A Framework for Automatic Synthesis of Fault-Tolerance[1]

Ali Ebnenasir   Sandeep S. Kulkarni
Software Engineering and Network Systems Laboratory
Department of Computer Science and Engineering
Michigan State University
East Lansing MI 48824 USA

Abstract

In this paper, we present a framework for adding fault-tolerance to existing fault-intolerant distributed programs. The input to our framework is an abstract structure of the fault-intolerant program, its specification, and a class of faults that perturbs the program. The output of our framework is the abstract structure of the fault-tolerant program. Our framework also enables one to add new heuristics for adding fault-tolerance. Further, it is possible to change the internal representation of different entities involved in synthesis while reusing the rest of the framework.

We have used this framework for automated synthesis of several fault-tolerant programs including token ring, Byzantine agreement, and agreement in the presence of Byzantine and failstop faults. These examples illustrate that the framework can be used for synthesizing programs that tolerate different types of faults (process restarts, Byzantine and failstop) and programs that are subject to multiple faults (Byzantine and failstop) simultaneously. We also note that our framework has been used for pedagogical purposes.

Keywords : Fault-tolerance, Automatic addition of fault-tolerance, Formal methods, Program synthesis, Distributed programs

1 Introduction

In the initial design of a fault-tolerant distributed program, it is often difficult to identify all the faults that may perturb the program. Hence, when new faults that affect an existing system are

[1]Email: ebnenasi@cse.msu.edu, sandeep@cse.msu.edu
Web: `http://www.cse.msu.edu/~{ebnenasi,sandeep}`
Tel: +1-517-355-2387, Fax: +1-517-432-1061
This work was partially sponsored by NSF CAREER CCR-0092724, DARPA Grant OSURS01-C-1901, ONR Grant N00014-01-1-0744, NSF grant EIA-0130724, and a grant from Michigan State University.

identified, it becomes necessary to upgrade the system to deal with those new faults. Moreover, during such addition of fault-tolerance, it is typically necessary to reuse the existing system as much as possible. Specifically, when the new fault *does not* occur, we expect the system to behave in the same way as it behaved before the upgrade.

It is desirable to use an automated synthesis algorithm while adding fault-tolerance to a distributed program as verifying it after the fact can be expensive. Since an automated synthesis algorithm ensures that the synthesized program is correct by construction there is no need for its proof of correctness. To automatically synthesize a fault-tolerant program, we can begin either with its formal specification, or with (the transitions of) an existing fault-intolerant program. In the context where we need to upgrade an existing program, it is desirable to follow the latter approach and reuse the existing program.

One of the difficulties in automating the addition of fault-tolerance to distributed programs is the complexity of such addition. In [1, 2], the authors have shown that, in general, the addition of fault-tolerance to distributed programs is NP-hard. To deal with this complexity and to enable the synthesis of programs that have large state space, heuristic-based approaches are proposed in [3–5]. These heuristic-based approaches reduce the complexity of the synthesis by forfeiting the completeness of fault-tolerance addition. In other words, if heuristics are applicable then a heuristic-based algorithm will generate a fault-tolerant program efficiently. However, if the heuristics are not applicable then the algorithm will declare failure even though it is possible to add fault-tolerance to the given fault-intolerant program.

The development and the implementation of heuristics is complicated by the fact that, for a given heuristic, we need to determine how that heuristic reduces the complexity of adding fault-tolerance. Furthermore, we need to identify if a heuristic is so restrictive that its use will cause the synthesis algorithm to declare failure very often. Also, in order to provide maximum efficiency, there exist situations where we need to apply heuristics in a specific order. Moreover, the developers of a fault-tolerant program may have additional insights about the order in which heuristics should be applied. Thus, we have to provide the possibility of changing the order of available heuristics (respectively, adding new heuristics) for the developers of fault-tolerance.

Goals. To address the above-mentioned issues, we develop a synthesis framework that has the following properties:

1. *Ability to add fault-tolerance to existing fault-intolerant programs.* One group of users who use our framework are the *developers of fault-tolerant programs*. For this group of users, the synthesis framework should provide mechanisms for the addition of fault-tolerance. Thus, at different stages in the synthesis of a fault-tolerant program, these developers should be able to interact with the framework in order to apply different heuristics (in their desired order) depending on the program being synthesized. Also, these developers should be able to query the framework to identify the intermediate versions of the fault-tolerant program so that they

can determine the heuristic that should be applied next.

2. *Ability to add new heuristics.* Another group of users are the *developers of heuristics* who need to evaluate the applicability of their new heuristics in reducing the complexity of fault-tolerance addition. In order to increase the efficiency of the synthesis, the developers of heuristics may need to *improve* the existing heuristics or *add* new heuristics for different tasks during the synthesis. Thus, our framework should allow improvements or additions of heuristics with a low overhead. In other words, the framework should be *extensible.*
3. *Ability to change internal representations.* The internal representation of entities such as programs and faults affects the efficiency of the synthesis of fault-tolerant programs. However, one cannot determine the ideal internal representation of these entities as each representation has its own advantages and disadvantages. Also, depending on the user requirements at run-time, the framework should switch between different internal representations of a particular entity. Hence, we should be able to modify the way these entities are represented with a low overhead.

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

- We present a framework for adding fault-tolerance to existing distributed programs. This framework allows the users to automatically (respectively, interactively) add fault-tolerance.
- We show that our framework permits one to add new heuristics for adding fault-tolerance. Towards this end, we describe the addition of several heuristics (based on the algorithms proposed in [3, 4]) for different steps involved in adding fault-tolerance.
- We show how one can easily change the internal representation of different entities in the framework. Towards this end, we utilize some of the design patterns in [6] effectively.
- We provide the option of obtaining an intermediate version of the synthesized program in Promela [7]; this option is especially useful if the heuristic being used fails, and it becomes necessary to analyze the intermediate version of the synthesized program to identify if another heuristic could be used or how a new heuristic can be developed.
- We note that our framework provides a suitable platform for teaching some basic concepts of distributed (e.g., distribution issues, non-determinism, etc) and fault-tolerant (e.g., faults, fault-tolerance, etc) systems. Hence, our framework is used for pedagogical purposes as well.

We have used the synthesis framework to develop a token ring protocol that tolerates restart of processes. We have also synthesized a fault-tolerant agreement protocol that tolerates Byzantine faults, and a fault-tolerant agreement protocol that tolerates both Byzantine faults and fail-stop faults. These examples illustrate the potential of our framework in dealing with the cases where programs are subject to multiple types of faults simultaneously.

Organization of the paper. In Section 2, we present preliminary concepts. Then, in Section 3, we illustrate how the developers of fault-tolerance can synthesize fault-tolerant programs using

our framework. In Section 4, we show how one can integrate new heuristics into our framework. In Section 5, we present the way in which one can change the internal representation of entities involved in the framework. In Section 6, we present an example fault-tolerant program synthesized using our framework. We discuss some issues related to our framework in Section 7. Finally, we make concluding remarks and discuss future work in Section 8.

2 Preliminaries

In this section, we present the theoretical background on which our synthesis framework is based. We present basic concepts in Section 2.1 and then we recall the problem statement for fault-tolerance addition in Section 2.2. Finally, in Section 2.3, we present a non-deterministic algorithm for solving the addition problem. We have adapted this algorithm from [1].

2.1 Basic Concepts

In this section, we give the definitions of programs, problem specifications, state predicates, faults, and fault-tolerance. The programs are specified in terms of their state space and their transitions. The definition of specifications is adapted from Alpern and Schneider [8]. The definition of faults and fault-tolerance is adapted from Arora and Gouda [9] and Kulkarni [10]. The issues of modeling distributed programs is adapted from [1, 11].

Program and specification. A program p is a finite set of variables and a finite set of processes. Each variable is associated with a finite domain of values. A state of p is obtained by assigning each variable a value from its respective domain. The state space of p, S_p, is the set of all possible states of p. Since the domain size of each variable is finite, the state space of the program contains a finite number of states. A process, say j, in p is associated with a set of program variables, say r_j, that it can read and a set of variables, say w_j, that it can write. Also, process j consists of a set of transitions δ_j; each transition is of the form (s_0, s_1), where $s_0, s_1 \in S_p$. The set of transitions of p, δ_p, is the union of the transitions of its processes.

A sequence of states, $\langle s_0, s_1, ...\rangle$, is a computation of p iff (if and only if) the following two conditions are satisfied: (1) $\forall j : j > 0 : (s_{j-1}, s_j) \in \delta_p$, and (2) if $\langle s_0, s_1, ...\rangle$ is finite and terminates in state s_l then there does not exist state s such that $(s_l, s) \in \delta_p$.

For program p, its safety specification is a subset of $\{(s_0, s_1) : s_0, s_1 \in S_p\}$ that represents a set of bad transitions that p should not execute.

A state predicate, S, of p is any subset of S_p. A state predicate S is closed in program p iff for any transition (s_0, s_1) of p, if $s_0 \in S$ then $s_1 \in S$. The invariant of program p is a state predicate S from where specification is satisfied and S is closed in p. The projection of program p on state predicate S, denoted as $p|S$, consists of transitions of p that start in S and end in S.

Distribution model. We model distribution by identifying how read/write restrictions on a process affect its transitions. A process j cannot include transitions that write a variable x, where $x \notin w_j$. In other words, the write restrictions identify the set of transitions that a process j can execute. Given a single transition (s_0, s_1), it appears that all the variables must be read to execute that transition. For this reason, read restrictions require us to group transitions and ensure that the entire group is included or the entire group is excluded. As an example, consider a program consisting of two variables a and b, with domains $\{0, 1\}$. Suppose that we have a process that cannot read b. Now, observe that the transition from the state $\langle a = 0, b = 0 \rangle$ to $\langle a = 1, b = 0 \rangle$ can be included iff the transition from $\langle a = 0, b = 1 \rangle$ to $\langle a = 1, b = 1 \rangle$ is also included. If we were to include only one of these transitions, we would need to read both a and b. However, when these two transitions are grouped, the value of b is irrelevant and, hence, we do not need to read it.

Faults. The faults that a program is subject to are systematically represented by transitions. A fault f for a program p with state space S_p, is a subset of the set $\{(s_0, s_1) : s_0, s_1 \in S_p\}$. A sequence of states, $\sigma = \langle s_0, s_1, ...\rangle$, is a computation of p in the presence of f (denoted $p[]f$) iff the following three conditions are satisfied: (1) every transition $t \in \sigma$ is a fault or program transition; (2) if σ is finite and terminates in s_l then there exists no program transition originating at s_l, and (3) the number of fault occurrences in σ is finite.

We say that a state predicate T is an f-span (read as fault-span) of p from S iff the following two conditions are satisfied: (1) $S \Rightarrow T$ and (2) T is closed in $p[]f$. Observe that for all computations of p that start at states where S is true, T is a boundary in the state space of p up to which (but not beyond which) the state of p may be perturbed by the occurrence of the transitions in f.

Fault-tolerance. Given a program p, its invariant, S, its specification, *spec*, and a class of faults, f, we say p is masking f-tolerant to *spec* from S iff the following two conditions hold: (i) p satisfies *spec* from S; (ii) there exists a state predicate T such that T is an f-span of p from S, $p[]f$ satisfies *spec* from T, and every computation of $p[]f$ that starts from a state in T has a state in S.

2.2 Problem Statement for Addition of Fault-Tolerance

For a given class of faults f, the objective of the addition of fault-tolerance to an existing fault-intolerant program p is to ensure no new behaviors are added in the absence of f and to add the necessary fault-tolerance behaviors in the presence of f. Hence, if S' is the invariant of p' then S' should not include states that do not belong to S. Also, $p' \mid S'$ should not include a transition that does not belong to $p \mid S'$. Otherwise, p' will have new computations in the absence of faults. Hence the problem of fault-tolerance addition is defined as follows (from [1]):

The Addition Problem

Given p, S, $spec$ and f such that p satisfies $spec$ from S

Identify p' and S' such that

$S' \subseteq S$,

$p'|S' \subseteq p|S'$, and

p' is masking f-tolerant to $spec$ from S'. □

2.3 Non-deterministic Synthesis Algorithm for Distributed Programs

Kulkarni and Arora [1] show that the addition of masking fault-tolerance to distributed programs is NP-hard. They present a non-deterministic algorithm for the addition of fault-tolerance to distributed programs in polynomial time. We repeat this algorithm in Figure 1.

```
Add_ft(p, f : transitions, S : state predicate, spec : specification, g0, g1, ..., gmax : groups of transitions)
{
    ms := {s0 : ∃s1, s2, ...sn : (∀j : 0≤j<n : (sj, s(j+1)) ∈ f) ∧  (s(n−1), sn) violates spec };
    mt := {(s0, s1) : ((s1 ∈ ms) ∨ (s0, s1) violates spec) };

    Guess S', T', and p' := ⋃ (gi : gi is chosen to be included in the fault-tolerant program);
    Verify the following
        (F1) p'|S' ⊆ p|S';
        (F2) S' ⇒ T'; T' is closed in p'[]f;              // T' is a fault-span of p'.
        (F3) T' ∩ ms = {}; (p'|T') ∩ mt = {};             // Safety cannot be violated from states in T'.
        (F4) (∀s0 : s0 ∈ T' : (∃s1 :: (s0, s1) ∈ p'));   // T' does not have deadlocks.
        (F5) S' ≠ {}; S' ⊆ S; S' is closed in p';          // S' is an invariant of p'.
        (F6) p'|(T' − S') is acyclic;                     // p' cannot stay in (T' − S') forever.
}
```

Figure 1: The non-deterministic algorithm.

This algorithm first computes the set of states ms from where safety can be violated by the execution of fault transitions alone. Thus, the fault-tolerant program should not reach a state in ms. Then it computes the set of transitions mt that violate safety or reach a state in ms. It follows that a fault-tolerant program should not execute a transition in mt. Then, the Add_ft algorithm non-deterministically guesses the fault-tolerant program, p', its invariant, S' and its fault-span, T'. Finally, the algorithm verifies that the synthesized (guessed) fault-tolerant program satisfies the three conditions of the addition problem (cf. Section 2.2). This goal is achieved by verifying the six formulae $F1$-$F6$. Since the algorithm is non-deterministic, there is no specific order in the verification of $F1$-$F6$. A heuristic based algorithm modifies the given fault-intolerant program so that predicates $F1$-$F6$ are satisfied, where we define a heuristic as follows:

Definition. A *heuristic* is a strategy that determines a specific order for verifying or satisfying (a subset of) $F1$-$F6$ in order to satisfy one of the requirements of the addition problem.

3 Adding Fault-Tolerance To Distributed Programs

In this section, we first describe the input and the output of our framework. Then, we give an overview of framework *fractions* that participate in the *automatic synthesis* of fault-tolerant

programs. We implement a deterministic version of *Add_ft* algorithm (cf. Section 2.3) to synthesize a fault-tolerant program. (We have adapted this algorithm from [3].) Further, we illustrate how the users can interact with the framework in order to *semi-automatically* synthesize a fault-tolerant program from its fault-intolerant version.

The input/output of the framework. The input to our framework is the abstract structure of the fault-intolerant programs, represented by Dijkstra's guarded commands [12]. A guarded command (action) is of the form $g \to st$, where g is a state predicate and st is a statement that updates the program variables. The guarded command $g \to st$ includes all program transitions $\{(s_0, s_1) : g$ holds at s_0 and the atomic execution of st at s_0 takes the program to state $s_1\}$. The output of our framework is also the abstract structure of the fault-tolerant program, represented by guarded commands.

We note that the abstract structure used by our framework enables us to model real-world applications including those written in common programming languages such as C. Specifically, there exist some automated techniques (e.g., [13]) for extracting the abstract structure of programs written in common programming languages. Moreover, after the synthesis of a fault-tolerant program, there exists some automated techniques (e.g., [14–16]) that allow us to refine the abstract structure of the fault-tolerant program while preserving its correctness and fault-tolerance properties.

The faults are also modeled as a set of guarded commands that change the values of program variables. The invariant and the safety specification of the fault-intolerant program are represented as state predicates (Boolean expressions over program variables). In addition, the synthesis framework takes the initial states of the fault-intolerant program as its input. While these initial states are included in the invariant of the fault-intolerant program, we find that explicitly listing them assists in adding fault-tolerance. Finally, the output is also in the guarded command language.

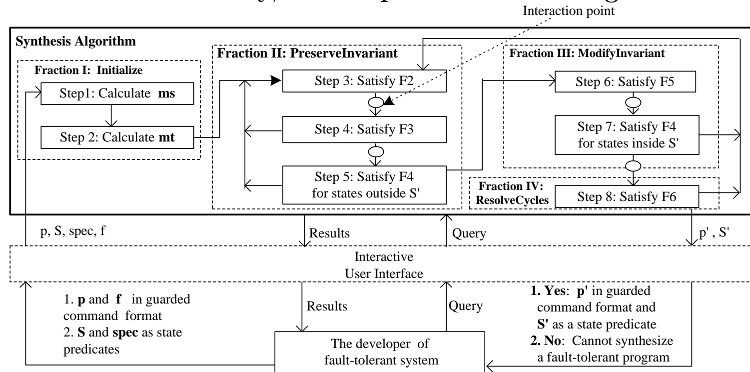

Figure 2: The framework deterministic execution mechanism.

Framework execution scenario. We now discuss the sample execution scenario for the case where fault-tolerance is added without any user interaction. In this scenario, the synthesis algorithm consists of four fractions: *Initialize, PreserveInvariant, ModifyInvariant*, and *ResolveCycles* (cf. Figure 2).

Before the execution of the synthesis algorithm, the framework uses initial states and program (respectively, fault) transitions to generate the state transition graph of the fault-intolerant program. Since this directed graph only includes those states of the state space that are *reachable* by program/fault transitions, we call it a *reachability graph* of the fault-intolerant program. (It also represents the fault-span of the fault-intolerant program.) After the expansion of the reachability graph, the framework executes every step of the synthesis algorithm on the reachability graph of the fault-intolerant program in order to derive a reachability graph of the fault-tolerant program.

In fraction (I) (cf. Figure 2), we calculate the sets of ms states and mt transitions (in the reachability graph). Then, we move to fraction (II) where we attempt to identify a valid fault-span T' that (i) is closed in $p'[]f$; (ii) does not include any ms state or safety violating transitions mt, and (iii) does not include any deadlock state outside the invariant. While executing in fraction (II), we leave the invariant S' unchanged. This is due to the fact that the addition problem requires that the invariant of the fault-tolerant program is a subset of the invariant of the fault-intolerant program. Thus, states inside the invariant of the fault-intolerant program are important; removing them prematurely can cause the automated synthesis to fail.

When we remove ms states (respectively, remove mt transitions) from T' in order to satisfy $F3$, the new fault-span will be a subset of initial T'. As a result, those transitions that start in the new fault-span and end in the part of T' that is not in the new fault-span violate the closure of the fault-span (i.e., $F2$). Hence, after satisfying $F3$, we may need to re-satisfy $F2$. A similar scenario can happen while resolving deadlock states (i.e., satisfying $F4$) outside the invariant. Hence, fraction (II) is an iterative procedure; i.e., after we establish $F3$ and $F4$, we have to re-ensure that $F2$ holds. The execution continues in fraction (II) until an iteration does not cause any changes or until the number of iterations exceeds a predetermined bound.

At the end of fraction (II), if the resulting program does not satisfy $F1$-$F6$, we modify the invariant S' in fraction (III) to ensure that the invariant S' is closed in the program p', i.e., $F5$ is satisfied. In fraction (III), we recalculate a valid invariant. In this fraction, the newly added transitions may violate the closure of the fault-span. Hence, when we exit fraction (III), the conditions $F2$-$F4$ may need to be re-satisfied. If $F2$-$F4$ are violated, we will jump to fraction (II) and attempt to re-satisfy $F2$-$F4$. Notice that in fraction (III), we satisfy $F4$ only for the invariant states; i.e., we ensure that there is no deadlock state inside the invariant whereas in fraction (II), we resolve deadlock states that are in the fault-span but outside the invariant.

If the values of p', S', and T' satisfy formulae $F2$-$F5$ at the end of fraction (III) then we will ensure that p' will not stay outside its invariant forever. Toward this end, we move into fraction (IV) where we remove non-progress cycles in $T'-S'$ (if any).

User interactions. Although the framework can automatically synthesize a fault-tolerant program without user intervention, there are some situations where (i) user intervention can help to speed up the synthesis of fault-tolerant programs, or (ii) a fully automatic approach fails. Hence,

our framework permits developers to semi-automatically supervise the synthesis procedure. In such *supervised synthesis*, fault-tolerance developers interact with the framework and apply their insights during the synthesis. In order to achieve this goal, we have devised some *interaction points* (cf. Figure 2) where the developers can stop the synthesis algorithm and query it.

At each interaction point, the users can make the following kinds of *queries*: (i) apply a specific heuristic for a particular task; (ii) apply some heuristics in a particular order; (iii) view the incoming program (respectively, fault) transitions to a particular state; (iv) view the outgoing program (respectively, fault) transitions from a particular state; (v) check the membership of a particular state (respectively, transition) to a specific set of states (respectively, transition); e.g., check the membership of a given state s in the set of ms states, and finally (vi) view the intermediate representation of the program that is being synthesized. Since the goal of the paper is to focus on the technical details of the framework and its application in adding fault-tolerance, for reasons of space, we omit the details about the user interface of the framework and about how users interact with it to apply different heuristics. We refer the reader the tutorial about using this framework at [17].

While we expect that the queries included in this version will be sufficient for a large class of programs, we also provide an alternative for the case where these queries are insufficient. Specifically, in this case, the users of our framework can obtain the corresponding intermediate program in Promela modeling language [7]; this program can then be checked by the SPIN model checker to determine the exact scenario where the intermediate version does not provide the required fault-tolerance. We note that while the code that interprets the counterexamples given by SPIN is not currently implemented, it will be available in the next version of the framework.

4 Integrating New Heuristics

In this section, we address the problem of adding new heuristics into our framework (i.e., the second goal mentioned in the Introduction). Specifically, we show how one can integrate a new heuristic into our framework so that the added heuristic will be available to the developers of fault-tolerance during synthesis. Since a new heuristic will be integrated into a new class or into a method of an existing class, the problem of *adding new heuristics to the framework* reduces to the problem of adding new classes (respectively, methods) to the framework.

We have used the ability to add heuristics for adding several heuristics from [2–4]. Of these heuristics, we now present the integration of the three heuristics that we added for resolving deadlocks and discuss our experience in adding them.

First heuristic. Kulkarni, Arora, and Chippada [3] present a heuristic for deadlock resolution that includes two passes. In the first pass, their heuristic tries to add single-step recovery transitions from a given deadlock state, s_d, to the invariant. Due to distribution issues, when their heuristic

adds a recovery transition, t_{rec} , it has to add the group, g_{rec} , of transitions that is associated with t_{rec}. Moreover, the addition of g_{rec} is not allowed if there exists a transition $(s_0, s_1) \in g_{rec}$ such that (i) $(s_0, s_1) \in mt$; (ii) $(s_0, s_1 \in S) \wedge (s_0, s_1) \notin p$; (iii) $(s_0 \in T') \wedge (s_1 \notin T')$, and (iv) $(s_0 \in S) \wedge (s_1 \notin S)$. If adding recovery to s_d is not possible, and s_d is directly reachable from the invariant by fault transitions then their heuristic does nothing in the first pass. Otherwise, their heuristic makes s_d unreachable.

In the second pass, if there still exists a deadlock state s_d that is directly reachable from the invariant by fault transitions then their heuristic will make s_d unreachable by removing the corresponding invariant state. At the end of deadlock resolution, if the invariant is empty then they declare that their heuristic could not synthesize a fault-tolerant program. We have integrated their heuristic into the framework using the DeadlockResolver1 class (cf. Figure 3) that inherits from the DeadlockResolver class.

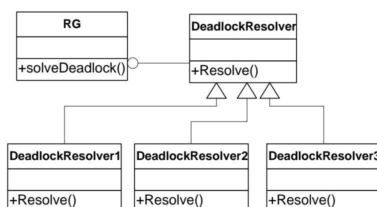

Figure 3: Integrating the deadlock resolution heuristics using Strategy pattern.

The class diagram of Figure 3 shows the Strategy design pattern [6] that we have applied to the method Resolve of the class DeadlockResolver. The class RG models the reachability graph of the program being synthesized whose methods are the steps of the synthesis algorithm. One of the methods of the RG class is the solveDeadlock method that we use to resolve deadlock states during the synthesis.

Second heuristic. The first heuristic only adds single-step recovery to deadlock states. As a result, it fails in cases where single-step recovery is not possible. For example, the first heuristic fails in the case that recovery from a deadlock state, say s'_d, is possible via another deadlock state, say s_d, from which we have already added a recovery transition to the invariant. Hence, we develop a new heuristic for adding *multi-step* recovery to deadlock states for the cases where single-step recovery to the invariant is not possible.

Our new heuristic also consists of two passes. In the first pass, we conduct a fixpoint computation that searches through the deadlock states outside the invariant (i.e., in the fault-span). In the first iteration of the fixpoint computation, we find all *deadlock* states from where single-step recovery to the invariant is possible. In the second iteration, we find all *deadlock* states from where single-step recovery is possible to recovery states explored in the first iteration. Continuing thus, we reach an iteration of the fixpoint computation where either no more deadlock states exist or no more recovery is possible. In the latter case, we choose to deal with the remaining deadlock states in the second pass. In the former case, at the end of the fixpoint computation, we will have a set of

states, *RecoveryStates*, from where there exists a multi-step recovery path to the invariant. (Notice that adding a recovery transition in a distributed program requires the satisfaction of the grouping requirements described in the first heuristic.)

In the second pass, we try to remove s_d if s_d is directly reachable by fault transitions from the invariant and no recovery can be added to s_d. If the removal of s_d requires the removal of one or more invariant states then we remove those invariant states. During deadlock resolution, if the invariant becomes empty then we declare that the synthesis framework failed to synthesize a fault-tolerant program.

In order to integrate this new heuristic into our framework, we extended a new class DeadlockResolver2 (cf. Figure 3) from the abstract class DeadlockResolver and then implemented our new heuristic in its Resolve method.

Third heuristic. The strategy of the third heuristic is similar to that in the second heuristic, except that the domain of the fixpoint computation includes all the states $s \in (T' - S')$. In other words, the third heuristic is more general than the second heuristic. (Likewise, the second heuristic is more general than the first heuristic.) We have also used this heuristic for enhancing the fault-tolerance of nonmasking programs – where the program only guarantees recovery to the invariant in the presence of faults and not necessarily a safe recovery – to masking fault-tolerance [4]. The integration of the third heuristic was fairly simple. We integrated the third heuristic into a class DeadlockResolver3 (cf. Figure 3) extended from the abstract class DeadlockResolver.

The application of heuristics. The first heuristic suffices for the synthesis of the fault-tolerant program presented in Section 6. However, in the synthesis of a version of the Byzantine agreement program containing four non-general processes, since the first heuristic failed, we applied the second heuristic.

The developers of fault-tolerance have the option to select one of the above heuristics during synthesis. Despite the generality of the third heuristic, it is not as efficient as the first two heuristics. Therefore, given a particular problem, the developers can either use their insight to choose the appropriate heuristic or they can rely on the framework to make that choice. The former choice provides more efficiency whereas the latter choice allows more automation.

5 Changing The Internal Representations

As we mentioned in the Introduction, it is difficult to determine a priori the internal representation that one should use for different entities, namely Program, Fault, Specification, and Invariant, involved in the synthesis of fault-tolerant programs. We model each entity of the framework as a class in the object-oriented design of the framework. Thus, it is necessary to provide the ability to modify the internal representation of these entities while reusing the remaining parts of the framework. In fact, there are situations where one needs to use one internal representation while

executing in one fraction of the framework, and a different internal representation for the same entity while executing in another fraction of the framework.

In this section, we argue that our framework enables such a change of internal representation for entities involved in our framework. Towards this end, we discuss our experience in changing the internal representation of entities, SafetySpecification and Invariant, in our framework. We find that the ability to modify the representation of entities in this fashion is especially useful for improving the efficiency of the framework as well as in simplifying the tasks involved in responding to the queries that the users make at interaction points. We discuss these applications next.

Improving the efficiency. The initial implementation of the SafetySpecification class consisted of a linked list whose elements would each represent a set of safety-violating transitions. The SafetySpecification class includes a method violates by which we can check whether a given transition t violates the safety of the specification or not. In order to check the safety of t, we needed to traverse the linked list structure of SafetySpecification. The traversal of the SafetySpecification structure was very time-consuming, especially when the size of the state space would become large. Since during the synthesis of a fault-tolerant program we need to invoke the method violates in many places, the efficiency of this method significantly degraded the overall efficiency of the synthesis. Hence, we changed the data structure used for the internal representation of the SafetySpecification class.

We replaced the linked list structure of the SafetySpecification class with a dummy data structure. Now, for a given transition t, we first take the source and destination states of t (specified as s_t and d_t). In order to check the safeness of t, we then substitute the values of the program variables at s_t and d_t into the state predicates that represent the safety *specification* (for an example, cf. Section 6). If the specification predicate holds for s_t and d_t then t violates safety. (Note that we represent safety specification as a set of transitions that the program is not allowed to execute.) We have applied the same approach for the Invariant class. Therefore, instead of traversing a huge linked list data structure, we check only a predicate in order to find out the safeness of a transition or the membership of a state to the invariant.

Reasoning about a query. As we discussed in this section, we have two different implementations for the SafetySpecification class based on *linked list* and *dummy* data structures. The latter data structure helps to improve the efficiency of the synthesis when we need to automatically synthesize a fault-tolerant program without user intervention. On the other hand, when users interact with our framework, they may need to know why a particular transition violates the safety of the specification. To answer this query, the framework uses the information stored in the linked list data structure in order to provide the required reasoning for the users. Thus, in such situations, the framework switches the implementation of the SafetySpecification class from a dummy to a linked list data structure to provide the required reasoning for the developers of fault-tolerance.

6 Example

In this section, we present a very simple example in order to demonstrate the way that the developers of fault-tolerance can synthesize fault-tolerant programs. The fault-intolerant program is a token ring program with 4 processes that is subject to state-corruption faults and the fault-tolerant program tolerates up to three corrupted processes. For reasons of space, we refer the reader to [17, 18] for other examples where this framework is used. We note that in these examples, the framework has been used to add fault-tolerance to different types of faults including Byzantine faults, failstop faults, and input corruption.

In Section 6.1, we show how the developers of fault-tolerance can specify the input fault-intolerant program. In Section 6.2, we present the output that the framework automatically generates.

6.1 Input

The user should specify the input fault-intolerant program, its variables, its invariant, its specification, and the faults in a text file. In this section, we describe the structure and the syntax of the input file.

Token ring program. The fault-intolerant program consists of four processes p_0, p_1, p_2, and p_3 arranged in a ring. Each process p_i, $0 \leq i \leq 3$, has a variable x_i with the domain $\{-1, 0, 1\}$. We say that process p_i, $1 \leq i \leq 3$, has the token if and only if $(x_i \neq x_{i-1})$ and fault transitions have not corrupted p_i and p_{i-1}. And, p_0 has the token if $(x_3 = x_0)$ and fault transitions have not corrupted p_0 and p_3. Process p_i, $1 \leq i \leq 3$, copies x_{i-1} to x_i if the value of x_i is different than x_{i-1}. This action passes the token to the next process. Also, if $(x_0 = x_3)$ then process p_0 copies the value of $(x_3 \oplus 1)$ to x_0, where $\oplus$ is addition in modulo 2. Now, if we initialize every x_i, $0 \leq i \leq 3$, with 0 then process p_0 has the token and the token circulates along the ring.

We specify the program variables in the *var* section as follows (cf. Lines 2-6 below).

```
1  program TokenRing
2   var
3  int x0, domain -1 .. 1;
4  int x1, domain -1 .. 1;
5  int x2, domain -1 .. 1;
6  int x3, domain -1 .. 1;
```

After the variable declaration, we write the actions of the processes. For example, the action of p_0 is as follows.

```
1   process  P0
2   begin
3     (x0 == x3) ->  x0 = ((x3+1)%2);
4   read  x0, x3;
5   write x0;
6   end
```

We specify the body of other processes similar to the specification of p_0.

Each process p_i, $1 \leq i \leq 3$, is only allowed to read x_{i-1} and x_i, and allowed to write x_i. Process p_0 is allowed to read x_3 and x_0, and write x_0. We specify the read/write restrictions of a process by *read* and *write* keywords inside the body of the process (cf. lines 4 and 5 in the body of p_0).

State-corruption faults. The faults may corrupt at most three processes. The faults are detectable in that a process that is corrupted can detect if it is in a corrupted state. Hence, we model the fault at process p_i by setting $x_i = -1$. Thus, one of the fault actions that corrupts x_0 is represented as follows:

```
1  fault TokenCorruption
2  begin
3     ( ((x0!=-1)&&(x1!=-1)) || ((x0!=-1)&&(x2!=-1)) ||
4       ((x0!=-1)&&(x3!=-1)) || ((x1!=-1)&&(x2!=-1)) ||
5       ((x1!=-1)&&(x3!=-1)) || ((x2!=-1)&&(x3!=-1)) ) -> x0 = -1;
6  end
```

Note that there exist no read/write restrictions for the fault transitions because we assume that fault transitions can read and write arbitrary program variables.

Safety specification. Based on the problem specification, the fault-tolerant program is not allowed to take a transition where a non-corrupted process copies a corrupted value from its neighbor. Also, the program should not reach a state where there exists more than one token. In the input file, we represent the specification as follows.

```
1 specification
2 noDestination
3 relation
4 ((x0s!=-1) || (x1s!=-1) || (x2s!=-1) || (x3s!=-1))  &&
5 ( ((x1s!=-1)&&(x1d==-1)) || ((x2s!=-1)&&(x2d==-1)) ||
6   ((x3s!=-1)&&(x3d==-1)) || ((x3s==-1)&&(x0s!=x0d)) )
```

The *specification* section is divided into two parts: *destination*, and *relation* parts. Intuitively, in the *destination* part (cf. line 2), we write a state predicate that identifies a set of states $s_{destination}$, where if a transition t reaches $s_{destination}$ then t violates safety. In the *relation* part (cf. line 3), we specify a condition that identifies a set of safety-violating transitions by a relation between their source and destination states. Note, that we have added a suffix "s" (respectively, suffix "d") to the variable names in the relation section that stands for *source* (respectively, *destination*). Since the relation condition specifies a set of transitions t_{spec} using their source and destination states, we need to distinguish between the value of a specific variable xi in the source state of t_{spec} (i.e., xis means the value of xi in the source state of t_{spec}) and in the destination state of t_{spec} (i.e., xid means the value of xi in the destination state of t_{spec}).

In the case that the program specification does not stipulate any destination condition on safety-violating transitions, we leave the destination section empty with the keyword *noDestination* (cf. line 2). (We use similar keyword *noRelation* for the cases where we do not have any relation conditions in the specification, respectively.)

Invariant. The invariant of the program consists of the states where no process is corrupted and there exists only one token in the ring. We represent the invariant of the program using the *invariant* keyword followed by a state predicate (Boolean function over program variables) that identifies the invariant states.

Initial states. Finally, in the input file, we identify one or more initial states for the expansion of the reachability graph using *init* and *state* keywords. The initial states are invariant states.

```
1 init
2 state x0 = 0; x1 = 0; x2 = 0; x3 = 0;
```

6.2 Output

In this section, we present the output of the synthesis framework for the token ring program. In particular, we present the guarded commands generated as the actions of process p_0. Since p_0 is a distinguished process, we present the structure of other processes in a parameterized format.

The actions of process p_0 are as follows:

```
1 (x0==-1) && (x3==1) -> x0 := 0;
2 (x0==1) && (x3==1)  -> x0 := 0;
3 (x0==0) && (x3==0)  -> x0 := 1;
4 (x0==-1) && (x3==0) -> x0 := 1;
```

The above actions mean that p_0 can copy the value of $(x_3 \oplus 1)$ to x_0 as long as $x_3 \neq -1$. Then, the framework generates the following actions for a typical process p_i $(1 \leq i \leq 3)$.

```
1 (xi==1) && (x(i-1)==0)   -> xi := 0;
2 (xi==-1) && (x(i-1)==0)  -> xi := 0;
3 (xi==0) && (x(i-1)==1)   -> xi := 1;
4 (xi==-1) && (x(i-1)==1)  -> xi := 1;
```

The above actions stipulate that each process p_i can copy the value of x_{i-1} to x_i if $(x_{i-1} \neq -1)$ holds.

The token ring program that we have *automatically* synthesized using our framework is the same as the program that was *manually* designed in [19]. To synthesize the fault-tolerant program in this context, any of the heuristics from Section 4 suffices. The synthesized token ring program prevents each process from copying a corrupted value from its predecessor. In other words, if faults corrupt the state of one or more processes then the fault-tolerant program prevents the propagation of state corruption. Also, since faults can corrupt at most three processes, at least one process always

remains uncorrupted. Thus, the uncorrupted process propagates its value to the entire ring, and the distributed program recovers to its invariant.

We note that our synthesis framework ensures that $F1$-$F6$ are satisfied before outputting the fault-tolerant program. Thus, the synthesized program is correct by construction. In this example, we found the ability to obtain the fault-tolerant program in Promela very useful; we further used the SPIN model checker for the verification of the Promela program.

More examples. We have also synthesized two agreement programs consisting of four non-general processes and a general process. First, we synthesized an agreement program that is masking fault-tolerant to Byzantine faults. Then, we synthesized another agreement program that simultaneously is subject to Byzantine and fail-stop faults. We refer the interested reader to [17,18] where we have some other examples and the code of the framework.

7 Discussion

In this section, we discuss some theoretical, practical, and pedagogical issues related to the development of our framework. We also discuss issues related to the applicability of our framework in the design of real-world fault-tolerant applications.

Complexity. In principal, the synthesis problem is harder than model checking in the sense that the complexity of model checking is polynomial in the size of the model [20] whereas the synthesis of distributed fault-tolerant programs is NP-complete [1,2]. The complexity of synthesis, however, can be reduced to polynomial time if we use appropriate heuristics and the heuristics are applicable. Thus, one of the important problems in synthesis is to identify heuristics that will keep the complexity of synthesis manageable. The framework proposed in this paper is especially useful for testing and developing such heuristics.

Besides the complexity of synthesis, we would like to note the finding in [21] that one of the important obstacles for automated synthesis is the efficiency/practicality of the synthesized program. We expect that our approach for reusing fault-intolerant programs will assist in overcoming this obstacle. Specifically, in such addition of fault-tolerance, there is a potential to preserve the efficiency of the fault-intolerant program, and in the programs synthesized so far, we have realized this potential, i.e., the framework could synthesize programs that were as efficient as manually designed programs.

Scalability. In practice, our framework provides the required platform for the implementation of our synthesis algorithms where the developers of fault-tolerance can benefit from the framework by synthesizing fault-tolerant programs. In this paper, we argued that using our framework, we synthesize fault-tolerant programs that tolerate different types of faults and are simultaneously subject to multiple faults. The largest state space among the programs that we have synthesized belongs to an agreement program that is simultaneously perturbed by Byzantine and fail-stop faults

(1.3 million states) [17, 18].

Although the state space of 1.3 million is much smaller than the state space of many practical applications, we argue that the framework has the potential in adding fault-tolerance to real-world applications. Towards this end, we discuss the following three points:

First, we argue that model checkers were also faced with similar problems with which our framework faces regarding the state space explosion. Researchers were using early versions of model checkers for checking small protocols and verifying the correctness of operating system kernels [22,23] despite a state space limit of about 500,000 states on an average workstation (in the early 90s) [22]. The state space handled by our framework is comparable to that reported by early model checkers. We expect that by incorporating the recent optimizations developed for model checking, it will be possible to increase the state space for which fault-tolerance can be added using our framework.

Second, we have not currently included these optimizing techniques in the current version of the synthesis framework as the goal of the framework is to study the effectiveness of different heuristics, different internal representation of programs, faults, and the ability to add fault-tolerance to different types of faults. There are several possible optimizations that can be applied to the framework to reduce the synthesis time. However, these optimizations are orthogonal to the issues at hand. For example, the techniques that are used to determine if a given group of transitions violates safety or if a given group of transitions is appropriate for adding recovery equally affect the above-mentioned goals. (One can either take advantage of SAT solvers to check the safeness of a group of transitions, or systematically check every transition of a given group of transitions.) While the design of the framework permits one to use these techniques, these techniques are not included in the current version as they are orthogonal to the issue of adding heuristics that focuses on (i) *which* recovery transitions should be added, (ii) *how* one should deal with safety-violating transitions, and so on. In other words, it is expected that the relative improvement of these optimizations will have the same effect on different heuristics.

Third, we are investigating the deployment of our framework on a parallel platform where we can take advantage of the processing power of multi-processors (respectively, multi-computers). As a result, we will be able to deal with programs with large state spaces.

Educational applications. Using our framework provides the opportunity to experience non-trivial concepts regarding distributed and fault-tolerant systems. We have used the synthesis framework in the graduate distributed system class as well as in a seminar on fault-tolerance.

In the class on distributed systems, the students find that the interactive nature of the framework is extremely useful in understanding several concepts about fault-tolerant programs. In this class, the students focused on re-synthesizing a fault-tolerant program for which the framework had been used successfully. In this case, the students began with the fault-intolerant program. First, they used the automated approach to obtain the fault-tolerant program. Subsequently, they focused

on interactive synthesis of the same fault-tolerant program. During this interactive synthesis, they applied different heuristics and observed the intermediate program. They explored the state transition diagram of the intermediate program and used the framework to understand why the intermediate program was not fault-tolerant. This allowed them to experience the non-deterministic execution of different processes of the program. Moreover, they could observe individual states and transitions in the global state transition diagram and could experience the effect of distribution restrictions on the complexity of the synthesis of fault-tolerant distributed programs.

In the seminar on fault-tolerance, the students used the framework to synthesize new fault-tolerant programs. They also used the intermediate versions of the program being synthesized to identify new heuristics. We would also like to note that our framework is also being used by Felix Gaertner, at Swiss Federal Institute of Technology, and Arshad Jhumka, at Chalmers University of Technology in Sweden.

To summarize, we argue that our synthesis framework has the potential to synthesize fault-tolerant programs with large state spaces (using state space reduction techniques proposed for model checking). It shows that small/moderate fault-tolerant programs that tolerate different types of faults can be synthesized efficiently. And, there is a potential to synthesize programs with larger state space by using optimizations developed in the literature.

8 Concluding Remarks and Future Work

In this paper, we presented a framework for adding fault-tolerance to existing fault-intolerant programs. Our notion of *program* refers to the abstract structure of programs (cf. Section 2), represented in Dijkstra's guarded command language [12]. Thus, the input to our framework is an abstract structure of the fault-intolerant program. The framework synthesizes the abstract structure of the fault-tolerant program.

We showed that our framework is extensible in that it permits easy addition of new heuristics that help in reducing the complexity of adding fault-tolerance. The framework also allows one to partially change the internal representation of different entities used in the synthesis while reusing other entities. These abilities are especially useful for testing different heuristics as well as testing the effect (in terms of space, time, etc.) of different internal representations of entities involved in synthesis. Finally, since we have developed the framework in Java, it is platform-independent; we have used this framework on Windows/Solaris environment. We also find that the choice of this implementation makes our framework suitable for pedagogical purposes.

Using our framework, we have synthesized fault-tolerant programs for, among others, token ring, agreement in the presence of Byzantine faults, and agreement in the presence of Byzantine and failstop faults. Thus, these examples demonstrate that the framework can be applied for the cases where we have different types of faults (process restart, Byzantine and failstop), and for

the case where a program is subject to multiple simultaneous faults. The input fault-intolerant programs used for these and other examples and the output fault-tolerant programs generated by our framework are available at [17].

Our approach differs from the previous work (e.g., [11, 24–27]) that focuses on synthesizing (fault-intolerant/fault-tolerant) programs from their specification. In the case where we need to modify an existing program to add fault-tolerance, we expect that reusing the existing program will be beneficial. As a result, the fault-tolerant program has the potential to preserve properties of fault-intolerant program that are hard to specify [26, 27] in a specification-based approach (e.g., efficiency). Moreover, as mentioned in Section 7, the framework provides a potential to synthesize efficient programs.

There are several future directions to this work. In [2], we have identified a class of specifications and programs for which failsafe fault-tolerance can be added in polynomial time (in the state space of the fault-intolerant program). We are currently developing heuristics that can study the structure of programs/specifications to determine if these conditions are met. With the use of these heuristics, we will be able to provide guarantees about finding a fault-tolerant program when it exists.

Another future direction includes the ability to add pre-synthesized fault-tolerance components during synthesis. Specifically, we have identified commonly occurring patterns in the specifications of components used in [10, 19, 28]. When these patterns are detected during synthesis, we can synthesize the corresponding components efficiently and use those components during synthesis of fault-tolerant programs. Yet another extension of the framework is to take advantage of the structural similarity of the processes in order to reduce the complexity of synthesis. In [26, 27], authors have identified techniques that reduce the complexity of synthesizing a (fault-intolerant) program from its specification. We are investigating how those techniques can be used for adding fault-tolerance to a fault-intolerant program.

Acknowledgment. We would like to thank Felix Gaertner and Arshad Jhumka for their comments and suggestions in improving the framework.

References

[1] S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. *Formal Techniques in Real-Time and Fault-Tolerant Systems*, 2000.

[2] S. S. Kulkarni and A. Ebnenasir. The complexity of adding failsafe fault-tolerance. *International Conference on Distributed Computing Systems*, 2002.

[3] S. S. Kulkarni, A. Arora, and A. Chippada. Polynomial time synthesis of byzantine agreement. *Symposium on Reliable Distributed Systems*, 2001.

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

[5] Felix C. Gaertner and Arshad Jhumka. Automating the addition of fail-safe fault-tolerance: Beyond fusion-closed specifications. Technical Report IC/2003/23, Swiss Federal Institute of Technology (EPFL), 2003.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1995.

[7] Spin language reference. `http://spinroot.com/spin/Man/promela.html`.

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

[9] A. Arora and M. G. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19(11):1015–1027, 1993.

[10] S. S. Kulkarni. *Component-based design of fault-tolerance*. PhD thesis, Ohio State University, 1999.

[11] P. Attie and A. Emerson. Synthesis of concurrent programs for an atomic read/write model of computation. *ACM TOPLAS (a preliminary version of this paper appeared in PODC96)*, 23(2), March 2001.

[12] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1990.

[13] Gerard Holzmann. Logic verification of ansi-c code with spin. *The Sixth SPIN Workshop*, 2000.

[14] M.G. Gouda and T. McGuire. Correctness preserving transformations for network protocol compilers. *Prepared for the Workshop on New Visions for Software Design and Productivity: Research and Applications*, 2001.

[15] M Nesterenko and A Arora. Stabilization-preserving atomicity refinement. *Journal of Parallel and Distributed Computing*, 62(5):766–791, 2002.

[16] M. Demirbas and A. Arora. Convergence refinement. *International Conference on Distributed Computing Systems*, 2002.

[17] A framework for automatic synthesis of fault-tolerance. `http://www.cse.msu.edu/~sandeep/software/Code/synthesis-framework/`.

[18] Sandeep S. Kulkarni and Ali Ebnenasir. A framework for automatic synthesis of fault-tolerance. Technical Report MSU-CSE-03-16, Computer Science and Engineering, Michigan State University, East Lansing MI 48824, Michigan, July 2003.

[19] A. Arora and S. S. Kulkarni. Component based design of multitolerant systems. *IEEE Transactions on Software Engineering*, 24(1):63–78, January 1998.

[20] E.M. Clarke, E.A.Emerson, and A.P.Sistla. Automatic verification of finite-state concurrent systems using temporal logic specification. *ACM Transactions on Programming Languages and Systems*, 1986.

[21] O. Kupferman and M.Y. Vardi. Synthesizing distributed systems. In *Proc. 16th IEEE Symp. on Logic in Computer Science*, July 2001.

[22] Audun Jusang. Security protocol verification using spin. *The First SPIN Workshop*, 1995.

[23] Gregory Duval and Jacques Julliand. Modeling and verification of rubis micro-kernel with spin. *The First SPIN Workshop*, 1995.

[24] A. Arora, P. C. Attie, and E. A. Emerson. Synthesis of fault-tolerant concurrent programs. *Proceedings of the 17th ACM Symposium on Principles of Distributed Computing (PODC)*, 1998.

[25] 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.

[26] P. Attie and E. Emerson. Synthesis of concurrent systems with many similar processes. *ACM Transactions on Programming Languages and Systems*, 20(1):51–115, 1998.

[27] P. Attie. Synthesis of large concurrent programs via pairwise composition. *CONCUR'99: 10th International Conference on Concurrency Theory*, 1999.

[28] A. Arora and S. S. Kulkarni. Detectors and correctors: A theory of fault-tolerance components. *International Conference on Distributed Computing Systems*, pages 436–443, May 1998.