Distributed Computing Column 46

Synthesizing Distributed and Concurrent Programs

Idit Keidar
Dept. of Electrical Engineering, Technion
Haifa, 32000, Israel
idish@ee.technion.ac.il

Software synthesis is experiencing a renaissance. After years of being narrowly deployed in a few domains, automated program synthesis now appears to be ready for prime time. There are two main factors contributing to this trend. First, much related technology has matured to the point that synthesis is now feasible. In particular, program synthesis benefits from advances in verification, decision procedures, and machine learning, as well as the availability of more powerful computers. Second, with the increase in software complexity, automated program synthesis is now needed more than ever. The latter is particularly important in the domains covered by this column, namely distributed and concurrent systems. As distributed protocols are now increasingly deployed in clouds and large data centers, synthesizing correct ones becomes more important. Likewise, automated synthesis can serve instrumental in alleviating the programming challenge raised by the multi-core revolution, (which has been extensively discussed in past instances of this column).

Today's column includes two articles highlighting recent advances in automated synthesis in these two domains. Both consider the case where a given program was generated to work correctly in a certain model, but needs to be fixed to work in a different model, under less restrictive assumptions. In the context of distributed programs, Borzoo Bonakdarpour and Sandeep S. Kulkarni discuss automated model repair – a mechanism for automatically fixing bugs in a program or increasing its fault tolerance. They provide a broad survey of techniques developed in the last decade for model repair in distributed computing, fault-tolerance, self-stabilization, and real-time systems. Second, Michael Kuperstein, Martin Vechev, and Eran Yahav present an approach for automatic fence inference in concurrent programs, allowing them to run under relaxed memory models. Again, we are given a program that is correct in some restricted model – in this case, a sequentially consistent memory model – and the problem is to synthesize one that behaves correctly under a relaxed model, such as those implemented in today's computer architectures.

Many thanks to Borzoo, Sandeep, Michael, Martin, and Eran for their contributions!

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[*] of distributed systems that are correct by construction.

[*]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

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 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

commands, and Promela, etc. In some cases, the techniques used in this report can be applied to UML state diagrams or even C code.

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 [23, 11]. 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, Σ be a logical specification, where $\mathcal{M}$ satisfies Σ, and Π be a logical property, where $\mathcal{M}$ does not satisfy Π. The repair problem is to devise an algorithm that generates a model $\mathcal{M}'$ such that $\mathcal{M}'$ satisfies Σ and Π 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 Σ is not available during the repair process. In other words, one needs to utilize the fact that $\mathcal{M}$ satisfies Σ to deduce that $\mathcal{M}'$ satisfies Σ. 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 Σ. That is, all computations from these state satisfy Σ. In turn, one is required to identify the set of legitimate states (say, L') from where the repaired program satisfies Σ and Π.

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 Σ and have no knowledge about behavior of $\mathcal{M}$ in states outside L then we cannot construct $\mathcal{M}'$ that satisfies Σ 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 Π 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:

$V_{\mathcal{BA}} = \{d.g, d.j, d.k, d.l\} \cup$ (decision variables)

$\{f.j, f.k, f.l\} \cup$ (finalized?)
$\{b.g, b.j, b.k, b.l\}$. (Byzantine?)

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[†]:

$$\mathcal{BA}1_j \ :: \ (d.j = \perp) \wedge (f.j = \mathit{false}) \ \longrightarrow \ d.j := d.g;$$
$$\mathcal{BA}2_j \ :: \ (d.j \neq \perp) \wedge (f.j = \mathit{false}) \ \longrightarrow \ f.j := \mathit{true};$$

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:

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

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.

[†]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.

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:

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)

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

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:

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

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), δ 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 δ time units. In this context, a repair algorithm is normally applied to enforce new timing constraints (i.e., a property Π) while respecting existing deadlines (i.e., specification Σ). 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 δ 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 Σ 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 Σ. 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 Σ 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 Π_s be the strongest *safety* property obtained from the specification Π[‡]. We say that a model $\mathcal{M}$ is *failsafe* fault-tolerant for Π if and only if $\mathcal{M}||\mathcal{M}_f$ satisfies Π_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 Π_r be a reachability property[§]. We say that a model $\mathcal{M}$ is *nonmasking* fault-tolerant for Π if and only if $\mathcal{M}||\mathcal{M}_f$ satisfies Π_r; i.e., a nonmasking system is only concerned with eventual reachability of some desirable behavior (typically Π but not required to be) in the presence of faults and, hence, may temporarily violate safety conditions in the presence of

[‡]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.

[§]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

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 Π; i.e., a masking system always satisfies its safety properties and eventually recovers to its legitimate states.

Given a model $\mathcal{M}$, a property Π, 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 Π (and of course the existing specification Σ). We call this problem 'addition of fault-tolerance to $\mathcal{M}$'. The type of property Π (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-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.

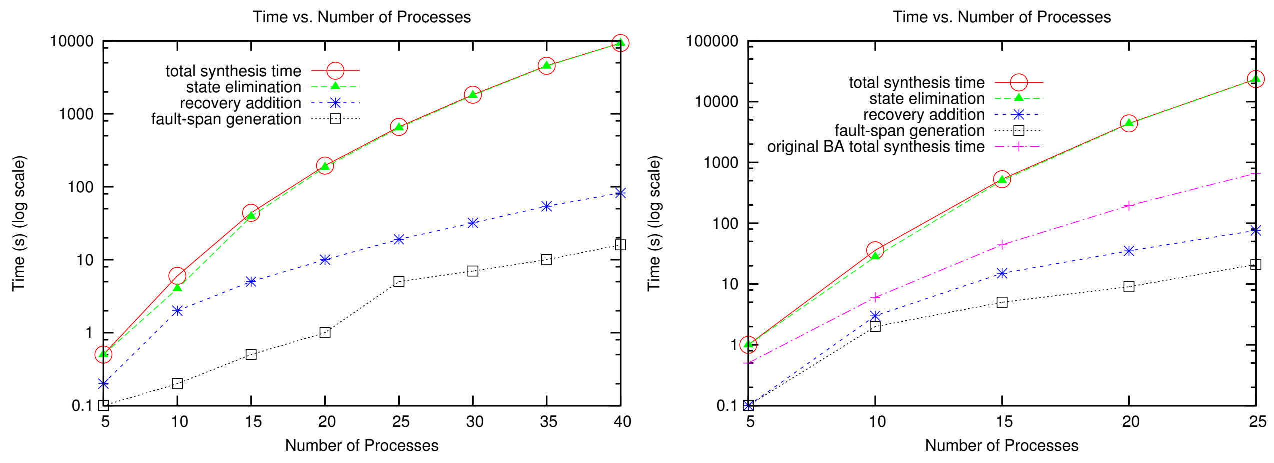
Based on this, in this section, we discuss existing solutions and bottlenecks observed from them. Specifically, in Section 6.1, we present experimental results for model repair in distributed systems. Section 6.2 focuses on heuristics and implementation techniques for adding fault-tolerance to fault-intolerant 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 [19, 16, 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



(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.

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 [19, 16] 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 [19, 16] 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.

Another heuristic for model repair of distributed programs has focused on the design of non-masking (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

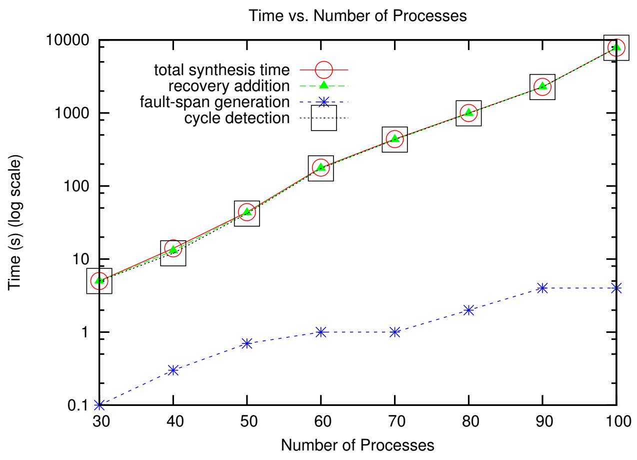


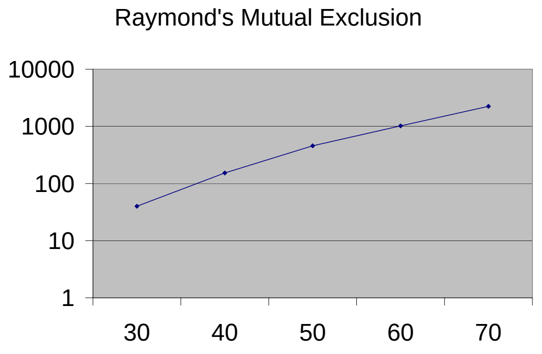
Figure 2: Experimental results for synthesizing fault-tolerant token ring.

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.

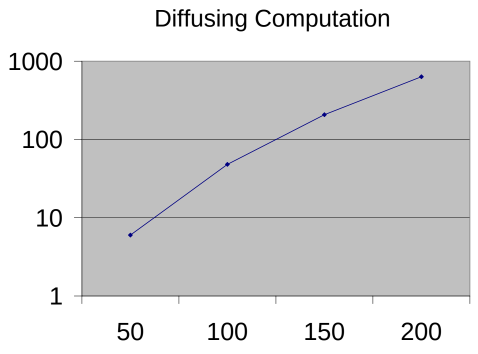
In general, there are three steps required to apply such heuristic in practice. Given the set of legitimate states, the first step identifies suitable constraints whose conjunction equals the set of legitimate states. The second step identifies an order among constraints. Finally, the third step designs 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.

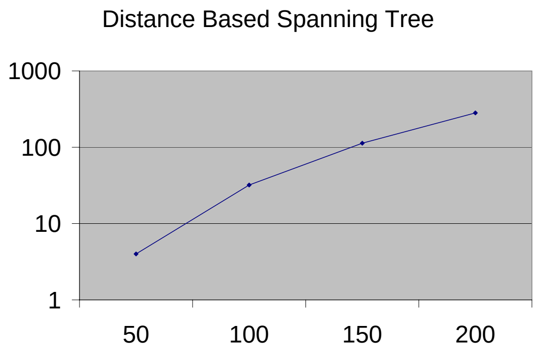
The boundaries of scalability can be further expanded by designing distributed and parallel algorithms for different tasks involved in the repair process. Specifically, these include (1) algorithms for adding failsafe and masking fault-tolerance, to existing fault-intolerant centralized programs whose 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].



(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.

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 Σ, this heuristic computes a set of legitimate states, say L such that $\mathcal{M}$ satisfies Σ from L. And, subsequently, utilize $\mathcal{M}$, Σ, 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 Σ, then this heuristic is relatively complete. In other words, given model $\mathcal{M}$, specification Σ and faults f, if there exists a predicate L such that the problem of model repair can be solved with inputs $\mathcal{M}$, Σ, 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 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.

	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

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 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

	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

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 $\Box(p \Rightarrow \Diamond q)$ [13, 12].

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.

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.

		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

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 Π, where $\mathcal{M}$ fails to satisfy Π, such that we obtain $\mathcal{M}'$ that satisfies Π 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.

Automatic Inference of Memory Fences

Michael Kuperstein
Dept. of Computer Science
Technion
mkuper@cs.technion.ac.il

Martin Vechev
Dept. of Computer Science
ETH Zürich
martin.vechev@inf.ethz.ch

Eran Yahav
Dept. of Computer Science
Technion
yahave@cs.technion.ac.il

Abstract

We addresses the problem of automatic verification and fence inference in concurrent programs running under relaxed memory models. Modern architectures implement relaxed memory models in which memory operations may be reordered and executed non-atomically. Instructions called *memory fences* are provided to the programmer, allowing control of this behavior. To ensure correctness of many algorithms, the programmer is often required to explicitly insert memory fences into her program. However, she must use as few fences as possible, or the benefits of the relaxed architecture may be lost. It is our goal to help automate the fence insertion process.

We present an algorithm for automatic inference of memory fences in concurrent programs, relieving the programmer from this complex task. Given a finite-state program, a safety specification and a description of the memory model our algorithm computes a set of ordering constraints that guarantee the correctness of the program under the memory model. The computed constraints are maximally permissive: removing any constraint from the solution would permit an execution violating the specification. These constraints are then realized as additional fences in the input program.

We implemented our approach in a pair of tools called FENDER and BLENDER and used them to infer correct and efficient placements of fences for several non-trivial algorithms, including practical mutual exclusion primitives and concurrent data structures.

1 Introduction

In 1979, in his seminal paper "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs" [18], Leslie Lamport defined the "sequential consistency" (SC) criterion for correctness of multiprocessor computers. Such a computer is called sequentially consistent if:

The result of any execution is the same as if the operations of all processes were executed in some sequential order, and the operations of each individual processor appear in the sequence in the order specified by its program.

It was Lamport's intent that any correct multiprocessor computer implementation must meet this criterion. However, modern hardware architectures are not, in fact, sequentially consistent. Instead, they implement so-called "relaxed" (or "weak") memory models (RMMs) [1]. These models enable improved hardware performance compared to sequentially consistent hardware [13]. This is achieved by allowing the CPU and memory subsystems to perform memory operations out of order and non-atomically. Unfortunately, this behavior poses an additional burden on the programmer. Even when the underlying architecture is sequentially consistent, highly-concurrent algorithms are notoriously hard to get right [24]. When programming for an architecture that implements an RMM, programmers must also reason about executions that have no sequential analogue. This reasoning is non-intuitive and may lead to subtle concurrency bugs.

To allow programmers avoid non-sequentially consistent executions, architectures provide special *memory fence* (also known as *memory barrier*) instructions. Very informally, a fence instruction restricts the CPU and memory subsystem's ability to reorder operations, thus eliminating some undesired non-SC executions. Finding a *correct and efficient* placement of memory fences for a given concurrent program is a challenging task. Using too many fences (over-fencing) hinders performance, while using too few fences (under-fencing) may allow unexpected incorrect executions to occur. Manually balancing between over- and under-fencing is very difficult, time-consuming and error-prone (cf. [14, 5]). Furthermore, the process of finding fences has to be repeated whenever the algorithm changes, and whenever it is ported to a different architecture.

As an example, consider the problem of implementing the Chase-Lev work-stealing queue [9] ("CL") on a relaxed memory model. Work stealing is a popular mechanism for efficient load-balancing used in runtime libraries for languages such as Java, Cilk [3] and X10. Fig. 1 shows an implementation of this data structure in C-like pseudo-code. For now, ignore the fence instructions that appear on unnumbered lines. CL maintains an expandable array of items called *wsq* and two indices *top* and *bottom*, initialized to 0. The queue is considered empty when $top \geq bottom$ When the queue is not empty, $top\%(queue \rightarrow size)$ points to the oldest element in the queue, while $bottom\%(queue \rightarrow size)$ points one past the newest element. The queue has a single owner thread that can only invoke the operations `push()` and `take()` which operate on one end of the queue, while other threads may call `steal()` to take items out from the opposite end. The queue can be dynamically expanded in response to a `push()` when additional space is required to store the item. This is done by the `push()` operation invoking the `expand()` procedure. For simplicity, we assume that items in the array are integers and that memory is collected by a garbage collector (manual memory management presents orthogonal challenges, cf. [23]).

We would like to guarantee that there are no out of bounds array accesses, no items are lost (by being overwritten before being read), and no "phantom" items are read after being removed. All these properties hold for the CL queue under the sequentially consistent memory model. However, they may be violated when it is used under a relaxed model.

Under weak memory models, e.g. the SPARC RMO [28] memory model, some of the memory operations in the code may be executed out of order. Tab. 5 shows possible RMO re-orderings that lead to violation of the specification. The column *locations* lists the two lines in a given method which contain memory operations that might get reordered and lead to a violation. The next column gives an example of an undesired effect when the operations at the two labels are reordered. The last column shows the type of fence that can be used to prevent the undesirable reordering. Informally, the type describes what kinds of memory operations

#	**Locations**	**Effect of Reorder**	**Needed Fence**
1	push:8:9	`steal()` returns phantom item	store-store
2	take:4:5	lost items	store-load
3	steal:2:3	lost items	load-load
4	steal:3:4	array access out of bounds	load-load
5	steal:7:8	lost items	load-store
6	expand:9:10	`steal()` returns phantom item	store-store

Table 5: Potential reorderings of operations in the Chase-Lev algorithm of Fig. 1 running on the RMO memory model

```
1   typedef struct {
2     long size;
3     int  *ap;
4   } item_t;
5
6   long top, bottom;
7   item_t *wsq;
```

```
1   void push(int task) {
2    long b = bottom;
3    long t = top;
4    item_t* q = wsq;
5    if (b-t ≥ q→size-1){
6      q = expand();
7    }
8    q→ap[b % q→size]=task;
     fence("store-store");
9    bottom = b + 1;
10  }
```

```
1   int take() {
2    long b = bottom - 1;
3    item_t* q = wsq;
4    bottom = b;
     fence("store-load");
5    long t = top;
6    if (b < t) {
7      bottom = t;
8      return EMPTY;
9    }
10   task = q→ap[b % q→size];
11   if (b > t)
12     return task;
13   if (!CAS(&top, t, t+1))
14     return EMPTY;
15   bottom = t + 1;
16   return task;
17  }
```

```
1   int steal() {
2    long t = top;
     fence("load-load");
3    long b = bottom;
     fence("load-load");
4    item_t* q = wsq;
5    if (t ≥ b)
6      return EMPTY;
7    task=q→ap[t % q→size];
     fence("load-store");
8    if (!CAS(&top, t, t+1))
9      return ABORT;
10   return task;
11  }
```

```
1   item_t* expand() {
2     int  newsize = wsq→size * 2;
3     int* newitems = (int *) malloc(newsize*sizeof(int));
4     item_t *newq =  (item_t *)malloc(sizeof(item_t));
5     for (long i = top; i < bottom; i++) {
6       newitems[i % newsize] = wsq→ap[i % wsq→size];
7     }
8     newq→size = newsize;
9     newq→ap = newitems;
      fence("store-store");
10    wsq = newq;
11    return newq;
12  }
```

Figure 1: Pseudo-code of the Chase-Lev work stealing queue

have to complete before other type of operations. For example, a store-load fence executed by a processor forces all stores issued by that processor to complete before any new loads by the same processor start.

For a more detailed example of the effect the memory model has on execution, we consider the failure described in line 2 of Tab. 5. This corresponds to a reordering of operations at lines 4 and 5 in the `take()` method: if these two lines are reordered, the read from *top* is executed before the write to *bottom*. The failure scenario involves one process running the `steal()` method in parallel to another processes running a sequence of `take();push();take();push()` as follows:

1. Initially the queue has one item with $top = 0$ and $bottom = 1$.
2. A `take()` reads *top* and gets preempted before executing line 6.
3. An entire `steal()` executes, correctly returns the item at index 0, and advances *top* to 1.
4. The `take()` resumes and succeeds, returning the same item as the previous `steal()`, setting *bottom* to 0.

5. A complete `push()` now pushes some item i.
6. A complete `take()` executes and returns `EMPTY` instead of item i.
7. A complete `push()` executes and overwrites item i (losing item i).

To guarantee correctness under RMO, the programmer can try to manually insert fences that avoid undesirable behavior. As an alternative to placing fences based purely on her intuition, the programmer may also use a tool such as CheckFence [5] that can check the correctness of a given fence placement. However, repeatedly adding fences to avoid each counterexample can easily lead to over-fencing: a fence used to fix a counterexample may be made redundant by another fence inferred for a later example. In practice, localizing a failure to a single reordering is challenging and time consuming as a single failure trace might include multiple instances of non-SC behavior. Furthermore, a single reordering can be exhibited as multiple failures, and it is sometimes hard to identify the cause underlying an observed failure trace.

In a nutshell, the programmer is required to manually produce Tab. 5: summarize and understand all counterexamples from a checking tool, localize the cause of failure to a single reordering, and propose a fix that eliminates the counterexample. Further, this process might have to be repeated manually every time the algorithm is modified or ported to a new memory model. Even a subtle change in the algorithm may require a complete re-examination.

It is easy to see that the process of manual fence inference does not scale. In this paper we present an algorithm that *automatically* infers *correct and efficient* fence placements for *finite-state* programs. Our inference algorithm is defined in a way that makes the dependencies on the underlying memory model explicit. This makes it possible to use our algorithm with various memory models. To demonstrate the applicability of our approach, we implemented a relaxed memory model that supports key features of several modern RMMs.

Requiring the input program to be finite-state means we must overcome several challenges for the algorithm to be practical. First, this requirement means the algorithm, taken as is, is not suitable for fence inference in open systems (such as library implementations). This is in contrast to our goal to apply the algorithm to concurrent data structures. To formally verify that a data structure meets a specification (and, consequentially, to infer a correct fence placement), one generally needs to verify the "most general client" which is usually not finite-state. We deal with this in a manner similar to other related work in the field (e.g., Burckhardt et al. [5]) by using representative clients. Another problem is that even if a program is finite state under sequential consistency it will often not be finite-state under a relaxed model. As this phenomenon is common in practice, a direct implementation of our algorithm fails to infer fences for many interesting programs. To solve this problem we developed the concept of *abstract memory models* (AMMs). Very informally, an abstract memory model is an over-approximation of a relaxed memory model, in the sense that any program behavior possible in the RMM is also possible in the abstract model. Our abstract memory models are designed so as a program that is finite-space under SC remains finite state under the AMM. By utilizing AMMs, we can use our algorithm for any program that is finite-space under SC. More detail on AMMs can be found in [17] and [15].

In this paper, we describe an algorithm that automatically infers a correct and efficient placement of memory fences in finite-state concurrent programs. The paper is based on work previously published as [16] and [15].

2 Fence Inference

2.1 Constraint Generation

We first present our inference algorithm in a general setting, without instantiating it for a specific memory model. We then prove that when properly instantiated, it is correct and optimal.

Goal The input to the algorithm is a finite-state program P, a safety specification S, and an operational description of the memory model M. We assume that P satisfies S under sequential consistency but not necessarily under M. The output of the algorithm is a new program P', that satisfies S under M, which is obtained by adding memory fences to P. For "reasonable" memory models, this problem always has a trivial solution, as placing a fence between every two memory operations will reduce the possible executions to those allowed under SC. Therefore, we also add an optimality constraint: we would like the program P' to have performance as close to the original program as possible. In other words, since there may be many possible ways to "fix" P by adding fences, we want to choose only the best solution(s).

Algorithm Structure Our algorithm follows these three steps:

1. Construct the transition system for P under M.
2. Find the set of "error states" E violating S in the transition system.
3. Compute a set of program locations in P s.t. adding fences in those locations would "cut off" the error states, and output a program P' with fences added in these locations.

Steps 1 and 2 are standard in the world of software verification (in particular, software model checking [10]). The focus of this work, and the "heart" of our algorithm is therefore step 3. The general approach we use (similarly to the work of Vechev et al. [33]) is not to try to compute fence locations directly, but use an intermediate constraint language.

The high-level idea is that we first choose a constraint language F and associate with every transition t some constraint $\chi(t)$ from F. Very informally, we say that a constraint $\chi(t)$ is *enforceable* if we can "cut the transition t off" from the transition system by adding (syntactic) fences to the program. We call adding such fences *enforcing* the constraint.

We can use this idea to break the problem down into the 3 following major sub-steps: constraint *generation*, *solving* and *implementation*.

1. (Generation) Compute a boolean formula ψ over the constraints that represents *all* of the ways to cut off all error states in E (that is, make them unreachable in the transition system).
2. (Solving) Find a minimal satisfying assignment to ψ. This gives us a minimal constraint set δ that, if enforced, will cut off all error states. Note that it is possible that there are several such minimal constraint sets.
3. (Implementation) Transform δ into a fence placement that enforces those constraints. Again, there may be many ways to implement the constraints as fences.

For this scheme to work, we need to compute ψ such that every satisfying assignment (constraint set) δ of ψ satisfies the following two properties:

- Every constraint in the set δ are enforceable using memory fences inserted into the program code of P.
- If all constraints in δ are enforced by inserting memory fences into P (creating a new program P'), then P' does not violate the specification S.

Additionally, we want the computed constraint formula ψ to be maximally permissive: a constraint set satisfies the formula *if and only if* enforcing it will make the modified program adhere to the specification. This means a minimal satisfying assignment δ of ψ represents a "globally minimal" constraint set: it is impossible to fix the program by enforcing only a strict subset of δ.

Transition System Construction The first stage of the algorithm is to construct the transition system (TS) for the program. The transition system is a graph that consists of vertices which represent the states that can be reached by running the program and edges that represent state transitions. The notation we use for the transition system of the program P is $\langle \sigma_0, \Sigma_P, T_P \rangle$, where Σ_P is the set of states, T_P is the set of transitions, and σ_0 the initial state. The transitions link a state to all of its possible successor states. Here, we assume the input program P is *finite-state under the memory model*. This means that when P is executed under the memory model, there is only a finite number of reachable states. Note that if a program is finite-state under sequential consistency, this does not imply it is also finite-state under a more relaxed model.

Since our algorithm is designed to work with *operational* memory models, given a program state σ we are able to directly compute its set of successors. This means we can construct the transition system iteratively, using a standard worklist algorithm.

Marking Error States Once the TS is constructed, we can identify a subset of *error states*: the set of states that violate the provided safety specification. It is well known that every LTL safety property ϕ can be expressed as Gp where p is a "past-formula", that is, a formula that only refers to the past of the computation. We further assume that that the specification is given as Gp where p is a state property - for example, an assertion on the values of program variables. This restriction is introduced for two reasons:

1. While our algorithm is sound in the general case, it is no longer necessarily optimal. This is because for a general past-formula, fixing the program may be possible not only by cutting off the error states themselves, but by cutting off some of their predecessors. It is possible that our algorithm can be extended to this case, but we did not explore this possibility in detail.
2. This restriction allows us to check whether a state is an error state immediately after we encounter it during exploration. This allows us not to explore any of the error states' descendants, which improves the algorithm's performance.

As many practically useful safety properties can be expressed as Gp where p is a state property, we believe this to be a reasonable restriction.

Constraints Our goal is to transform an input program P into a new output program P' that satisfies S. At this stage, it is convenient to "abstract away" the two programs and focus purely on transition systems. Given a transition system $\langle \sigma_0, \Sigma_P, T_P \rangle$ under a memory model M, we can identify some transitions as *avoidable* and others as *unavoidable*. A transition $t = \sigma_1 \longrightarrow \sigma_2$ is considered avoidable if it is possible to construct a program P' by adding fences to P s.t. $\langle \sigma_0, \Sigma_{P'}, T_{P'} \rangle$ does not contain a transition that corresponds to t. Since discussing two separate transition systems (for P and P') is cumbersome, we informally refer to this process as *cutting* t from the transition system $\langle \sigma_0, \Sigma_P, T_P \rangle$.

More practically, we need to pick (according to the memory model) a set of constraints such that every such constraint can be *enforced* by adding memory fences to P. We then associate with every transition $\sigma_1 \xrightarrow{t} \sigma_2$ a set of constraints $\chi(t)$, that satisfies the following properties:

- If at least one of the constraints in $\chi(t)$ is enforced, then σ_2 is no longer reachable from σ_1 in P'.
- If none of the constraints are enforced, and σ_1 is reachable in P', then σ_2 is also reachable.

In other words, $\chi(t)$ precisely captures *all the ways* to cut the transition t.

This means a transition is avoidable if and only if $\chi(t)$ is non-empty. We can "lift" this definition from transitions to program traces and states: An avoidable trace is a trace in which *at least one* transition is avoidable, and an avoidable state is a state such that *all* program traces leading to it are avoidable.

Given a transition system and a specification we wish to find a constraint set which would cut all traces leading to error states. One possible approach, in the spirit of previous work by Vechev et al. [33], is to enumerate all (acyclic) traces leading to an error state and try to prevent each trace by enforcing appropriate constraints. However, such enumeration does not scale to practical programs as the number of traces can be exponential in the size of the transition system which is itself exponential in program length. Instead, our algorithm works on a state-by-state basis by assigning an *avoid formula* to each state. The avoid formula of a state captures all the ways to cut that state from the transition system.

Suppose we want to cut the state σ. Let the incoming transitions of σ be $t_1, \ldots, t_k$, with source states $\sigma_1, \ldots, \sigma_k$ respectively. To cut σ, we must make it unreachable through all of its incoming transitions. For each transition t_i, this means either cutting t_i itself or removing the source state σ_i. More concretely, we must either enforce some constraint in $\chi(t_i)$ or recursively find the avoid formula for σ_i and enforce some satisfying assignment of that formula. This is in fact recursive only if the transitions system $\langle \sigma_0, \Sigma_P, T_P \rangle$ is acyclic - if it contains cycles, the avoid formula of σ_i may itself depend on the avoid formula of σ. This suggests that the desired avoid formula for a state is a fixed point of a function that relates the avoid formula of a state to those of its predecessor states.

Constraint Generation Algorithm Now that the definitions are in place, we can present the algorithm used for the constraint formula generation phase. Instead of dealing directly with formulae, we will for convenience present the algorithm in terms of boolean functions ("avoid functions"). We will, however, often abuse notation and identify boolean variables with atomic propositions, and monotone boolean functions with monotone propositional formulae that define those functions. In particular:

- For a function f and an assignment of values to variables δ we will use $\delta \models f$ to mean that $f(\delta) = \mathbf{tt}$.
- For functions f_1, f_2, we use $f_1 \implies f_2$ to mean $f_1 \sqsubseteq f_2$, $f_1 \vee f_2$ to mean $f_1 \sqcup f_2$, etc.

Let $\mathbb{V}$ be a set of variables, representing possible constraints on execution. Let F be the set of monotone boolean functions over $\mathbb{V}$ with the standard order relation (also known as the free distributive lattice over $\mathbb{V}$).

Let $\langle \sigma_0, \Sigma_P, T_P \rangle$ be a transition system and σ_0 the initial state. Then a legal *labeling function* is a function $L : \Sigma_P \to F$, such that $L(\sigma_0) = \mathbf{ff}$. Intuitively, the labeling function L attaches an avoid function to a state. We require $L(\sigma_0)$ to always be false as the initial state can never be avoided. For a given transition system $\langle \sigma_0, \Sigma_P, T_P \rangle$, we denote by Λ_P the set of all legal labeling functions for that transition system.

Given a labeling function L and a state $\sigma \in \Sigma_P$, we define:

$$avoid(L, \sigma) = \bigwedge \{L(\mu) \vee \chi(t) \mid (\mu \xrightarrow{t} \sigma) \in T_P\}$$

This formalizes the previously presented intuition: given a labeling L, to avoid a state σ we must avoid all incoming transitions $\mu \xrightarrow{t} \sigma$, either by cutting t (using $\chi(t)$) or by enforcing $L(\mu)$. In the above definition, we abuse notation by treating $\chi(t)$ not as a subset of $\mathbb{V}$, but rather as the function represented by $\bigvee \{p \mid p \in \chi(t)\}$. We will continue this abuse throughout this paper. Whether χ is treated as a set or a function should always be clear from the context.
Using *avoid* we define an operator $trans : (\Sigma_P \to F) \to (\Sigma_P \to F)$ that updates the labeling to the "next generation" of avoid functions:

$$trans[L] = \lambda\sigma \in \Sigma_P . L(\sigma) \wedge avoid(L, \sigma)$$

If L is legal, then so is $trans(L)$ because:

- $trans[L](\sigma_0) = (L(\sigma_0) \wedge ...) = \mathbf{ff}$
- $avoid(L, \sigma)$ is monotone, as is $L(\sigma)$, and a conjunction of two monotone functions is also monotone.

The algorithm to find the desired labeling function is now very simple: we take the initial labeling function L_0 defined below, and iteratively apply *trans* until a fixed point is reached.

$$L_0 = \lambda\sigma \in \Sigma_P . \begin{cases} \mathbf{ff} & \text{if } \sigma = \sigma_0 \\ \mathbf{tt} & \text{if } \sigma \neq \sigma_0 \end{cases}$$

¿From this point on, we refer to the L function to which the fixed point computation converges as av.

However, directly applying this algorithm is inefficient, for two reasons. First, it requires maintaining two copies of the transition system. More importantly, a lot of unnecessary computation is performed because it is possible that in every application of *trans* only few $L(\sigma)$ values actually change. Therefore we use an optimized version based on the standard "chaotic iteration" method due to Cousot & Cousot [11]. This version is shown in Algorithm 1.

Lines 2-4 of the algorithm set the initial labeling to L_0. The labeling is then updated in the following fashion. First, the entire transition system is added to a workset. Then, if the worklist is not empty we pick an arbitrary state σ, and update it from $L(\sigma)$ to $trans[L](\sigma)$ (lines 8 - 10). We then check whether $L(\sigma)$ was changed by the application of *trans*. If it has, we may need to update the labeling of its descendant states, so we add all descendants of σ to the workset. When the workset becomes empty, a fixed point has been reached, so we can return the conjunction of constraints for the error states.

Example Consider the simple concurrent program shown in Fig. 2(a). X and Y are integer variables shared between processes A and B, while R_1 and R_2 are integer variables local to process B. For illustrative purposes, the memory model we use here is a simplified version of RLX (described formally in [16]). In this model any two (data and control) independent instructions can be reordered. However, as opposed to full RLX, stores to shared memory are preformed atomically.

To model the RMM effects, RLX uses "execution buffers" — similar to the "reordering box" of [26] and "local instruction buffer" of [34]. Informally, every processor p processes its instruction stream in its original order. However, "processing" does not in fact mean the instruction is executed. Rather, every instruction is placed in a buffer $E(p)$. An instruction is actually executed when it is removed from the buffer by the environment. If the buffers behave in a FIFO fashion, this is equivalent to sequential consistency. However if the executed instruction is not necessarily the oldest in the buffer, relaxed behavior occurs. In this framework, different RMMs can be specified by providing different rules for removing instructions from

ALGORITHM 1: Constraint Generation

```
Input: Program P, Specification S, Memory Model M
Output: Program P' satisfying S
1  compute $\langle \sigma_0, \Sigma_P, T_P \rangle$ under the memory model M
2  $L(\sigma_0) \leftarrow false$
3  foreach state $\sigma \in \Sigma_P \setminus \{\sigma_0\}$ do
4      $L(\sigma) \leftarrow true$
5  workset$\leftarrow \Sigma_P \setminus \{\sigma_0\}$
6  while workset is not empty do
7      $\sigma \leftarrow$ select and remove state from workset
8      $\varphi \leftarrow L(\sigma)$
9      foreach transition $t = (\mu \longrightarrow \sigma) \in T_P$ do
10         $\varphi \leftarrow \varphi \wedge (L(\mu) \vee \chi(t))$
11     if $L(\sigma) \not\equiv \varphi$ then
12         $L(\sigma) \leftarrow \varphi$
13         add all successors of $\sigma$ in $\Sigma_P$ to workset
14 $\psi \leftarrow \bigwedge\{L(\sigma) \mid \sigma \nvDash S\}$
15 return $\psi$.
```

the buffer. A complete definition of RLX semantics can be found in [16, 15].

The constraint language we use in the example consists of constraints on execution order. The constraint $[L_1 \prec L_2]$ where L_1 and L_2 are program labels means we forbid L_2 to bypass L_1. That is, if $[L_1 \prec L_2]$ is enforced, and L_1 precedes L_2 in program order, then L_1 must be executed before L_2. We provide more details about constraints later in this section.

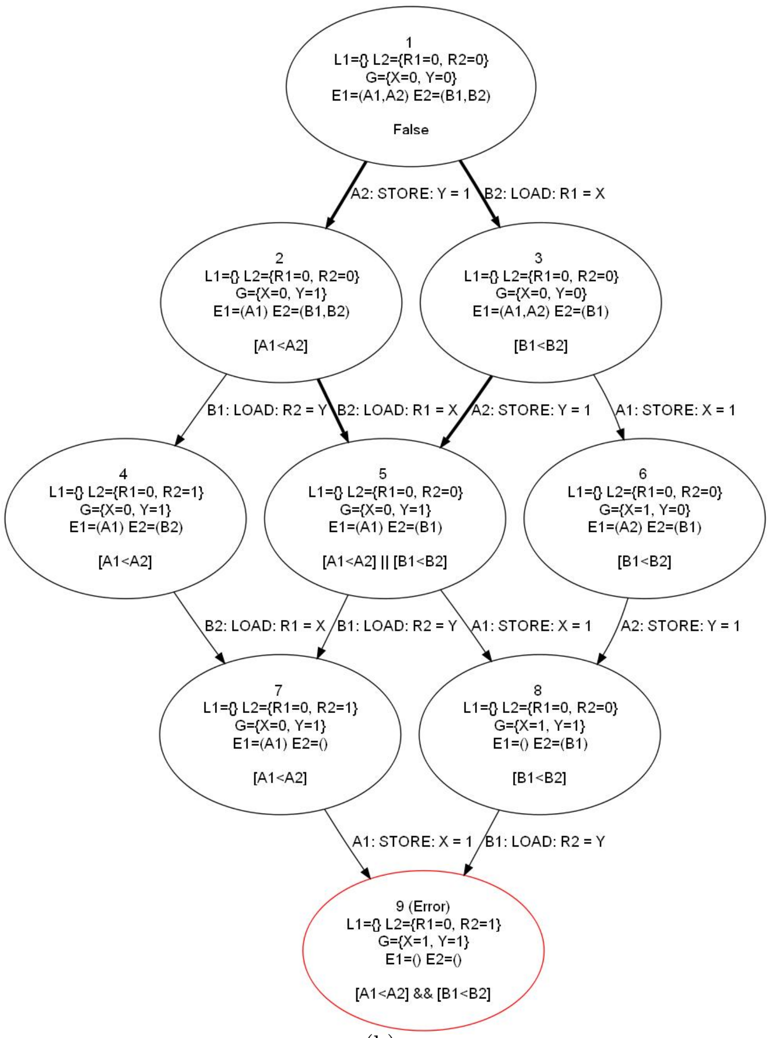
Fig. 2(b) shows part of the transition system of this program running on this specific memory model. We only show states that can lead to an error state, as the rest of the transition system is not relevant to the example. Inside each state in the figure we show: (i) assignments to the local variables of each process (L_1 and L_2), and the global variables G; (ii) the execution buffer of each process (E_1 and E_2); (iii) the (final) avoid formula of the state. Since stores are atomic, we do not show B_σ.

For this program, our specification is that $R_1 \geq R_2$ in the program's final state. In the initial state (state 1) all four variables have the value 0. The transition system also contains a single error state (state 9) where $R_1 = 0$ and $R_2 = 1$ (state 9). Since the transition system is acyclic, we can find $av(\sigma)$ by topologically sorting the states, and then computing av once for each state. For example:

- Since state 1 is the initial state, the avoid formula is necessarily **ff**.
- The avoid formula for state 2 is computed by taking the disjunction of avoiding the transition A_2 and avoiding the source state of the transition (state 1). To do so we first need to know $\chi(1 \longrightarrow 2)$. Informally, we need to know whether A_2 is executed out of order, and which alternative instructions could have been executed by A instead. If we examine the execution buffer E_1 of state 1 and look at the instructions that precede A_2, we find that A_2 is executed out of order, and that A_1 precedes it in the buffer. This implies we can enforce the constraint $[A_1 \prec A_2]$ as a way to avoid the transition A_2. Since the source state (state 1) cannot be avoided, the avoid formula for state 2 is simply $[A_1 \prec A_2]$. The formula $[B_1 \prec B_2]$ for state 3 is obtained similarly.
- The transition from state 2 to state 4 is taken "in order", that is, it doesn't violate any enforceable constraint. Therefore, the transition itself cannot be avoided and the only way to avoid reaching 4 is through enforcing the avoid formula of its predecessor, state 2. So the avoid formula of state 4 is also $[A_1 \prec A_2]$.
- State 5 has two incoming transitions: B_2 and A_2. B_2 is taken out of order from state 2 and can be prevented by enforcing the constraint $[B_1 \prec B_2]$. The constraint for the source state 2 is $[A_1 \prec A_2]$, so

```
        R1 = R2 = X = Y = 0;
       A:                       B:
A1: STORE X = 1        B1: LOAD R1 = Y
A2: STORE Y = 1        B2: LOAD R2 = X
```

(a)



(b)

Figure 2: An example program (a) and its partial transition system (b). Avoidable transitions are drawn with thicker lines

the overall constraint is $[B_1 \prec B_2] \vee [A_1 \prec A_2]$. Similarly, we perform the computation for transition A_2 from state 3 which generates an identical constraint. The final avoid formula for state 5 is thus the conjunction of $[B_1 \prec B_2] \vee [A_1 \prec A_2]$ with itself. In other words, it is $[B_1 \prec B_2] \vee [A_1 \prec A_2]$.

- For the error state 9, the two incoming transitions are executed in-order and cannot be avoided. The overall constraint is thus generated as a conjunction of the constraints of the predecessor states 7 and 8, and it is $[B_1 \prec B_2] \wedge [A_1 \prec A_2]$.

Note that since there is only one error state, the resulting overall formula is the avoid formula of that error state: $[B_1 \prec B_2] \wedge [A_1 \prec A_2]$.

Handling Boolean Functions The algorithm, as presented above, "hides" several representation and performance issues related to boolean functions. The clearest issue is that the algorithm returns a boolean function that represents a constraint formula. However, to actually place fences we require not the formula but rather its minimal satisfying assignments. We "offload" this task to standard SAT solving tools. As our experience shows, the SAT-solving stage is not a performance bottleneck.

A bigger issue is the fact *every* step of the algorithm requires an equivalence check of two boolean functions (the test $L(\sigma) \not\equiv \varphi$ in Line 11). This is NP-hard in general, and remains NP-hard even under the restriction that both functions are monotonic. With an explicit formulae representation those checks become very computationally expensive. However, if the functions are represented as Binary Decision Diagrams (BDDs) [4], then the equivalence check is, in our experience, also not a practical bottleneck.

Algorithm Correctness and Optimality To show our algorithm is correct, we need to demonstrate that (a) the fixed point computation terminates, and (b) once it terminates, enforcing the avoid formula $av(\sigma)$ indeed cuts the state σ. The algorithm we presented above is not only sound, but also maximally permissive. That is, the formula $av(\sigma)$ is the "weakest" (most permissive) formula that describes the constraints that must be enforced to make σ unreachable. The interested reader can find proofs of the above claims in [15].

2.2 Instantiation for a concrete model

In the previous section, we presented a general algorithm for inferring optimal constraints. When we instantiate this algorithm for a specific memory model, we first need to choose the type of constraints we *can* actually enforce. That means our constraints must satisfy at least the following basic property: For every transition t to which we assign $\chi(t) \neq \emptyset$ it is possible use fences to construct a program P' for which the transition system does not contain t.

The constraint language appropriate for RLX is the language of "ordering constraints" of the form $\psi = [l_1 \prec l_2]$ where l_1, l_2 are program labels. Intuitively, enforcing the constraint $[l_1 \prec l_2]$ means that P' cannot execute the instruction with label l_2 out of order with respect to the instruction at label l_1. We then define $\chi(t)$ for a transition $t = \sigma \longrightarrow \sigma_2$ with label l_t to be $\chi(t) = \{[l \prec l_t] \mid l <_{\sigma,p} l_t\}$. The relation $l <_{\sigma,p} l_t$ holds when:

1. The transition t was caused by executing the instruction at label l_t by some process p.
2. The execution buffer $E(p)$ of state σ contained an instance of the instruction at l before the instruction at l_t.

This is equivalent to saying the transition t represents the instruction at l_t being executed while bypassing l.

To show that the chosen constraint language is useful, we need to show a correspondence between the constraints and syntactic fences. More concretely, we need to show that:

- We know how to enforce any constraint formula produced by the algorithm by adding fences to P.
- Enforcement can be done efficiently. To see this is a non-trivial property, consider the constraint language consisting of a single constraint β, where the enforcement mechanism is "If $\psi = \beta$ add a fence between every two instructions in P". Clearly, we can enforce this constraint, and enforcing it would create a correct program, but this is not the desired outcome.

First, we can show that adding fences can never introduce new error states. It is clear that adding nop operations (with new labels) to a program has no effect on the program's behavior. So it is enough to show that the set of behaviors a program P with a fence at label l has is a subset of the possible behaviors of P with a nop at the same label. This can show this through a simple simulation argument.

Lemma 2.1. *Let P be a program with a* nop *instruction at label l, and P' the program P with the* nop *replaced by a* fence. *Then $\langle \sigma_0, \Sigma_P, T_P \rangle$ simulates $\langle \sigma_0, \Sigma_{P'}, T_{P'} \rangle$.*

This lemma is trivially extendable to replacing any number of nop instructions by fences. After we have established inserting fences cannot add new error states, the next thing we need to show is that we can in fact use syntactic fences to cut any transition t s.t. $\chi(t) \neq \emptyset$. This is established by the following lemma.

Lemma 2.2. *Let P be a program, $t = \sigma_1 \xrightarrow{l_t} \sigma_2$ a transition in $\langle \sigma_0, \Sigma_P, T_P \rangle$ and $v \in \chi(t)$, where $v = [l \prec l_t]$. Let P' be a modification of P s.t. a fence instruction is placed on every control path between l and l_t. Then there is no t' in $\langle \sigma_0, \Sigma_{P'}, T_{P'} \rangle$ that corresponds to t.*

A corollary of the lemma above is that we can enforce any constraint $v = [l_1 \prec l_2]$ (thus cutting any transition t s.t. $v \in \chi(t)$) by placing a fence on every control path between l_1 and l_2. Using these lemmas we can prove the main soundness theorem.

Theorem 2.3. *Let P be a program, S a specification, $\psi = \bigwedge \{av(\sigma) \mid \sigma \nvDash S\}$ and $\delta \models \psi$. Let P' be the program P modified s.t. for any $[l_1 \prec l_2] \in \delta$ a* fence *instruction is placed on every control path between l_1 and l_2. Then $\forall \sigma \in \Sigma_{P'}.\sigma \models S$.*

To show that the produced constraints are optimal, we can prove the following theorem.

Theorem 2.4. *Let P be a program, S a specification, $\psi = \bigwedge \{av(\sigma) \mid \sigma \nvDash S\}$, and P' the program P modified by inserting fences. If for every satisfying assignment $\delta \models \psi$ there exists $[l_1 \prec l_2] \in \delta$ s.t. there is no fence on* any *control path between l_1 and l_2, then there is some $\sigma \in \Sigma_{P'}$ s.t. $\sigma \not\models S$.*

2.3 Synthesizing Fences from Constraints

Theorem 2.3 shows that we can syntactically implement any solution to the constraint formula ψ produced by our algorithm. It shows that if for every constraint $[l_1 \prec l_2]$ that needs to be enforced fences are placed on all control-flow paths between l_1 and l_2 then the resulting program is safe. Unfortunately, while Theorem 2.4 shows a fence must be placed on *some* control path between l_1 and l_2, it does not require placing a fence on *all* of them. There are, in fact, several reasons a fence placement constructed by simply taking some minimal satisfying assignment δ of ψ and adding fences on all control-flow paths may be suboptimal:

- It is not even clear which optimality metric we should use. The number of (static) fences added to the program seems like a convenient choice, but may be misleading. Several fences placed before a loop may have a much smaller (dynamic) execution cost than a single fence placed inside the loop body.
- Theorem 2.4 shows we must add a fence on *some* control path of every constraint that belongs to a minimal satisfying assignment, as opposed to *all* control-flow paths. This is not a weakness of the theorem, as placing fences on all paths is in fact not always required. This may happen for two reasons. First, some control-flow paths may be infeasible, and putting fences on these paths is thus unnecessary. More subtly, it is possible that a given re-ordering of instructions is only harmful on some execution paths. Our chosen constraint language does not preserve enough information to make these distinctions. We could use an alternative constraint language to preserve it, but this would dramatically increase the size of Λ_P — the number of possible constraints would be exponential in the number of labels, as opposed to quadratic.
- Often, it is possible to satisfy several constraints with a single fence. Thus a judicious placement of fences is still required, even once a minimal assignment to the constraint formula is known. Moreover, different minimal assignments may lead to different placement tradeoffs.

We resolve the first issue by working with the natural partial order on fence placements: a set of added fences C is better then a set C' if $C' \subseteq C$. We then produce all minimal incomparable placements and leave the choice between them to the programmer. Choosing between incomparable (by containment) fence placements is a separate hard problem, which we leave to future work.

The second issue could be resolved by adopting a more precise "flow-sensitive" constraint language. This could be done by encoding in the constraints $\chi(t)$ of a transition $t = \sigma_1 \longrightarrow \sigma_2$ information about program paths that lead to t. Moreover, if we used a "context-sensitive" implementation mechanism instead of fences (for example "conditional fences" — fences that are only sometimes executed, depending on the current program state) we could use even finer constraints. For the input programs we used, none of these improvements were necessary. Therefore, we also defer examination of these alternatives to the future.

The third issue requires further examination. While there are in general many ways to implement a given constraint $v = [l_1 \prec l_2]$, for simple programs it usually sufficient (while clearly not optimal in general) to consider two options:

- Place a fence immediately after instruction l_1
- Place a fence immediately before instruction l_2 (if there are branch instructions pointing to l_2 they should point to the newly added fence).

This is complicated slightly by the fact that even in this case, there is interdependence between constraints. For example, consider a program with three statements with labels l_1, l_2, l_3 in sequence and the constraint formula $v_1 \wedge v_2$ where $v_1 = [l_1 \prec l_2], v_2 = [l_1 \prec l_3]$. Obtaining the (only) solution $\{v_1, v_2\}$ and then deciding to place a fence immediately before l_2 (to enforce v_1) and before l_3 (to enforce v_2) will result in a placement that contains two fences, instead of the expected single fence after l_1. We solve this by replacing the constraint formula ψ with a new formula ζ.

A fence may only be placed after an existing code label. Therefore, for each label we define a new variable v_l. We also define a function *prev* that returns for each label l the preceding (in the program code) label. We then produce ζ by replacing every variable $v = [l_1 \prec l_2]$ in ψ with the clause $v_{l_1} \vee v_{prev(l_2)}$. It is easy to show that every satisfying assignment to ζ still produces a sound fence placement. However, it alleviates the interdependence problem by "off-loading" it to a SAT solver. In the preceding example, the formula $v_1 \wedge v_2$ is transformed into $v_{l_1} \wedge (v_{l_1} \vee v_{l_2})$, with the minimal solution $\{v_{l_1}\}$ as desired.

Limitations The main drawback of the algorithm described in this section is the fact that it requires explicit enumeration of the program's state-space. While this is possible for some programs, many programs for which we want to infer fences do not allow such explicit enumeration because the state-space is not finite. This might happen due to a combination of several reasons. Three common reasons are:

1. We are interested in inferring a fence placement for an open system (e.g. library code), and not a single finite-state program.
2. The program for which we wish to infer fences utilizes a potentially unbounded number of heap locations.
3. The program is finite-state under SC but not finite-state under the desired relaxed memory model.

In case (1), the problem boils down to the fact we are not interested in placing fences in a single program. Rather, we want to place fences in the code of a library *implementation* such that it remains correct irrespective of the code *using* the library (the data structure *client*). A different way to phrase this is to say we want the *most general* client (which represents all possible clients) of the library to be correct. Unfortunately, the most general client itself is usually not a finite-state program. For example, consider a queue implementation that uses a linked list as the underlying data representation. A client that may add an unbounded number of elements to the queue will use unbounded memory, and the state-space for the client/queue combination is unbounded. In general, this is a hard problem that we do not try to completely solve. We attempt to reduce it using two methods: (a) Hand-picked clients that we believe are representative of the data structure's behavior. (b) Exhaustive enumeration of clients up to a specified bound on the number of operations. Neither of these two solutions produces a sound verification (or fence inference) procedure. However, in practice these methods allow us to infer optimal fences for realistic data structures. We have verified that the results are indeed optimal by manually comparing them to fence placements found in the literature.

In case (2), the problem is that due to use of an unbounded number of heap locations, the program is infinite-state even under the sequentially-consistent model. One way to deal with this problem is to "work around it" by applying the algorithm to slightly different programs, and dealing with the difference separately (e.g. using finite-state clients instead of the infinite-state most general client as in case (1)). Another is to use heap abstractions.

Regarding case (3), as Atig et al. have shown [2], this is in general a very hard problem. Given a finite-state (under SC) program P, deciding reachability for the same program under SPARC TSO or PSO has non-primitive recursive complexity. Further, under SPARC RMO, reachability for SC-finite-state programs becomes undecidable. One way to deal with problems of this kind is through the use of abstract interpretation — a technique explored in [17] and [15].

```
 1  void enqueue(queue_t *queue, value_t value)
 2  {
 3      node_t *node, *tail, *next;
 4      node = new_node();
 5      node->value = value;
 6      node->next = 0;
 7      fence("store-store");
 8      while (true) {
 9          tail = queue->tail;
10          fence("load-load");
11          next = tail->next;
12          fence("load-load");
13          if (tail == queue->tail)
14          if (next == 0) {
15          if (cas(&tail->next,
16              (unsigned) next, (unsigned) node))
17              break;
18          } else
19              cas(&queue->tail,
20                  (unsigned) tail, (unsigned) next);
21      }
22      fence("store-store");
23      cas(&queue->tail,
24          (unsigned) tail, (unsigned) node);
25  }
```

Figure 3: Enqueue operation of the Michael-Scott queue (from [5])

3 Experimental Evaluation

We have implemented our algorithm in a pair of tools called FENDER and BLENDER. FENDER is a direct implementation of the fence inference algorithm of section 2 for the RLX framework. In BLENDER we adapted the implementation to work with a wider range of memory models [17]. To give a flavor of the capabilities of these tools, we present the fence inference results for the Michael-Scott nonblocking queue [25]. This queue is one of few algorithms for which a correct fence placement (for RMO) has been previously published [5]. We refer to that placement of fences as the "reference placement". The reference placement uses 7 fences, 4 in `enqueue()`, and 3 in `dequeue()`. As [5] notes, all of the fences were found using small test-cases. Our hypothesis was that by running FENDER with a small number of test-cases, we can automatically infer the appropriate fences.

Under RMO (which is closest to the model used by [5]), a small set of clients produced 20 different sets of 4 constraints. Using the local fence placement method there are only 4 different ways to implement those sets using 3 fences: 1 fence in `enqueue()` and 2 in `dequeue()`. One of those placements was, as expected, a proper subset of the reference placement found in [5], and the others were similar.

Fig. 3 is copied verbatim from [5] and shows the `enqueue()` method for the algorithm (including 4 of the 7 fences placed using CheckFence). The reference placement contains 7 fences, while our tool inferred only 3 of these 7, which may seem, at first glance, insufficient. However, manual examination of the 4 missing fences confirms that they are in fact redundant in our model.

- The load-load fence on line 10 of Fig. 3 prevents the load on line 11 from being executed before the load on line 9 (note that the two loads are data dependent). To the best of our understanding SPARC RMO only allows control speculation, but not data speculation, which means this fence is in fact not necessary.
- The store-store fence on line 22 prevents the CAS on line 23 from being executed before the CAS on line 15. However, this may only happen if the CAS on line 23 is executed speculatively, since its execution is control-dependent on the *success* of the CAS in line 15. Under RMO operations that write to memory (and, in particular, CAS operations) may not be executed speculatively, so this fence

is never needed.

- The two load-load fences on line 12 and on line 57 of the `dequeue()` code given in [5] enforce the correct execution of a construct meant to solve a certain type of ABA problem that only occurs when immediate reuse of memory is allowed. However, under the assumption of automatic memory management, the statements in lines 13 and 58 are redundant (see [23]). Since the correct execution of these two statements is no longer important, FENDERcorrectly omits the two fences that "protect" them.

Under PSO, only sets of two constraints (implementable by a single store-store fence) in the `enqueue()` method were inferred. This is consistent with the fact that, under this model, loads are not reordered with each other so load-load constraints are unnecessary. Under TSO, no fences were inferred, again consistent with our expectations, and with the claim in [5] that under the x86 memory model (which resembles TSO), no fences should be necessary.

A wide range of our experimental results, as well as details on our methodology, appear in [16, 17].

4 Related Work

Several automated techniques to place memory fences in concurrent programs have been developed over the years. A large body of work dating back to the late 1980s relies on the concepts of *delay set analysis* of Shasha & Snir [27] for reasoning about relaxed memory models. This analysis enables one to find all potential conflicts (more or less equivalent to data races), and place fences accordingly. A fence inference scheme based on delay set analysis was successfully implemented in the "Pensieve" Java compiler [19, 12, 29], which can effectively process large amounts of code. However, a violation of SC does not necessarily cause a violation of any high-level properties. Thus those algorithms are often needlessly conservative. Unlike this previous work, the approach outlined in this paper, uses a high-level specification and allows a trade-off between performance and optimality of the solution.

Another possible approach to fence inference is to use a verification tool combined with syntactic exploration. There exist several techniques for program testing and verification under relaxed memory models, and tools have been developed that implement these techniques (cf. [5, 6, 8, 7, 20].) To utilize a verification tool (e.g. CheckFence [5]) for inference, the programmer may use an iterative process. She starts with an initial fence placement and if the placement is incorrect, she has to examine the (non-trivial) counterexample from the verification tool, understand the cause of error and attempt to fix it by placing a memory fence at some program location. It is also possible to use the tool by starting with a very conservative placement and choose fences to remove until a counterexample is encountered. This process, while simple, may easily lead to a "local minimum" and an inefficient placement. In [21], Linden & Wolper automate this approach, using the technique described in [20] as the underlying verification tool. However, their tool still suffers from the same problem - it does not necessarily provide a globally optimal solution.

5 Conclusion

We presented a novel fence inference algorithm and demonstrated its practical effectiveness by evaluating it on various challenging state-of-the-art concurrent algorithms. The work presented here is a small sample from our wider work on *synthesis of synchronization* in concurrent programs (e.g., [30, 32, 31, 33, 22]). In the future, we plan to extend our techniques to handle infinite-state programs (e.g., heap-manipulating programs) running on relaxed memory models.

References

[1] Adve, S. V., and Gharachorloo, K. Shared memory consistency models: A tutorial. *IEEE Computer 29* (1995), 66–76.

[2] Atig, M. F., Bouajjani, A., Burckhardt, S., and Musuvathi, M. On the verification problem for weak memory models. In *POPL* (2010), pp. 7–18.

[3] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. Cilk: an efficient multithreaded runtime system. In *PPOPP '95.*

[4] Bryant, R. E. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv. 24*, 3 (1992), 293–318.

[5] Burckhardt, S., Alur, R., and Martin, M. M. K. Checkfence: checking consistency of concurrent data types on relaxed memory models. In *PLDI* (2007), pp. 12–21.

[6] Burckhardt, S., and Musuvathi, M. Effective program verification for relaxed memory models. In *CAV* (2008), pp. 107–120.

[7] Burnim, J., Sen, K., and Stergiou, C. Testing concurrent programs on relaxed memory models. In *ISSTA '11*, pp. 122–132.

[8] Burnim, J., Sen, K., and Stergiou, C. Sound and complete monitoring of sequential consistency for relaxed memory models. In *TACAS'11* (2011), pp. 11–25.

[9] Chase, D., and Lev, Y. Dynamic circular work-stealing deque. In *SPAA* (2005), pp. 21–28.

[10] Clarke, E. M., Grumberg, O., and Peled, D. *Model Checking.* The MIT Press, 1999.

[11] Cousot, P., and Cousot, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *POPL* (1977), pp. 238–252.

[12] Fang, X., Lee, J., and Midkiff, S. P. Automatic fence insertion for shared memory multiprocessing. In *ICS* (2003), pp. 285–294.

[13] Gharachorloo, K., Gupta, A., and Hennessy, J. Performance evaluation of memory consistency models for shared-memory multiprocessors. In *ASPLOS'91.*

[14] Herlihy, M., and Shavit, N. *The Art of Multiprocessor Programming.* Morgan Kauffman, Feb. 2008.

[15] Kuperstein, M. Preserving correctness under relaxed memory models. Master's thesis, Technion, 2012.

[16] Kuperstein, M., Vechev, M., and Yahav, E. Automatic inference of memory fences. In *FMCAD* (2010), pp. 111–119.

[17] Kuperstein, M., Vechev, M., and Yahav, E. Partial-coherence abstractions for relaxed memory models. In *PLDI '11* (2011), pp. 187–198.

[18] Lamport, L. How to make a multiprocessor computer that correctly executes multiprocess program. *IEEE Trans. Comput. 28*, 9 (1979), 690–691.

[19] Lee, J., and Padua, D. A. Hiding relaxed memory consistency with a compiler. *IEEE Trans. Comput. 50*, 8 (2001), 824–833.

[20] Linden, A., and Wolper, P. An automata-based symbolic approach for verifying programs on relaxed memory models. In *SPIN* (2010), pp. 212–226.

[21] Linden, A., and Wolper, P. A verification-based approach to memory fence insertion in relaxed memory systems. In *SPIN* (2011), pp. 144–160.

[22] Liu, F., Nedev, N., Prisadnikov, N., Vechev, M., and Yahav, E. Dynamic synthesis for relaxed memory models. PLDI '12, to appear.

[23] Michael, M. M. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *PODC* (2002), pp. 21–30.

[24] Michael, M. M., and Scott, M. L. Correction of a memory management method for lock-free data structures. Tech. rep., 1995.

[25] Michael, M. M., and Scott, M. L. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC* (1996), pp. 267–275.

[26] Park, S., and Dill, D. L. An executable specification and verifier for relaxed memory order. *IEEE Transactions on Computers 48* (1999).

[27] Shasha, D., and Snir, M. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst. 10*, 2 (1988), 282–312.

[28] SPARC International Inc. *The SPARC architecture manual (version 9).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[29] Sura, Z., Wong, C.-L., Fang, X., Lee, J., Midkiff, S. P., and Padua, D. A. Automatic implementation of programming language consistency models. *LNCS 2481* (2005), 172.

[30] Vechev, M., and Yahav, E. Deriving linearizable fine-grained concurrent objects. In *PLDI* (2008), pp. 125–135.

[31] Vechev, M., Yahav, E., Bacon, D. F., and Rinetzky, N. CGCExplorer: a semi-automated search procedure for provably correct concurrent collectors. In *PLDI* (2007), pp. 456–467.

[32] Vechev, M., Yahav, E., and Yorsh, G. Inferring synchronization under limited observability. In *TACAS* (2009), pp. 139–154.

[33] Vechev, M., Yahav, E., and Yorsh, G. Abstraction-guided synthesis of synchronization. In *POPL '10* (2010).

[34] Yang, Y., Gopalakrishnan, G., and Lindstrom, G. UMM: an operational memory model specification framework with integrated model checking capability. *Concurr. Comput. : Pract. Exper. 17*, 5-6 (2005), 465–487.