to have higher complexity.

In this article, we review the efforts in the past decade on model repair. The problem of model repair has been studied in different contexts, such as distributed computing, fault-tolerant systems, and systems sensitive to timing or physical constraints. These research activities have mainly focused on two broad directions:

- **Complexity analysis.** The first step in dealing with the repair problems is identifying the complexity hierarchy. Complexity analysis is essential in order to identify cases where automated repair (1) is likely to be successful via developing efficient sound and complete algorithms or (2) would require development of efficient heuristics to reduce the complexity.

- **Designing efficient heuristics.** In order to tackle the high complexity of repair problems (e.g., NP-complete in the state space), it is necessary to develop efficient heuristics and efficient data structures. Some of the commonly used data structures include binary decision diagrams (BBDs), difference bound matrices (DBMs), etc. They allow a compact representation of Boolean formulae (used to represent programs, specifications etc). Specifically, various symbolic heuristics using binary decision diagrams (i.e., BDD-based) have enabled repair of moderate-sized models (i.e., $10^{80}$ and beyond). These heuristics demonstrate the feasibility of synthesizing classic fault-tolerant distributed protocols, such as Byzantine agreement [46], token ring [30], and distributed mutual exclusion [52] with fairly large number of processes.

**Organization.** The rest of the article is organized as follows. In Section 2, we discuss problems related to model repair. In Section 3, we state the basic model repair problem. In Section 4, we illustrate the problem of model repair in the context of an example where we discuss synthesis of the classic byzantine agreement problem. Using the example in Section 4, we identify variations of the model repair problem in different contexts. Specifically, in Section 5, we identify issues that one needs to handle while applying model repair in distributed programs, real-time programs, fault-tolerant programs, etc. In Section 6, we

[1] Models in this report typically refer to models used in model checking literature. They can be thought of as finite state automata (FSA). Various compact representations of FSA are used in the literature. These include guarded commands, and Promela, etc. In some cases, the techniques used in this report can be applied to UML state diagrams or even C code.

discuss the state of the art in terms of implementation results for model repair. Section 7 is dedicated to results on complexity analysis of the repair problem and its variations. Finally, in Section 8, we discuss open problems and future research directions.

## 2 Related Work

The seminal work on model synthesis from temporal logic specification was introduced in [33, 49]. 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 [51]. The idea of transforming a fault-intolerant system into a fault-tolerant system using controller synthesis was first developed by Chao and Lim [28]. Similar to our idea of addition of fault-tolerance, Chao and Lim consider faults as a system malfunction and failures as something that should not occur in any execution path. Their control objective is a set of states that should be reachable by controllable actions. Girault and Rutten [36] use classic controller synthesis techniques to synthesize fault-tolerant protocols. However, their work falls short of generating recovery mechanism for a fault-intolerant system.

Automated *model repair* is a relatively new area of research. Model repair with respect to Computation Tree Logic (CTL) properties was first considered in [22] using AI techniques. A formal algorithm for model repair in the context of CTL is presented in [57]. Model repair for CTL using abstraction techniques has been studied in [26]. The theory of model repair for memoryless LTL properties was considered in [39] in a game-theoretic fashion; i.e., a repaired model is obtained by synthesizing a winning strategy for a 2-player game. In [12], the authors explore the model repair for a fragment of LTL (the UNITY language [24]). Most results in [12] focus on complexity analysis of model repair for different variations of UNITY properties. Model repair in other contexts includes the work in [10] for probabilistic systems and in [53] for Boolean programs.

*Program sketching* [55] is another synthesis method for achieving correctness by construction. In this line of research, a program is generated from high-level specification and a finite set of programming constructs.

Game-theoretic approaches for synthesizing controllers and reactive programs [50] are generally based on the model of two-player games [56]. In such games, a program makes moves in response to the moves of its environment. *Quantitative synthesis* of programs from temporal specifications is game-theoretic technique based on pay-off games and has been studied [11, 23]. Such synthesis takes quantitative objectives (e.g., the number times that a process can request entering critical section) into account. Since there are normally multiple solutions to a synthesis problem, quantitative objectives allows one to guide synthesis algorithms to generate realistic programs.

Lily and Anzu [37, 38] are two tools for synthesizing automata from LTL specification. QUASY [25] synthesizes programs from temporal logic by taking quantitative objectives into account. Tools for supervisory controller synthesis include [4, 35]. SYCRAFT [18] and FTSyn [31] are two tools for synthesizing fault-tolerant distributed models. FTSyn's technology is based on generating explicit state space, while SYCRAFT uses binary decision diagrams as compact representation of state space.

## 3 The Basic Model Repair Problem

The model repair problem is as follows. Let $\mathcal{M}$ be a model, $\Sigma$ be a logical specification, where $\mathcal{M}$ satisfies $\Sigma$, and $\Pi$ be a logical property, where $\mathcal{M}$ does not satisfy $\Pi$. The repair problem is to devise an algorithm that generates a model $\mathcal{M}'$ such that $\mathcal{M}'$ satisfies $\Sigma$ and $\Pi$ simultaneously. In the discussion in this report, we assume that a model is represented by a state-transition system and a property (or specification) is represented in some temporal logic (e.g., LTL [32] or one of its fragments such as UNITY [24]).

Typically, in model repair, it is assumed that property $\Sigma$ is not available during the repair process. In other words, one needs to utilize the fact that $\mathcal{M}$ satisfies $\Sigma$ to deduce that $\mathcal{M}'$ satisfies $\Sigma$. To achieve this, one of the inputs to model repair problem typically includes a set of legitimate states (say $L$), i.e., states from where the original program satisfies $\Sigma$. That is, all computations from these state satisfy $\Sigma$. In turn, one is required to identify the set of legitimate states (say, $L'$) from where the repaired program satisfies $\Sigma$ and $\Pi$.

When model repair is being done to add safety and/or liveness properties, it is expected that $L'$ is a subset of $L$. This is due to the fact that since if we do not know $\Sigma$ and have no knowledge about behavior of $\mathcal{M}$ in states outside $L$ then we cannot construct $\mathcal{M}'$ that satisfies $\Sigma$ from states outside $L$.

When model repair is being done to add fault-tolerance, however, one does need to concern with states reached outside $L$. In this case, one implicit requirement (for certain levels of tolerance) is that the program recovers to a state in $L$ after faults stop. Moreover, in this case, the requirement $\Pi$ specifies constraints that have to be satisfied by the repaired program during recovery.

We illustrate the model repair algorithm with an example from distributed computing in Section 4. Then, we utilize this example to compare variations of model repair problems as well as their implementation.

## 4 Illustration of Model Repair in Distributed Systems

In this section, we illustrate the problem of model repair in the context of an example from distributed systems. We choose the problem of *Byzantine agreement* (denoted $\mathcal{BA}$). Specifically, we begin with a fault-intolerant version of $\mathcal{BA}$ and show how model repair can be applied to add fault-tolerance. The repaired protocol is identical to the protocol originally proposed in [47].

### 4.1 Input to Model Repair Algorithm

The input to the model repair algorithm consists of a fault-intolerant program, legitimate states from where it works correctly, faults and the safety specification that needs to be satisfied during addition of fault-tolerance.

**Fault-intolerant Program.** The fault-intolerant version of $\mathcal{BA}$ consists of a *general*, say $g$, and three (or more) *non-general* processes: $j$, $k$, and $l$. Since the general process only provides a decision, it is modeled implicitly by two variables. The non-general processes are modeled explicitly based on the actions they perform. Each process of $\mathcal{BA}$ maintains a decision variable $d$; for the general, the decision can be either $0$ or $1$, and for the non-general processes, the decision can be $0$, $1$, or $\perp$, where the value $\perp$ denotes that the corresponding process has not yet received the decision from the general. Each non-general process also maintains a Boolean variable $f$ that denotes whether or not that process has finalized its decision. For each process, a Boolean variable $b$ shows whether or not the process is Byzantine. Thus, the state space of each process is obtained by the variables in the following set:

$$\begin{aligned} V_{\mathcal{BA}} = {} & \{d.g, d.j, d.k, d.l\} \cup && \text{(decision variables)} \\ & \{f.j, f.k, f.l\} \cup && \text{(finalized?)} \\ & \{b.g, b.j, b.k, b.l\}. && \text{(Byzantine?)} \end{aligned}$$

Note that even though we introduced a variable $b.g$ to denote whether the general is byzantine, it is expected that $j$ cannot learn whether $g$ is byzantine. This can be captured by ensuring that $j$ cannot read $b.g$. More generally, in this program, the set of variables that a non-general process, say $j$, is allowed to read and write are respectively:

$R_j = \{b.j, d.j, f.j, d.k, d.l, d.g\}$, and
$W_j = \{d.j, f.j\}$.

Note that for simplicity, we have modeled the program so that $j$ can read $d.k$ of process $k$. Without this, it would be necessary to introduce variables such as $d.k.j$ which denotes the value that $k$ sends to $j$. Also, the read/write restrictions of processes $k$ and $l$ can be symmetrically instantiated.

In the absence of faults, each non-general process copies the decision from the general and then finalizes (outputs) that decision, provided it is non-Byzantine. Thus, the transitions of a non-general process, say $j$, is specified by the following two actions[2]:

$$\begin{array}{lll} \mathcal{BA}1_j :: & (d.j = \bot) \wedge (f.j = \mathit{false}) & \longrightarrow \quad d.j := d.g; \\ \mathcal{BA}2_j :: & (d.j \neq \bot) \wedge (f.j = \mathit{false}) & \longrightarrow \quad f.j := \mathit{true}; \end{array}$$

**Safety Specification in the presence of faults.** The safety specification of $\mathcal{BA}$ requires *validity*, *agreement*, and *persistency*, where

- *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.
- *Agreement* means that the final decision of any two non-Byzantine processes must be equal.
- *Persistency* requires that once a non-Byzantine process finalizes (outputs) its decision, it cannot change it.

Thus, $SPEC_{bt_{\mathcal{BA}}}$ captures the set of transitions that violate the safety specification:

$$\begin{array}{l} SPEC_{bt_{\mathcal{BA}}} \;\; = \\ \quad (\exists p \in \{j,k,l\} \;::\; \neg b'.g \;\wedge\; \neg b'.p \;\wedge\; (d'.p \neq \bot) \;\wedge\; f'.p \;\wedge\; (d'.p \neq d'.g)) \\ \quad \vee \\ \quad (\exists p, q \in \{j,k,l\} \;::\; \neg b'.p \;\wedge\; \neg b'.q \;\wedge\; f'.p \;\wedge\; f'.q \;\wedge\; (d'.p \neq \bot) \;\wedge\; (d'.q \neq \bot) \;\wedge\; (d'.p \neq d'.q)) \\ \quad \vee \\ \quad (\exists p \in \{j,k,l\} \;::\; \neg b.p \;\wedge\; \neg b'.p \;\wedge\; f.p \;\wedge\; ((d.p \neq d'.p) \;\vee\; (f.p \neq f'.p))). \end{array}$$

Note that the liveness specification is not used since the implicit liveness requirement is that the program will recover to the legitimate states from where it will satisfy the liveness requirement, namely termination, which requires every non-byzantine process to finalize its decision.

**Legitimate states.** Observe that if the input program starts from an arbitrary state, it may not satisfy its specification. Specifically, if we start the program in a state where '$d.j = 0, f.j = 1, d.g = 1, b.g = 0, b.j = 0$', then the safety specification is immediately violated. Hence, an algorithm for model repair utilizes a legitimate state predicate that identifies the set of states from where the input program satisfies its specification. While there are several possible legitimate state predicates, the weakest predicate for this program can be characterized as follows:

[2]A action of the form $g \longrightarrow st$ corresponds to transitions of the form $(s_0, s_1)$ where $g$ is true in $s_0$ and $s_1$ is obtained by executing $st$ from state $s_0$.

1. First, we consider the set of states where the general is non-Byzantine. In this case:
   - one of the non-general processes may be Byzantine,
   - if a non-general process, say $j$, is non-Byzantine, it is necessary that $d.j$ be initialized to either $\perp$ or $d.g$, and
   - an undecided non-Byzantine process does not finalize its decision.
2. We also consider the set of states where the general is Byzantine. In this case, $g$ can change its decision arbitrarily. It follows that other processes are non-Byzantine and $d.j, d.k$ and $d.l$ are initialized to the same value that is different from $\perp$.

Thus, the legitimate state predicate is as follows:

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

An alert reader can easily verify that $\mathcal{BA}$ satisfies $SPEC_{\overline{bt}_{\mathcal{BA}}}$ from $L_{\mathcal{BA}}$.

**Faults.** The *fault-intolerant* version of $\mathcal{BA}$ is obtained by considering assumptions about faults and the effect of a fault. Specifically, since fault-tolerance can only be provided in the presence of one byzantine fault, we model the corresponding action as $F_0$. Moreover, if a process is byzantine, it can deceive others by sending incorrect values. For shared-memory programs, this can be achieved by allowing a byzantine process to change the variables it controls. Specifically, the fault transitions that affect a process, say $j$, of $\mathcal{BA}$ are as follows: (We include similar actions for $k$, $l$, and $g$)

$$\begin{array}{llll} F_0 & :: \ \neg b.g \wedge \neg b.j \wedge \neg b.k \wedge \neg b.l & \longrightarrow & b.j := true; \\ F_1 & :: \ b.j & \longrightarrow & d.j, \ f.j \ := \ 0|1, \ \ false|true; \end{array}$$

where $d.j \ := \ 0|1$ means that $d.j$ could be assigned either 0 or 1. In case of the general process, the second action does not change the value of any $f$-variable.

**Remark.** Note that although action $F_0$ appears to 'read' whether other processes are byzantine, this should not be thought of as if the fault is reading the variables $b$ of all processes. It is only capturing the assumption that if one process has become byzantine already then another cannot become byzantine again. Also, action $F_1$ captures the assumption that a byzantine process can only change variables it controls. Moreover, by changing variable $d$, a byzantine process can attempt to deceive other processes.

## 4.2 Application of Model Repair

The program in the previous subsection identifies a fault-intolerant version of byzantine agreement. It works correctly as long as no process misbehaves. Repairing such a distributed protocol is, in-general, NP-complete in the size of the state space. Hence, we need to develop heuristics to manage the complexity. Instead of describing the heuristics in detail, we illustrate how they work in the context of byzantine agreement.

### 4.2.1 Handling Distribution During Repair

Observe that (a non-general) process $j$ cannot read all variables in the program. This requirement is essential. Without this requirement, one could construct a repaired program where action of $j$ is of the form '$b.k = false \rightarrow \ldots$'. Clearly, such a program is not realizable for byzantine agreement.

This read restriction results in creation of *groups of transitions*. To illustrate this, consider the case where $j$ finalizes its decision. Furthermore, for sake of illustration, let the initial state be one where '$d.j = 0, d.k = 1, d.l = 0, d.g = 0, f.j = false, b.j = false$'. Note that this list includes all the variables $j$ can read. The action by which $j$ finalizes its decision corresponds to a group of transitions if we consider all possible values of variables $(namely, b.g, b.k, b.l, f.k, f.l)$ that $j$ cannot read. In other words, the action by which $j$ finalizes its decision corresponds to a group of 32 $(2{*}2{*}2{*}2{*}2)$ transitions. Moreover, if we include all 32 of these transitions, then their execution can be realized. However, if we only include a subset of these transitions then the corresponding program cannot be realized as it would require $j$ to read one or more of the variables that it cannot read.

### 4.2.2 Heuristics for Repairing Distributed Programs

One of the requirements of adding masking fault-tolerance is that the program satisfies the safety specification $SPEC_{bt_{\mathcal{BA}}}$ (consisting of validity, agreement and persistency) in the presence of faults. Hence, if we want to include a transition, say $(s_0, s_1)$, in the repaired program then the simplest requirement that one can enforce is that none of the transitions grouped with $(s_0, s_1)$ should violate safety. However, this requirement is too strict and results in failure to repair the program.

**Sample heuristics.** To explain one of the heuristics, we continue with the scenario considered earlier in Section 4.2.1. One of the transitions in that group corresponds to the case where '$b.k = b.l = b.g = false, f.k = f.l = true$'. This transition violates the agreement. However, we can observe that in this transition, the agreement is violated even in the initial state of the transition. One of the heuristics is to ignore such safety violation. This heuristic is always sound since the initial state of the transition where safety is violated must be made unreachable in any masking fault-tolerant program. Hence, if the initial state of a transition is unreachable, one does not concern with its effect.

However, in this scenario, there is a transition in the group that violates safety although safety is not violated in the initial state. Specifically, if '$b.j = b.k = false, f.k = true$', then this action results in a state where $j$ and $k$ have finalized with different decisions. In other words, in this state, the agreement requirement is violated. Hence, we cannot allow $j$ to finalize in the state where '$d.j = 0, d.k = 1, d.l = 0, d.g = 0, f.j = false, b.j = false$'.

Another heuristic we utilize is to evaluate the states reached in the *current program* in the presence of faults. Initially, the *current program* refers to the fault-intolerant program. If a safety violating transition originates in a state that is not currently reached by the *current program* in the presence of faults, we overlook that safety violating transition. This heuristic is based on the idea that if the state is not reached by the current program then it is likely that it will not be reached in the final program either. Moreover, as we repair the program by adding or removing transitions, this process is repeated to re-evaluate whether a safety violating transition can be overlooked.

**Adding recovery.** Preventing $j$ to finalize in a state where '$d.j = 0, d.k = 1, d.l = 0, d.g = 0, f.j = false, b.j = false$' creates a deadlock state since the original program does not allow any other transition for process $j$ in this situation. The heuristic for dealing with deadlock states is to first attempt to add recovery transitions. These recovery transitions can be single-step (i.e., transitions that take the program to a legitimate state) or multi-step (e.g., transitions that take the program to a state from where recovery was added earlier. Observe that in this case, we can add recovery transition that allows $j$ to change its decision to 1. Note that the group obtained by this action does not violate safety specification.

**Eliminating deadlock states/states that violate safety.** When adding recovery from a state is impossible, e.g., because any action for adding recovery contains transitions that violate safety and these safety violations cannot be ignored based on the heuristics, we choose to eliminate the deadlock states. Eliminating the deadlock states involves identifying a path that leads to the deadlock state and removing one of the transitions on that path. Specifically, one deadlock state encountered during repair is one where '$d.j = 0, d.k = 1, d.l = 1, d.g = 1, f.j = true, b.j = false$'. Essentially, in this state, a process has a decision that is in minority although it has finalized its decision. In this case, $j$ cannot change its decision since it has finalized. For this reason, we need to remove one or more transitions that cause the program to reach such a state. While the details of this elimination process is beyond the scope of this document, we note that eliminating this state results in removal of transition where $j$ finalizes its decision in the state where '$d.j = 0, d.k = \perp, d.l = \perp, d.g = 1, f.j = false, b.j = false$'. Intuitively, this change can be thought of as a process cannot finalize its decision while the decision of other processes is $\perp$.

## 4.3 Repaired Program

The output of the algorithm with respect to program $\mathcal{BA}$ is program $\mathcal{BA}'$ which tolerates the Byzantine faults identified in earlier in this section in the sense that $\mathcal{BA}'$ never violates its specification and it does not deadlock when faults occur. We note that the synthesized program is identical to the canonical version of Byzantine agreement program manually designed in [46]. The actions of the synthesized program for a non-general process $j$ are as follows:

$$\begin{array}{llll}
\mathcal{BA}'1_j :: & d.j = \perp \ \wedge \ f.j = \mathit{false} & \longrightarrow & d.j := d.g; \\
\mathcal{BA}'2_j :: & d.j \neq \perp \ \wedge \ f.j = \mathit{false} \ \wedge \ (d.k = \perp \ \vee \ d.k = d.j) \ \wedge & & \\
 & (d.l = \perp \ \vee d.l = d.j) \ \wedge (d.k \neq \perp \vee d.l \neq \perp) & \longrightarrow & f.j := \mathit{true}; \\
\mathcal{BA}'3_j :: & d.j = 1 \ \wedge \ d.k = 0 \ \wedge \ d.l = 0 \ \wedge \ f.j = \mathit{false} & \longrightarrow & d.j, \ f.j := 0, \ \mathit{false}|\mathit{true}; \\
\mathcal{BA}'4_j :: & d.j = 0 \ \wedge \ d.k = 1 \ \wedge \ d.l = 1 \ \wedge \ f.j = \mathit{false} & \longrightarrow & d.j, \ f.j := 1, \ \mathit{false}|\mathit{true}; \\
\mathcal{BA}'5_j :: & d.j \neq \perp \ \wedge \ f.j = \mathit{false} \ \wedge & & \\
 & ((d.j = d.k \wedge d.j \neq d.l) \ \vee \ (d.j = d.l \wedge d.j \neq d.k)) & \longrightarrow & f.j := \mathit{true};
\end{array}$$

Notice that action $\mathcal{BA}'1$ is unchanged, actions $\mathcal{BA}'3$ and $\mathcal{BA}'4$ are recovery actions, and actions $\mathcal{BA}'2$ and $\mathcal{BA}'5$ are strengthened actions, i.e., actions whose guard is restricted.

# 5 Variations of The Model Repair Problem

In Section 4, we illustrated the application of model repair in case of adding fault-tolerance to a distributed program. One characteristic of repairing distributed program was the notion of grouping of transitions in that the repaired program had to either include an entire group or exclude it entirely. It could not choose to include only a part of the grouped transitions. In this section, we identify different variations of the model repair problem based on the structure of a model, expressiveness of properties, the ability of a model to tolerate faults, and its exposure to physical processes. Specifically, we identify crucial requirements (such as grouping of transitions) introduced in model repair in these variations.

We describe the following variations. In Section 5.1, we identify the requirements of the model repair problem for distributed systems. Then, in Section 5.2, we discuss the problem where the model is subject to timing constraints. Section 5.3 focuses on different fault-tolerance requirements during repair. In Section 5.4, we present a relaxed variation of the problem where the set of legitimate states of the original program is not available (in the problem variation described in Subsection 5.3, this set is given as input).

We also note that the variations discussed in this section are not mutually exclusive. Specifically, we can consider model repair in a distributed, real time program where we need to add fault-tolerance while satisfying physical constraints. However, for such a problem, one would need to consider constraints imposed by distribution, timing constraints, fault-tolerance and physical constraints. Each of the subsequent section identifies constraints created due to individual variation. We discuss some of these combinations (e.g., fault-tolerance and real-time) where a sufficient set of results are available.

## 5.1 Distribution

As the example in Section 4 illustrated, a model $\mathcal{M}$ consists of a finite set of processes ($\mathcal{M}_1 \cdots \mathcal{M}_n$) working across a network or cluster of workstations. In such a setting, each process has only a partial view of the entire system. Specifically, each process is associated with a set of variables that it can read and a set of variables that it can write in one atomic step. These sets define *read/write restrictions* for each process. (Recall that in Section 4, this was achieved by specifying read/write restrictions on all non-general processes.)

As the example in Section 4 illustrated, for distributed programs, it is necessary that a model repair algorithm must respect read/write restrictions when repairing a process $\mathcal{M}_i$, $1 \leq i \leq n$, and hence, model $\mathcal{M}$. Otherwise, the algorithm may change the atomicity assumptions of $\mathcal{M}$. As illustrated in Section 4, this was achieved through grouping of transitions and requiring that the repaired program had to select a set of groups and it could not choose to include only a subset of transitions in that group.

## 5.2 Timing Constraints

In real-time systems, completion of tasks is often associated with deadlines. A timed model $\mathcal{M}$ is represented by a set of timed guarded commands (or theoretically a timed automaton [7]) and specification of the model is represented by Metric Temporal Logic (MTL) [8]. A timing constraint is a property of the form:

$$\Pi \equiv \Box(p \Rightarrow \Diamond_{\leq\delta} q),$$

where $p$ and $q$ are two logical propositions (i.e., state predicates), $\delta$ is an integer, $\Box$ is the temporal operator 'always', and $\Diamond$ is the temporal operator 'eventually'. This property means that when proposition $p$ holds, then proposition $q$ must hold within $\delta$ time units. In this context, a repair algorithm is normally applied to enforce new timing constraints (i.e., a property $\Pi$) while respecting existing deadlines (i.e., specification $\Sigma$). We note that if $\delta = \infty$, then the above property is called a *leads-to* property and means that if $p$ holds, then $q$ should *eventually* hold.

We note that there are variations of above timing constraint, e.g., $\Pi \equiv \Box(p \Rightarrow \Diamond_{\geq\delta} q)$, that have been considered in the literature. Intuitively, this constraint requires that if $p$ is true in some state then $q$ must eventually become true. However, it cannot become true before $\delta$ time units. We omit them here due to lack of substantial complexity results and/or algorithms for such constraints.

## 5.3 Fault-tolerance

Observe that in Section 4, the input to model repair consisted of a fault-intolerant program, faults, specification and the set of legitimate states. Such modeling can be applied in general. Next, we discuss issues in repairing a program to add different types of fault-tolerance requirements.

In the context of adding fault-tolerance a model $\mathcal{M}$ (given in any formal representation) can always be represented in terms of a set of states and transitions. A *fault* can simply be modeled by a transition in the state space of $\mathcal{M}$ that has not been anticipated in $\mathcal{M}$. A *fault model* $\mathcal{M}_f$ is then a state-transition system as well. We note that such representation is possible notwithstanding the type of faults (be they stuck-at, crash, fail-stop, timing, performance, Byzantine, message loss, etc.), the nature of the faults (be they permanent, transient, or intermittent), or the ability of the program to observe the effects of the faults (be they detectable or undetectable). An example of byzantine faults is achieved in Section 4. Moreover, in [27], authors have shown the feasibility of modeling different types of operational faults (as opposed to design faults) using transition systems. Thus, the model $\mathcal{M}$ in the presence of faults $\mathcal{M}_f$ can be obtained by parallel composition of $\mathcal{M}$ and $\mathcal{M}_f$, denoted $\mathcal{M}||\mathcal{M}_f$.

*Fault-tolerance* is the ability of a model $\mathcal{M}$ to satisfy its specification $\Sigma$ in the presence of $\mathcal{M}_f$. More formally, $\mathcal{M}$ is *fault-tolerant* to faults $\mathcal{M}_f$ if and only if $\mathcal{M}||\mathcal{M}_f$ satisfies its specification $\Sigma$. In order to distinguish the behavior of a model in the absence and presence of faults, we consider a notion of *legitimate states*; i.e., a set $L$ of states from where $\mathcal{M}$ always satisfies $\Sigma$ and never reaches a state not in $L$. The model $\mathcal{M}||\mathcal{M}_f$, however, may reach states outside $L$.

In its legitimate states, the fault-tolerant program is expected to satisfy its specification. However, if faults perturb it, it may satisfy a subset of the specification. A *level* of fault-tolerance is determined by the requirements that $\mathcal{M}||\mathcal{M}_f$ is required to satisfy. More formally, let $\Pi_s$ be the strongest *safety* property obtained from the specification $\Pi$[3]. We say that a model $\mathcal{M}$ is *failsafe* fault-tolerant for $\Pi$ if and only if $\mathcal{M}||\mathcal{M}_f$ satisfies $\Pi_s$; i.e., a failsafe system is only concerned with satisfaction of safety in the presence of faults and, hence, may deadlock or never reach its legitimate states in the presence of faults. Now, let $\Pi_r$ be a reachability property[4]. We say that a model $\mathcal{M}$ is *nonmasking* fault-tolerant for $\Pi$ if and only if $\mathcal{M}||\mathcal{M}_f$ satisfies $\Pi_r$; i.e., a nonmasking system is only concerned with eventual reachability of some desirable behavior (typically $\Pi$ but not required to be) in the presence of faults and, hence, may temporarily violate safety conditions in the presence of faults. The desirable behavior is typically the set of legitimate states. Hence, we have $\Pi_r \equiv \Box\Diamond L$. This property is known as *recovery*. Finally, we say that a model $\mathcal{M}$ is *masking* fault-tolerant for $\Pi \equiv \Pi_r \wedge \Pi_s$ if and only if $\mathcal{M}||\mathcal{M}_f$ satisfies $\Pi$; i.e., a masking system always satisfies its safety properties and eventually recovers to its legitimate states.

Given a model $\mathcal{M}$, a property $\Pi$, a set of legitimate states $L$, and a fault model $\mathcal{M}_f$, the repair problem in this context is to generate a model $\mathcal{M}'$, such that $\mathcal{M}'||\mathcal{M}_f$ satisfies $\Pi$ (and of course the existing specification $\Sigma$). We call this problem 'addition of fault-tolerance to $\mathcal{M}$'. The type of property $\Pi$ (safety, reachability, or both) determines the level of fault-tolerance to be added to $\mathcal{M}$. The set $L$ is the set of states from where execution of $\mathcal{M}$ is closed. An example of this set is the set of reachable states of $\mathcal{M}$.

In the context of fault-tolerance in real-time systems, we can consider three types of requirements in the presence of faults: untimed safety constraints (denoted hereafter as safety constraints for brevity), timing constraints and recovery to legitimate states (e.g., leads-to properties with/without time constraints) [15]. Hence, we can consider eight possible levels of tolerance by considering which subset of these three are satisfied. Since one of the levels corresponds to the case where none of these properties is satisfied, we can ignore it. Also, systems that satisfy timing constraints without satisfying the untimed safety constraints are not practical. Hence, five interesting levels of tolerance are applicable in the context of real-time models. Table 1 illustrates these levels of tolerance. *Soft* and *hard* fault-tolerance capture the notion of satisfaction of timing constraints in the absence and presence of faults. For example, hard-

[3]A safety property [5] can be characterized by a set of *bad* computation prefixes. That is, a set of state sequences that should not occur in any computation. In addition, given a specification, it can be expressed as an intersection of a safety property and a liveness property.

[4]Reachability property is a liveness property in [5].

|  | Untimed Safety Constraints | Timed Constraints | Recovery to Legitimate States |
|---|---|---|---|
| Soft-failsafe | Yes |  |  |
| Hard-failsafe | Yes | Yes |  |
| Nonmasking |  |  | Yes |
| Soft-masking | Yes |  | Yes |
| Hard-masking | Yes | Yes | Yes |

Table 1: Levels of fault-tolerance for real-time systems

failsafe fault-tolerance requires that in the presence of faults, the model guarantees the untimed safety constraints and timing constraints. However, it may not recover to legitimate states after the occurrence of faults. Moreover, a soft-masking model is one that satisfies timing-independent safety properties and recovery in the presence of faults (i.e., timing constraints are satisfied only in the absence of faults).

## 5.4 The Issue of Legitimate States

The issue of legitimate states is based on the observation that the input to byzantine agreement program from Section 4 consisted of a fault-intolerant program ($\mathcal{BA}1$ and $\mathcal{BA}2$), faults ($F_0$ and $F_1$), specification ($SPEC_{bt_{\mathcal{BA}}}$) and the set of legitimate states ($L_{\mathcal{BA}}$). Observe that the first three are essential for model repair problem in the context of adding fault-tolerance. Specifically, these three respectively identify the original model, faults that need to be tolerated and expectations in the presence of faults.

On the contrary, identifying the legitimate states from where the fault-intolerant program satisfies its specification is a difficult task. Our experience in this context shows that while identifying the other three arguments is often straightforward, identifying precise legitimate states requires significant effort. This motivates the idea of model repair in the context of addition of fault-tolerance where the input only consists of the fault-intolerant model, faults and the specification; i.e., the legitimate states are not given as input. In this context, there are several questions to address:

1. Is the new formulation relatively complete? In other words, if it is possible to perform model repair using the original problem formulation, is it guaranteed that it could be solved using the formulation with no legitimate states?

2. Is the complexity of both formulations in the same class?

3. Is the increased time cost, if any, small comparable to the overall cost of model repair?

These questions will be answered in Section 7.3.

# 6 Implementation Techniques and Scalability

As one can imagine, the problem of model repair is at least as difficult as the problem of model checking where one checks if the given model satisfies the given property. For this reason, it is necessary to develop heuristics to manage the complexity. Examples of some of these heuristics is discussed in Section 4. Additionally, it is necessary to develop and/or utilize highly efficient data structures. Moreover, even with heuristics and efficient data structures, it is necessary to evaluate bottlenecks involved in model repair. These bottlenecks can assist in making it easier to develop more efficient solutions.

Based on this, in this section, we discuss existing solutions and bottlenecks observed from them. Specifically, in Section 6.1, we present experimental results for model repair in distributed systems. Section 6.2 focuses on heuristics and implementation techniques for adding fault-tolerance to fault-intolerant models. In Section 6.3, we present heuristics that deal with model repair where the set of legitimate states is not available.

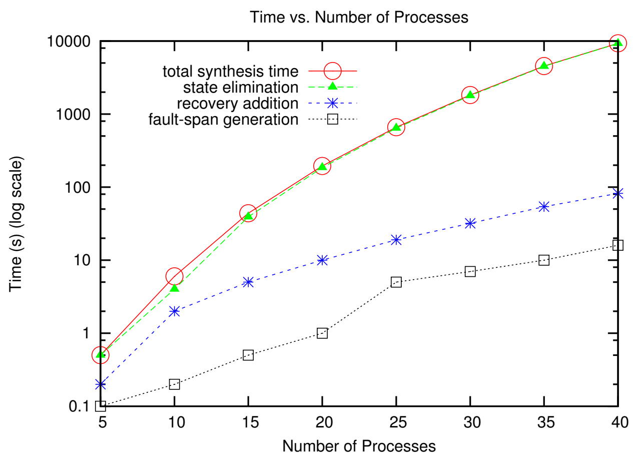
## 6.1 Distribution

By identifying the crux of the of NP-completeness result for adding UNITY properties to a distributed model [17], efficient heuristics are developed to add a *leads-to* property to a distributed model. As mentioned earlier, a leads-to property is of the form $\Box(p \Rightarrow \Diamond q)$, where $p$ and $q$ are two state predicates. Leads-to is a progress property and it captures many reachability constraints such as *recovery*. This (symbolic) heuristic is based on the BDD technology (binary decision diagrams) [21]. A BDD is a directed acyclic graph and makes testing of Boolean expressions, such as satisfiability and equivalence straightforward and extremely efficient. Using this heuristic has permitted synthesis of the recovery behavior in the well-known *Byzantine agreement* problem [46] and Dijkstra's *token ring* [30] for a large number of distributed processes (40 processes and beyond).

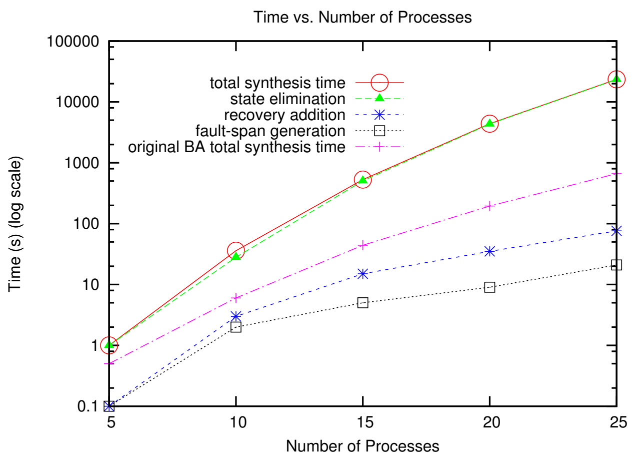
## 6.2 Fault-tolerance

Since addition of fault-tolerance to distributed models is NP-complete [17, 40, 43], a set of efficient heuristics are introduced in [16, 19, 41, 44] to tackle the intractability of the repair problem. The algorithm proposed in [41] can synthesize a rich class of fault-tolerant distributed models such as Byzantine agreement [46], token ring [9], and triple modular redundancy. This (explicit-state) algorithm is the backbone of the tool FTSyn [31]. The main drawback of this algorithm and the tool FTSyn is scalability. The symbolic heuristics proposed in [16, 19] mitigate the scalability problem. The BBD-based implementation of these heuristics has successfully synthesized classic problems in fault-tolerant distributed computing such as Byzantine agreement [46], token ring [9], and Byzantine agreement with fail-stop faults [54] with a large number of processes. The heuristics presented in [16, 19] are realized in the tool SYCRAFT [18]. This tool has also successfully synthesized sensor network protocols such as Infuse [42].

In the following discussion, we discuss the implementation of SYCRAFT [18], which uses the heuristics in Section 4. Specifically, the graphs in Figure 1 (in logscale) show the time required to synthesize a fault-tolerant version of the Byzantine agreement problem versus the number of processes (size of reachable states is beyond $10^{40}$). Note that the cost of synthesis is not incremental; i.e., each point on the graph shows the synthesis time of a particular number of processes from scratch. The graphs also analyze different bottlenecks involved during model repair. Based on the discussion in Section 4, we consider three tasks in the repair algorithm: *fault-span generation*, *adding recovery*, and *elimination of deadlocked states*. Fault-span generation refers to the amount of time spent to compute the set of reachable states in the presence of faults. State elimination is deadlock resolution (i.e., ensuring that the repaired model does not have reachable deadlock states). Recovery addition refers to the amount of time spent to synthesize recovery paths. As can be seen in the case of Byzantine agreement, deadlock resolution takes the majority of repair time. To the contrary, in Figure 2, the main bottleneck for synthesizing a fault-tolerant version of the token ring problem is reachability analysis (i.e., fault-span generation) and not deadlock resolution.



(a) Byzantine agreement



(b) Byzantine agreement with fail-stop and Byzantine faults

Figure 1: Experimental results for synthesizing Byzantine agreement and Byzantine agreement with fail-stop and Byzantine faults.

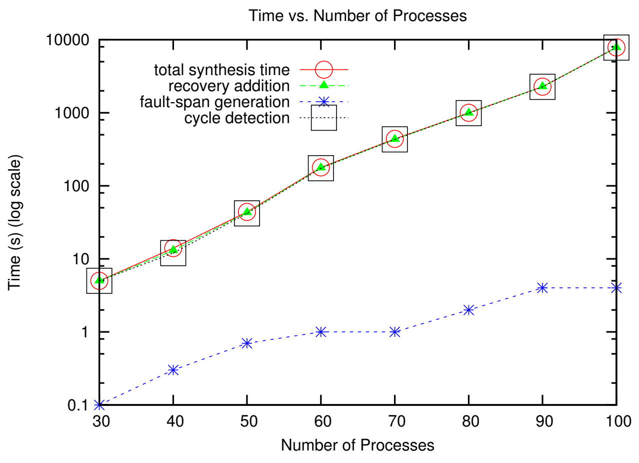
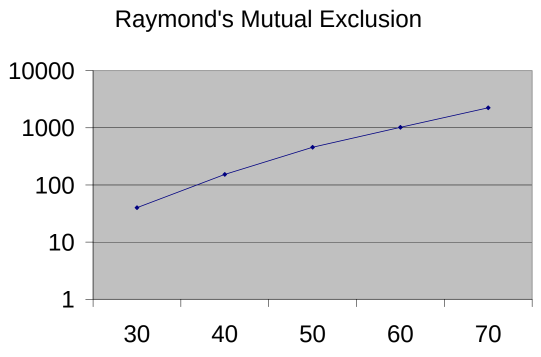
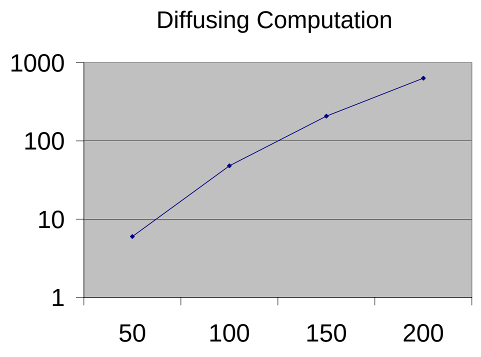


Figure 2: Experimental results for synthesizing fault-tolerant token ring.

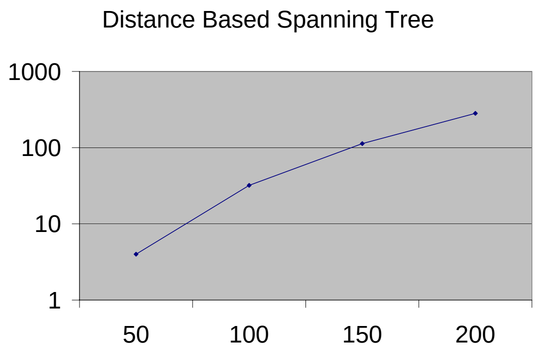
Another heuristic for model repair of distributed programs has focused on the design of nonmasking (or stabilizing) programs. Similar to a nonmasking program, a stabilizing program is guaranteed to recover to a legitimate state even if faults perturb it. In addition, this recovery is guaranteed from an arbitrary state. This heuristic relies on instances where the legitimate states are described by a set of conjunctive predicates, say $C_1, C_2, \ldots, C_n$. Therefore, if the fault perturbs the program outside its legitimate state, then it is necessary to restore it to a legitimate state; i.e., a state where all constraints, namely $C_1, C_2, \ldots, C_n$ are satisfied. With this motivation, this heuristic attempts to design recovery actions that satisfy individual constraints. One way to design recovery actions is to require that they do not violate any of the recovery constraints. However, this is too restrictive for many programs. Hence, one can create a (partial or total) order among the constraints and require that any recovery action that restores one constraint can violate any of the subsequent constraints although it cannot violate any of the preceding constraints. Thus, an action that restores the program to a state that satisfies $C_3$ can violate $C_4 \cdots C_n$. However, it cannot violate $C_1$ and $C_2$.



(a) Raymond's Mutual Exclusion



(b) Diffusing Computation



(c) Spanning Tree

Figure 3: Experimental results for synthesizing stabilizing programs: total synthesis times vs. number of processes.

In general, there are three steps required to apply such heuristic in practice. Given the set of legitimate states, the first step identifies suitable constraints whose conjunction equals the set of legitimate states. The second step identifies an order among constraints. Finally, the third step designs recovery actions to satisfy the constraints. In general, the first step is impossible to automate in its entirety. And, user input is expected to be crucial for the first step. The second step can be automated by considering different possible orders among constraints. The number of different orders that one needs to consider in the worst case is $O(n^2)$ where $n$ is the number of constraints, i.e., one does not need to consider $O(n!)$ orders. For several case studies however, considering a few, $O(1)$, constraints randomly suffices to identify a fault-tolerant program.

Once the order among constraints is identified, the third step can be automated fairly easily. Figure 3 shows experimental results for different case studies. Again, these results illustrate the feasibility of this heuristic for designing stabilizing programs.

The boundaries of scalability can be further expanded by designing distributed and parallel algorithms for different tasks involved in the repair process. Specifically, these include (1) algorithms for adding failsafe and masking fault-tolerance, to existing fault-intolerant centralized programs whose state space is distributed on a network or cluster of workstations [20], (2) algorithms for deadlock resolution in distributed models [1], and (3) algorithms for parallelizing constraint-based addition of fault-tolerance to distributed models [2].

### 6.3 Legitimate States

As discussed earlier, the set of legitimate states is one of the inputs to the model repair algorithm. The tool SYCRAFT [18] were designed with the set of legitimate states as one of the inputs. This is due to the fact that without the set of legitimate states, the complexity of partial repair can increase [3] from $P$ to NP-complete. One way to reduce this complexity is to utilize a heuristic where the input to model repair is utilized to construct a set of legitimate states. Specifically, given a model $\mathcal{M}$ and its specification $\Sigma$, this heuristic computes a set of legitimate states, say $L$ such that $\mathcal{M}$ satisfies $\Sigma$ from $L$. And, subsequently, utilize $\mathcal{M}$, $\Sigma$, $L$ and faults as the input to the tool that was described in Section 6.2.

If the computed set of legitimate states is the largest possible set from where $\mathcal{M}$ satisfies $\Sigma$, then this heuristic is relatively complete. In other words, given model $\mathcal{M}$, specification $\Sigma$ and faults $f$, if there exists a predicate $L$ such that the problem of model repair can be solved with inputs $\mathcal{M}$, $\Sigma$, $f$ and $L$, then this heuristic is guaranteed to work in the context of centralized models.

Hence, when this heuristic is applicable, the increased cost due to unavailability of the legitimate states corresponds to the time it takes to compute the largest set of legitimate states. It turns out that for many examples, the cost of computing this set of legitimate states is significantly less than the time it takes to perform model repair. Hence, when this heuristic is applicable, the increased cost of model repair is small (1% or less in several case studies) even if the set of legitimate states is not available explicitly. However, in the general case, the lack of availability of legitimate states can increase the cost substantially.

## 7 Complexity Results

As mentioned in Section 4, the distributed nature of the byzantine agreement problem requires one to model the issue of grouping of transitions during model repair. This, in turn, has made the problem of repairing distributed models NP-complete. In this section, we characterize the model repair problem in terms of its complexity in the context of several issues considered in Section 5.

The goal of this section is to identify the complexity of different instances of model repair considered in Section 5 to assist researchers interested in the topic of model repair to identify known hurdles (from complexity perspective) that need to be overcome in applying model repair in practice. In general, the results in Section 6 are based on utilizing the crux of the hardness result to develop heuristics. Also, the complexity analysis can also assist in *level of completeness* one can expect in tools designed for model repair. For example, when the complexity is low, it is expected that one could build tools that always succeed in repairing the given model. However, when complexity is high, one needs to build specialized tools; i.e., tools that are targeted towards specific domains (e.g., tools specialized for adding fault-tolerance to a specific type of fault), tools designed for specific types of properties etc.

Our classification follows a similar structure as in Section 5; i.e., in Section 7.1, we first identify the complexity results based on the need for repairing a distributed model. Subsequently, in Section 7.2, we discuss the complexity results when the model needs to be repaired for adding timing constraints. In

|  | Centralized | Distributed |
|---|---|---|
| Failsafe | P | NP-complete |
| Nonmasking | P | ? |
| Masking | P | NP-complete |
| Safety Property | P | NP-complete |
| Safety Property & One Leads-to Property | P | NP-complete |
| Two leads-to Properties | NP-complete | NP-complete |
| Safety (BP model) | NP-complete | NP-complete |

Table 2: Complexity comparison of model repair for centralized and distributed programs

Section 7.3, we evaluate the effect on complexity when legitimate states of the original program are not available.

## 7.1 Distribution

Here, we compare the cost of model repair of a centralized model, where all variables can be read and written atomically and the cost of model repair for a distributed model; i.e., a model that consists of several processes and each process can only read a subset of the variables. Table 2 provides this comparison. In most cases, the distribution causes the problem to be NP-complete [17, 40, 43]. The only exception is for the addition of nonmasking fault-tolerance, where the complexity result is currently unknown although it is conjectured that the problem cannot be solved in polynomial time. The problem of model repair can be solved in polynomial time in most instances except for the case where repair is performed for adding a safety property and two leads-to properties. Another instance is where the safety specification is represented by a set of *bad pairs* of transitions (as opposed to one bad transition) [45]. This model is denoted by the BP model in Table 2.

There are some surprising results in this table. In particular, the problem of adding a single leads-to property in distributed models is NP-complete even in the absence of faults. However, this result cannot be extended to nonmasking programs. This is because adding a leads-to property has a constraint that requires that new behavior be not added to the original model. However, in the context of fault-tolerance this requirement is imposed on fault-free behavior. In other words, while repairing a model to add fault-tolerance, it is essential that recovery can be added when faults occur. Hence, the requirement for preserving the behavior is only restricted to fault-free behavior. Thus, when adding a leads-to property (in the absence of faults), there is an implicit (safety) requirement that new transitions cannot be used.

Another surprising result is that repairing a centralized model to add a single leads-to property is achieved in polynomial time. However, adding two leads-to properties is NP-complete [12]. This is because a composition of the algorithm to add single leads-to property in a step-wise manner is incomplete. In other words, the choices made in adding the first leads-to property may make it impossible to add the second leads-to property. Hence, choices for both leads-to properties need to be considered collectively. This increases the complexity substantially.

## 7.2 Real-time

Here, we consider the case where the given model needs to be repaired for adding timing constraints. In this work, we assume that the original model is specified as a timed automaton [7] and the repaired

|  | Centralized | Distributed |
|---|---|---|
| Soft-failsfe | P | NP-complete |
| Hard-Failsafe | P | NP-complete |
| Nonmasking | P | ? |
| Soft-Masking | P | NP-complete |
| Hard-Masking | NP-complete | NP-complete |

Table 3: Complexity comparison for model repair for real-time programs

model is also a timed automaton. The timed automaton model is specified in terms of a set of locations and a set of clock constraints. A transition can simply increase the clock variables by a fixed value; i.e., all clock variables are increased by the same value. To evaluate the behavior of the timed automaton, we need to consider the idea of *region graph* [7], where the infinite state space of the timed automaton (caused due to real values of clock variables) into a finite set of equivalence classes.

The algorithms for repairing a model in timed utilize region (or more efficient versions thereof such as zone automaton [6]). However, since the focus of this section is complexity results and the complexity class is not affected by the choice, the results are presented in terms of these of the region graph.

As mentioned in Section 5, there are five possible levels of tolerance in this context. The complexity results for repairing a timed automaton model to add different levels of fault-tolerance are as shown in Table 3. In terms of centralized programs, repairing a program to add hard-masking fault-tolerance is NP-complete. However, the problem of model repair is in polynomial time for other levels of tolerance. For distributed programs, the results are similar to that of Section 7.1; i.e., except for the nonmasking fault-tolerance, the problem is NP-complete in the size of the region graph. We note that the complexities in Table 3 are in the size of a time-abstract bisimulation of the given model. In cases where the complexity is polynomial time in the size of the region graph, it is straightforward to show that the complexity is PSPACE-complete in the size of the given model [14, 15].

In the absence of fault, repairing a model with respect to a timing constraint or a simple timing-independent safety property can be done in polynomial time. However, adding a timing constraint $\Box(p \Rightarrow \Diamond_{\leq \delta}\, q)$ while preserving most behaviors of the original model is NP-complete (in the region graph) even if the original model satisfies $\Box(p \Rightarrow \Diamond q)$ [12, 13].

## 7.3 Legitimate States

In the context of adding masking and nonmasking fault-tolerance, it is necessary for the program to recover to legitimate states after faults stop occurring. One of the questions raised in the context of model repair is as follows: *If such legitimate states were not available explicitly during model repair then does it affect the complexity of repair?* Interestingly, the answer to this question depends upon the type of repair one is interested in. Specifically, we can consider two variations of model repair: (1) *total repair*, where it is required that the entire fault-free behavior is maintained during repair, and (2) *partial repair*, where a subset of fault-free behaviors is maintained during repair. Intuitively, in partial repair, one can remove behaviors that prevent one from obtaining fault-tolerance.

It turns out that the problem of partial repair can have higher complexity in certain settings than that for total repair. Specifically, the complexity of total repair is not affected by the lack of explicit legitimate states. However, the complexity of partial repair can increase substantially if explicit legitimate states are not available. Table 4 illustrates the complexity comparison for both types of repair.

| | | **Repair *Without* Explicit Legitimate States** | | **Repair *With* Explicit Legitimate States** | |
|---|---|---|---|---|---|
| | | **Partial** | **Total** | **Partial** | **Total** |
| High Atomicity | Failsafe | NP-complete | P | P | P |
| | Nonmasking | NP-complete | P | P | P |
| | Masking | NP-complete | P | P | P |
| Distributed Programs | Failsafe | NP-complete | NP-complete | NP-complete | NP-complete |
| | Nonmasking | NP-complete | ? | ? | ? |
| | Masking | NP-complete | NP-complete | NP-complete | NP-complete |

Table 4: Complexity comparison based on availability of legitimate states during repair

The intuitive explanation for the change in complexity is as follows: If there are some fault-free behaviors that conflict with the fault-tolerance requirements, then total repair is required to declare failure. However, with partial repair, one needs to determine which behaviors should be removed. Arbitrary approaches for removing fault-free behaviors do not work since they can result in removal of all behaviors thereby making the repaired model to have no behaviors.

With respect to distributed programs, however, the lack of legitimate states does not change the complexity class since the problem is already NP-complete. And, even without explicit legitimate states, it is trivial to solve the problem in NP.

## 8 Open Problems and Future Directions

Model repair is one step beyond formal verification. We believe that model repair is the next-generation technology in assuring system-wide correctness. There still exist numerous issues in automated model repair for further investigation. We categorize these issues into two groups of incremental open problems and future research directions.

**Incremental open problems** Some open problems aim at improving our existing methods and algorithms, or, solving the problem using alternative approaches.

- *(Employing techniques from model checking)* Currently, our only technique borrowed from model checking techniques is using symbolic techniques and BDDs. Other techniques whose application in model repair is non-trivial include SAT/QBF-based methods, abstract interpretation, symmetry reduction, partitioning, and partial order reduction.

- *(Complexity issues)* There are still open questions on the complexity of model repair. Examples include complexity of synthesizing self-stabilizing distributed models, synthesis of nonmasking real-time and distributed models, repairing component-based models using minimum number and size of added interactions, addition of multi-tolerance to real-time models, etc.

**New research directions** We now discuss areas of research where automated model repair has not been studied extensively.

- *(Compositional synthesis)* A line of research that has not been addressed is compositional synthesis, where different components along with their interfaces are automatically synthesized. This notion of compositional synthesis is perhaps more sensible in model repair, as some components may not require repair at all and resources can be directed to identifying and repairing components and/or interfaces where the error exists.

- *(Model repair for security policies)* Consider the infamous bug in the well-known Needham-Schreoder's authentication protocol, where an agent could be impersonated. This bug was identified by model checking the protocol in the presence of an intruder [48]. This example is not an isolated incident. In fact, according to veracode.com, 58% of existing software applications are vulnerable to cyber attacks that exploited the U.S. Department of Defense and Google. These examples and reports simply show the advantage of using model repair to deal with software application vulnerabilities. We intend to conduct extensive research in this area by exploiting the recent advances in reasoning about security policies (e.g., [29]) and develop next-generation security-aware compilers.

- *(Marrying model repair with other research areas)* An interesting line of research is to incorporate techniques from other disciplines of computer science to develop highly advanced model repair techniques. Examples include using machine learning and graph mining techniques, game theory and in particular the notion of Nash equilibrium, and biologically inspired methods.

- *(Knowledge-based model repair)* Another promising direction is applying knowledge-based techniques. The notion of knowledge in epistemic logic [34] is an elegant way to express the perception of a computing entity in a system about the entire system or other entities. Although there have been elegant results on reasoning about distributed systems using epistemic logic, the power of knowledge-based formalisms has not been extensively explored to ensure system-wide correctness. We plan to study how to express and reason about the correctness of multiple and often conflicting concerns (e.g., security, fault-tolerance, time-predictability, distribution) by considering the state of knowledge of agents in a system using epistemic logic.

- *(Probabilistic model repair)* Model repair for probabilistic systems has not been studied beyond the work in [10]. The goal in this problem is to repair a probabilistic system $\mathcal{M}$ with respect to a probabilistic temporal logic property $\Pi$, where $\mathcal{M}$ fails to satisfy $\Pi$, such that we obtain $\mathcal{M}'$ that satisfies $\Pi$ and differs from $\mathcal{M}$ only in the transition flows of those states in $\mathcal{M}$ that are deemed controllable. There are still numerous questions in this line of research such as preserving existing probabilistic properties, distribution, fault-tolerance, etc.

## 9 Acknowledgements

We would like to thank our collaborators, Fuad Abujarad, Anish Arora, Jingshu Chen, Ali Ebnenasir, and Yiyan Lin. We also thank Idit Keidar for her encouragements to write this article and her constructive comments during the review process.

# References

[1] F. Abujarad, B. Bonakdarpour, and S. S. Kulkarni. Parallelizing deadlock resolution in symbolic synthesis of distributed programs. In *Parallel and Distributed Methods in verifiCation (PDMC)*, 2009.

[2] F. Abujarad and S. S. Kulkarni. Multicore constraint-based automated stabilization. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 47–61, 2009.

[3] F. Abujarad and S. S. Kulkarni. Complexity issues in automated model revision without explicit legitimate states. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 206–220, 2010.

[4] K. Akesson, M. Fabian, H. Flordal, and A. Vahidi. Supremica a tool for verification and synthesis of discrete event supervisors. In *Mediterranean Conference on Control and Automation*, 2003.

[5] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[6] R. Alur, C. Courcoubetis, N. Halbwachs, D. L. Dill, and H. Wong-Toi. Minimization of timed transition systems. In *International Conference on Concurrency Theory (CONCUR)*, pages 340–354, 1992.

[7] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[8] R. Alur and T.A. Henzinger. A really temporal logic. *Journal of the ACM*, 41(1):181–204, 1994.

[9] 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.

[10] E. Bartocci, R. Grosu, P. Katsaros, C. R. Ramakrishnan, and S. A. Smolka. Model repair for probabilistic systems. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 326–340, 2011.

[11] R. Bloem, K. Chatterjee, T. A. Henzinger, and B. Jobstmann. Better quality in synthesis through quantitative objectives. In *Computer Aided Verification (CAV)*, pages 140–156, 2009.

[12] 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.

[13] B. Bonakdarpour and S. S. Kulkarni. Automated incremental synthesis of timed automata. In *International Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, LNCS 4346, pages 261–276, 2006.

[14] B. Bonakdarpour and S. S. Kulkarni. Automated revision of legacy real-time programs:work in progress. In *IEEE Real-Time and Embedded, Technology and Applications Symposium (RTAS)*, 2006.

[15] 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.

[16] 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.

[17] B. Bonakdarpour and S. S. Kulkarni. Revising distributed UNITY programs is NP-complete. In *Principles of Distributed Systems (OPODIS)*, pages 408–427, 2008.

[18] B. Bonakdarpour and S. S. Kulkarni. SYCRAFT: A tool for synthesizing fault-tolerant distributed programs. In *Concurrency Theory (CONCUR)*, pages 167–171, 2008.

[19] B. Bonakdarpour, S. S. Kulkarni, and F. Abujarad. Symbolic synthesis of masking fault-tolerant programs. *Distributed Computing*. To appear.

[20] B. Bonakdarpour, S. S. Kulkarni, and Fuad Abujarad. Distributed synthesis of fault-tolerant programs in the high atomicity model. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, LNCS 4838, pages 21–36, 2007.

[21] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[22] F. Buccafurri, T. Eiter, G. Gottlob, and N. Leone. Enhancing model checking in verification by ai techniques. *Artificial Intelligence*, 112:57–104, 1999.

[23] P. Cerný, K. Chatterjee, T. A. Henzinger, A. Radhakrishna, and R. Singh. Quantitative synthesis for concurrent programs. In *Computer Aided Verification (CAV)*, pages 243–259, 2011.

[24] K. M. Chandy and J. Misra. *Parallel program design: a foundation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[25] K. Chatterjee, T. A. Henzinger, B. Jobstmann, and R. Singh. QUASY: Quantitative synthesis tool. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 267–271, 2011.

[26] G. Chatzieleftheriou, B. Bonakdarpour, S. A. Smolka, and P. Katsaros. Abstract model repair. In *NASA Formal Methods Symposium (NFM)*, 2012. To appear.

[27] J. Chen and A. S. Kulkarni. Effectiveness of transition systems to model faults. In *Logical Aspects of Fault-Tolerance (LAFT)*, 2011.

[28] 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.

[29] M. R. Clarkson and F. B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[30] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ., 1990.

[31] A. Ebnenasir, S. S. Kulkarni, and A. Arora. FTSyn: a framework for automatic synthesis of fault-tolerance. *International Journal of Software Tools for Technology Transfer (STTT)*, 10(5):455–471, 2008.

[32] E. A Emerson. *Handbook of Theoretical Computer Science*, volume B, chapter 16: Temporal and Modal Logics, pages 995–1067. Elsevier Science Publishers B. V., Amsterdam, 1990.

[33] E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.

[34] R. Fagin, J.Y. Halpern nad Y. Moses, and M. Vardi. *Reasoning About Knowledge*. The MIT Press, 1995.

[35] L. Feng and W. M. Wonham. TCT: A computation tool for supervisory control synthesis. In *International Workshop on Discrete Event Systems*, pages 388–389, 2006.

[36] 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.

[37] B. Jobstmann and R. Bloem. *Lily - A LInear Logic Synthesizer*. http://www.ist.tugraz.at/staff/jobstmann/lily/.

[38] B. Jobstmann, S. Galler, M. Weiglhofer, and R. Bloem. Anzu: A tool for property synthesis. In *Computer Aided Verification (CAV)*, pages 258–262, 2007.

[39] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *Computer Aided Verification (CAV)*, pages 226–238, 2005.

[40] 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.

[41] S. S. Kulkarni, A. Arora, and A. Chippada. Polynomial time synthesis of Byzantine agreement. In *Symposium on Reliable Distributed Systems (SRDS)*, pages 130–140, 2001.

[42] S. S. Kulkarni and M. Arumugam. Infuse: A TDMA based data dissemination protocol for sensor networks. *International Journal on Distributed Sensor Networks (IJDSN)*, 2(1):55–78, 2006.

[43] S. S. Kulkarni and A. Ebnenasir. The complexity of adding failsafe fault-tolerance. *International Conference on Distributed Computing Systems (ICDCS)*, pages 337–344, 2002.

[44] S. S. Kulkarni and A. Ebnenasir. Enhancing the fault-tolerance of nonmasking programs. *International Conference on Distributed Computing Systems*, 2003.

[45] S. S. Kulkarni and A. Ebnenasir. Adding fault-tolerance using pre-synthesized components. In *European Dependable Computing Conference (EDCC)*, pages 72–90, 2005.

[46] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[47] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 1982.

[48] G. Lowe. Breaking and fixing the needham-schroeder public-key protocol using FDR. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 147–166, 1996.

[49] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(1):68–93, 1984.

[50] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Principles of Programming Languages (POPL)*, pages 179–190, 1989.

[51] P. J. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, 1989.

[52] K. Raymond. A tree based algorithm for mutual exclusion. *ACM Transactions on Computer Systems*, 7:61–77, 1989.

[53] R. Samanta, J. V. Deshmukh, and E. A. Emerson. Automatic generation of local repairs for boolean programs. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 1–10, 2008.

[54] R. D. Schlichting and F. B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computers*, 1(3):222–238, 1983.

[55] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. *ACM SIGPLAN Notices*, 41(11):404–415, 2006.

[56] W. Thomas. On the synthesis of strategies in infinite games. In *Theoretical Aspects of Computer Science (STACS)*, pages 1–13, 1995.

[57] Y. Zhang and Y. Ding. CTL model update for system modifications. *Journal of Artificial Intelligence*, 31:113–155, January 2008.