DiConic Addition of Failsafe Fault-Tolerance*

Ali Ebnenasir
Computer Science Department
Michigan Technological University
Houghton MI 49931, USA
aebnenas@mtu.edu

ABSTRACT

We present a divide-and-conquer method, called DiConic, for automatic addition of failsafe fault-tolerance to distributed programs, where a failsafe program guarantees to meet its safety specification even when faults occur. Specifically, instead of adding fault-tolerance to a program as a whole, we separately revise program actions so that the entire program becomes failsafe fault-tolerant. Our DiConic algorithm has the potential to utilize the processing power of a large number of machines working in parallel, thereby enabling automatic addition of failsafe fault-tolerance to distributed programs with a large number of processes. We formulate our DiConic synthesis algorithm in terms of the satisfiability problem and demonstrate our approach for the classic Byzantine Generals problem and an industrial application.

Categories and Subject Descriptors: D.1.2 [Programming Techniques]: Automatic Programming; D.2.4 [Software Engineering]: Software/Program Verification; C.4 [Performance of Systems]: Fault Tolerance; D.2.2 [Software Engineering]: Design Tools and Techniques; C.2.4 [Computer-Communication Networks]: Distributed Systems.

General Terms: Reliability, Verification, Design.

Keywords: Addition of fault-tolerance, Formal methods, Divide and conquer, Satisfiability.

1. INTRODUCTION

Software intensive systems are inevitably subject to the occurrence of unanticipated faults as it is difficult (if not impossible) to anticipate all types of faults in early stages of development. Automatic addition of fault-tolerance to programs provides a systematic approach for equipping an existing program with necessary fault-tolerance functionalities while preserving its correctness in the absence of faults. However, the exponential complexity of automatic addition of fault-tolerance to *distributed* programs [22] urges us to devise new paradigms for the addition of fault-tolerance that scale better than existing techniques. In this paper, we present a *DIvide-and-CONquer* paradigm, called DiConic, that decomposes the problem of automatic addition of *failsafe* fault-tolerance – where a failsafe program meets its safety specification even when faults occur – into a set of sub-problems, where in each sub-problem, we focus on revising an individual program action (specified in terms of Dijkstra's guarded commands language [9]) in such a way that the entire program becomes failsafe fault-tolerant.

Most existing approaches [17, 25, 23, 28, 8, 29, 5, 18, 20, 13, 7] for automated synthesis of fault-tolerant programs generate an *integrated* model of a program (implemented as a reachability graph in enumerative techniques [18, 20, 13] or as a set of Binary Decision Diagrams (BDDs) in symbolic methods [29, 7]) which makes it difficult to scale such methods as the size[1] of the program increases. For example, specification-based approaches [17, 5] synthesize the abstract structure of a fault-tolerant program as a finite-state machine from the satisfiability proof of its temporal logic specification. Control-theoretic techniques [23, 8] use synchronous automata-theoretic product to generate finite state automata that represent fault-tolerant discrete-event controllers. Game-theoretic methods [25, 28] synthesize a winning strategy that captures all program behaviors. Previous work on addition of fault-tolerance to concurrent and distributed programs [18, 20, 13] takes an integrated model (represented as a finite-state machine) of an existing program and automatically revises the program model in order to generate a model of a fault-tolerant version thereof. Symbolic techniques [29, 7] enable the synthesis of larger programs, however, they still suffer from the complexity of synthesis as they deal with the synthesis problem as a whole without decomposing the problem. While all aforementioned approaches present important and useful techniques for synthesizing fault-tolerant programs and mitigating space/time complexity of addition, we believe that decomposing the synthesis problem so that each sub-problem can be solved separately is equally important and is orthogonal to complexity issues.

In order to decompose the problem of automatic addition of failsafe fault-tolerance, we focus on the following questions: Given a *fault-intolerant* program that satisfies its

*This work was sponsored by a grant from Michigan Technological University.

[1]In this work, we determine the size of the program by the number of its processes, which has a direct relation with the total number of program actions, variables, and state space.

specification in the absence of faults, but provides no guarantees in the presence of faults, *how can we determine whether or not each program action should be revised so that the entire program becomes fault-tolerant?* If indeed a program action should be revised, *how do we independently identify the nature of the revision for that action?* These questions are indeed special cases of a more general question, which is the focus of our research: Given a program that meets its specification and a new desired property raised due to the change in requirements, *how do we revise each program action so that the revised program satisfies the new property while preserving its existing specification?* To address this question for the addition of failsafe fault-tolerance, we present a sound distributed algorithm that has two components: *synthesis coordinator* and *synthesis node*. Corresponding to each program action, we instantiate a synthesis node that determines how that action should be revised towards the synthesis of a failsafe version of the input program. The synthesis coordinator synchronizes the activities of synthesis nodes. There is only one synchronization point between a synthesis node and the synthesis coordinator, which helps decreasing the communication cost of our algorithm. As the number of program actions increases one can instantiate new synthesis nodes and deploy them on different parallel machines.

Our proposed synthesis algorithm is a distributed fixpoint computation where in each round of the computation synthesis nodes report their results to the synthesis coordinator. More specifically, our algorithm takes a fault-intolerant program (as a set of actions) and a set of fault actions, and instantiates (i) N synthesis nodes, where N is the total number of program and fault actions, and (ii) one coordinator node. Each synthesis node performs a set of verification and revision operations that could be done in parallel with revisions performed on other actions in other synthesis nodes. Afterwards, each synthesis node reports its revised action to the coordinator node and waits for coordinator's reply. At the end of each round, the collection of the actions in all synthesis nodes comprises an *intermediate program*. The coordinator verifies some constraints on the intermediate program and broadcasts the results to all synthesis nodes that are waiting to enter the next round of synthesis. A synthesis node terminates in two possible cases. First, the synthesis node *locally* determines that its associated action should be removed in the failsafe program. Second, the synthesis node receives a termination message from the coordinator. The coordinator terminates either by finding a failsafe program or by declaring failure.

We have used the Yices Satisfiability Modulo Theories (SMT) solver [10] (developed at SRI International) and symbolic techniques to apply our DiConic approach to several case studies (available in [12]) among which (i) the classic problem of Byzantine Generals (BG) [24], and (ii) a simplified version of an altitude switch controller program that is similar to its manually designed version (developed at the Naval Research Laboratory [6]). The organization of the paper is as follows: We present preliminary concepts in Section 2. In Section 3, we represent the problem of adding failsafe fault-tolerance. Subsequently, in Section 4, we reiterate an enumerative solution due to Kulkarni *et al.* [19] where they add failsafe fault-tolerance to distributed programs represented as finite-state machines. Then, in Section 5, we present our DiConic algorithm. We use the BG problem as a running example to demonstrate our approach. In Section 6, we use our DiConic approach to add failsafe fault-tolerance to a simplified version of an altitude switch controller (from [6]). We summarize our case studies in Section 7. We discuss related work in Section 8. Finally, we make concluding remarks and discuss future work in Section 9.

2. PRELIMINARIES

In this section, we provide formal definitions of programs, problem specifications, faults, and failsafe fault-tolerance. The definition of specification is adapted from Alpern and Schneider [1]. The definition of programs, faults and fault-tolerance is adapted from Arora and Gouda [2] and Arora and Kulkarni [3]. The issues of modeling distributed programs is adapted from Attie and Emerson [4], and Kulkarni and Arora [18]. To illustrate our modeling approach, we use the Byzantine Generals (BG) problem [24] as a running example and use sans serif font for the exposition of the BG example.

Programs and processes. A *program* $p = \langle V_p, \Pi_p \rangle$ is a tuple of a finite set V_p of variables and a finite set Π_p of processes. Each variable $v_i \in V_p$, for $1 \leq i \leq n$, has a finite non-empty domain D_i. A *state* s of a program p is a valuation $\langle d_1, d_2, \cdots, d_n \rangle$ of program variables $\langle v_1, v_2, \cdots, v_n \rangle$, where d_i is a value in D_i. The *state space* $\mathcal{Q}_p$ is the set of all possible states of p. A *transition* of p is of the form (s, s'), where s and s' are program states. A *process* P_j , $1 \leq j \leq k$, includes a finite set of actions. An action is a guarded command (due to Dijkstra [9]) of the form $grd \rightarrow stmt$, where grd is a Boolean expression specified over V_p and $stmt$ is an assignment that *atomically* updates zero or more variables. An assignment always terminates once executed. The set of program actions is the union of the actions of all its processes. We represent the new values of updated variables as *primed* values. For example, if an action updates the value of an integer variable v_1 from 0 to 1, then we have $v_1 = 0$ and $v_1' = 1$.

BG example. We consider a canonical version of the Byzantine generals problem [24] where there are 4 distributed processes P_g, P_j, P_k, and P_l such that P_g is the general and P_j, P_k, and P_l are the non-generals. (An identical explanation is applicable for arbitrary number of non-generals.) In the fault-intolerant BG program, the general sends its decision to non-generals and subsequently non-generals output their decisions. Thus, each process has a variable d to represent its decision, a Boolean variable b to represent if that process is Byzantine, and a variable f to represent if that process has finalized (output) its decision. A Byzantine process may arbitrarily change its d and/or f values. The program variables and their domains are as follows:

$d.g : \{0,1\}$; $d.j, d.k, d.l : \{0,1,\perp\}$
// $\perp$ denotes uninitialized decision
$b.g, b.j, b.k, b.l : \{true, false\}$
// $b.j = true$ iff (if and only if) P_j is Byzantine
$f.j, f.k, f.l : \{false, true\}$
// $f.j = true$ iff P_j has finalized its decision

We represent the actions of the non-general process P_j as follows. We label these actions with BG_1 and BG_2. (The actions of other non-generals are similar.)

$$\begin{array}{llll} BG_1: & d.j = \perp \;\wedge\; \neg f.j & \longrightarrow & d.j := d.g; \\ BG_2: & d.j \neq \perp \;\wedge\; \neg f.j & \longrightarrow & f.j := true; \end{array}$$

A non-general process that has not yet decided copies the decision of the general. When a non-general process decides, it can finalize its decision.

State and transition predicates. A *state predicate* of p is a subset of $\mathcal{Q}_p$ specified as a Boolean expression over V_p.[2] An *unprimed* state predicate is specified only in terms of unprimed variables. Likewise, a *primed* state predicate includes only primed variables. A *transition predicate* (adapted from [11, 14]) is a subset of $\mathcal{Q}_p \times \mathcal{Q}_p$ represented as a Boolean expression over both unprimed and primed variables. We say a state predicate X *holds at a state* s (respectively, $s \in X$) if and only if (iff) X evaluates to true at s. Note that, a state predicate X also represents a transition predicate that includes all transitions (s, s'), where either $s \in X$ and s' is an arbitrary state, or s is an arbitrary state and $s' \in X$. An action $grd \rightarrow stmt$ is enabled at a state s iff grd holds at s. A process $P_j \in \Pi_p$ is enabled at s iff there exists an action of P_j that is enabled at s. We define a function *Primed* (respectively, *UnPrimed*) that takes a state predicate X (respectively, X') and substitutes each variable in X (respectively, X') with its primed (respectively, unprimed) version, thereby returning a state predicate X' (respectively, X). The function *getPrimed* (respectively, *getUnPrimed*) takes a transition predicate T and returns a primed (respectively, an unprimed) state predicate representing the set of destination (respectively, source) states of all transitions in T.

BG example. We define a state predicate $\mathcal{I}_1$ that captures the set of states in which the general is not Byzantine and at most one non-general could be Byzantine. (Process variables p and q represent non-general processes in the quantifications.)

$$\begin{aligned}\mathcal{I}_1 = \; & \neg b.g \wedge (\neg b.j \vee \neg b.k) \wedge (\neg b.k \vee \neg b.l) \wedge (\neg b.l \vee \neg b.j) \\ & \wedge (\forall p :: \neg b.p \Rightarrow (d.p = \bot \vee d.p = d.g)) \\ & \wedge (\forall p :: (\neg b.p \wedge f.p) \Rightarrow (d.p \neq \bot))\end{aligned}$$

In this case, a non-general non-Byzantine process is either undecided or its decision is the same as that of general. Moreover, all non-general non-Byzantine processes that are finalized have decided on a non-$\bot$ value. As another example, the state predicate $\mathcal{I}_2$ captures a set of states where the general is Byzantine and all non-generals have decided on the same value.

$$\mathcal{I}_2 = \; b.g \wedge \neg b.j \wedge \neg b.k \wedge \neg b.l \wedge \; (d.j = d.k = d.l \wedge d.j \neq \bot)$$

Closure. A state predicate X is *closed in an action* $grd \rightarrow stmt$ iff executing $stmt$ from a state $s \in (X \wedge grd)$ results in a state in X. We say a state predicate X is *closed in a program* p iff X is closed all actions of p.

BG example. The state predicates $\mathcal{I}_1$ and $\mathcal{I}_2$ are closed in actions BG_1 and BG_2. We leave it to the reader to verify this claim.

Program computations and execution semantics. We consider a nondeterministic interleaving of all program actions generating a sequence of states. A *computation* of a program $p = \langle V_p, \Pi_p \rangle$ is a sequence $\sigma = \langle s_0, s_1, \cdots \rangle$ of states that satisfies the following conditions: (1) for each transition (s_i, s_{i+1}) $(i \geq 0)$ in σ, there exists an action $grd \rightarrow stmt$ in some process $P_j \in \Pi_p$ such that grd holds at s_i and the execution of $stmt$ at s_i yields s_{i+1}; (2) σ is *maximal* in that either σ is infinite or if it is finite, then no action is enabled in its final state, and (3) σ is *fair*; i.e., if a process P_j is continuously enabled in σ, then eventually some action of P_j will be executed.

[2]An individual state s, the empty set, and the entire state space (i.e., the universal set) are special cases of a state predicates.

Distribution model. We model distribution issues as a set of read/write restrictions imposed on program processes. More specifically, we associate with each process P_j a set of variables that it is allowed to read, denoted r_j, and a set of variables that P_j can write, denoted w_j. We assume that for each process P_j, $w_j \subseteq r_j$; i.e., if a process can write a variable, then that variable is readable too. No action in a process P_j is allowed to update a variable $v \notin w_j$. In addition to the read/write restrictions for each process, we allow additional write constraints on actions in that a variable that is writeable for the process may not be writeable for a specific action. The read/write restrictions for each action can be specified as the transition predicate $rwRest \equiv (\forall v : v \notin w_j : v = v')$; i.e., the value of an unprimed variable $v \notin w_j$ should be equal to the value of its primed version. Note that the transition predicate $rwRest$ imposes a similar constraint on unreadable variables because if a variable cannot be read, then it cannot be written either. Using read/write restrictions $rwRest$, we formally specify an action $grd \rightarrow stmt$ as a transition predicate $grd \wedge Primed(stmtExpr) \wedge rwRest$, where $stmtExpr$ is a Boolean expression generated from the assignment $stmt$. For example, an assignment $x := 1$ can be specified as the expression $x' = 1$.

BG example. Each non-general non-Byzantine process P_j is allowed to only read $r_j = \{b.j, d.j, f.j, d.k, d.l, d.g\}$ and it can only write $w_j = \{d.j, f.j\}$. Hence, we have $w_j \subseteq r_j$. As an example of a transition predicate, we represent the action BG_1 as $((d.j = \bot) \wedge (f.j = 0)) \wedge (d'.j = d'.g) \wedge rwRest$, where $rwRest$ is the transition predicate $(d.k = d'.k) \wedge (d.l = d'.l) \wedge (d.g = d'.g) \wedge (b.g = b'.g) \wedge (b.j = b'.j) \wedge (b.k = b'.k) \wedge (b.l = b'.l) \wedge (f.j = f'.j) \wedge (f.k = f'.k) \wedge (f.l = f'.l)$. Note that while P_j is allowed to write $f.j$, we forbid the action BG_1 to write $f.j$ as this is the responsibility of the action BG_2.

Specifications. We follow Alpern and Schneider [1] in representing the specification *spec* of a program as the conjunction of a *safety* specification (denoted $\mathcal{S}$) and a *liveness* specification (denoted $\mathcal{L}$); i.e., $spec = \mathcal{S} \cap \mathcal{L}$. Intuitively, $\mathcal{S}$ stipulates that nothing bad ever happens. Formally, we represent $\mathcal{S}$ as a transition predicate whose transitions must not appear in program computations. An action $AC : grd \rightarrow stmt$ *satisfies* $\mathcal{S}$ from a state predicate X iff the transition predicate $\mathcal{X} \wedge grd \wedge Primed(stmtExpr) \wedge \mathcal{S} \wedge rwRest$ is equal to the empty set, where $rwRest$ captures the read/write restrictions of AC. A program p satisfies its safety specification $\mathcal{S}$ from $\mathcal{X}$ iff all actions of p satisfy $\mathcal{S}$ from $\mathcal{X}$. In terms of program computations, a program satisfies $\mathcal{S}$ from $\mathcal{X}$ iff all program computations starting in $\mathcal{X}$ satisfy $\mathcal{S}$. A program computation $c = \langle s_0, s_1, \cdots \rangle$ *satisfies* $\mathcal{S}$ from $\mathcal{X}$ iff $(s_0 \in \mathcal{X}) \wedge (\forall (s_i, s_{i+1}) : i \geq 0 : (s_i, s_{i+1}) \notin \mathcal{S})$. Otherwise, the computation c *violates* $\mathcal{S}$. The liveness specification $\mathcal{L}$ states that something good will eventually occur. More precisely, we define $\mathcal{L}$ as a set of infinite sequences of states. A computation is in $\mathcal{L}$ if it contains a suffix[3] that is in $\mathcal{L}$. Note that there may exist infinitely long sequences that have no suffix in the particular set of infinite sequences specified by

[3]A suffix of a sequence $\langle s_0, s_1, \cdots \rangle$ is a subsequence $\langle s_j, s_{j+1}, \cdots \rangle$ for some $j \geq 0$.

$\mathcal{L}$. For example, given a state s that does not appear in any sequence $\langle s_0, s_1, \cdots \rangle$ belonging to $\mathcal{L}$, the infinite sequence generated by a self-loop on s would not belong to $\mathcal{L}$. A computation $c = \langle s_0, s_1, \cdots \rangle$ *satisfies* $\mathcal{L}$ from a state predicate X iff ($s_0 \in \mathcal{X}$) and c has a suffix in $\mathcal{L}$. We say a program p satisfies its liveness specification $\mathcal{L}$ from a state predicate X iff all computations of p satisfy $\mathcal{L}$ from X. A program p *satisfies its specification spec* from a state predicate X iff p satisfies its safety and liveness specifications from X.

BG example. The safety specification of the BG program requires that *Validity* and *Agreement* be satisfied. Validity stipulates that if the general is not Byzantine and a non-Byzantine non-general has finalized its decision, then the decision of that non-general process is the same as that of the general. Agreement requires that if two non-Byzantine non-generals have finalized their decisions, then their decisions are identical. Hence, the program should not execute transitions that reach the primed state predicate $\mathcal{R}_1$, where

$$\begin{aligned}\mathcal{R}_1 = \; & (\exists p, q :: \neg b'.p \wedge \neg b'.q \wedge d'.p \neq \bot \wedge d'.q \neq \bot \wedge \\ & \qquad d'.p \neq d'.q \wedge f'.p \wedge f'.q) \vee \\ & (\exists p :: \neg b'.g \wedge \neg b'.p \wedge d'.p \neq \bot \wedge d'.p \neq d'.g \wedge f'.p)\end{aligned}$$

Moreover, when a non-Byzantine process finalizes, it is not allowed to change its decision. Thus, the set of transitions of the following transition predicate should not be executed as well:

$$\mathcal{S}_2 = \; \neg b.j \wedge \neg b'.j \wedge f.j \;\wedge\; (d.j \neq d'.j \vee f.j \neq f'.j)$$

Let $\mathcal{S}_1$ be the transition predicate representing all transitions that reach a state in $\mathcal{R}_1$. We specify the safety specification of the BG program as the transition predicate $\mathcal{S}_1 \vee \mathcal{S}_2$.

Invariants. A state predicate $\mathcal{I}$ is an invariant of a program p for its specification *spec* iff the following conditions are satisfied: (1) $\mathcal{I}$ is closed in p, and (2) p satisfies *spec* from $\mathcal{I}$.

BG example. The state predicate $\mathcal{I}_{BG} = \mathcal{I}_1 \vee \mathcal{I}_2$ is indeed an invariant of the BG program.

Faults and fault-span. We represent a fault-type F as a set of actions. Fault actions differ from program actions in that the program does not have execution control over fault actions. A *computation of a program* $p = \langle V_p, \Pi_p \rangle$ *in the presence of fault* F is a sequence $\sigma = \langle s_0, s_1, \cdots \rangle$ of states that satisfies the following conditions: (1) for each transition (s_i, s_{i+1}) $(i \geq 0)$ in σ, there exists either a program or a fault action $grd \rightarrow stmt$ such that grd holds at s_i and the execution of $stmt$ at s_i yields s_{i+1}; (2) σ is *maximal*, and (3) σ is *fair*. A state predicate $\mathcal{FS}$ is called a fault-span of a program p from its invariant $\mathcal{I}$ for fault F (denoted F-span) iff the following conditions are satisfied: (1) $\mathcal{I} \Rightarrow \mathcal{FS}$, and (2) $\mathcal{FS}$ is closed in program p and fault actions. Intuitively, the F-span of p from $\mathcal{I}$ is a boundary around $\mathcal{I}$ to which the state of p can be perturbed by fault and program actions.

BG example. The fault action F_1 may cause *at most* one non-Byzantine process to become Byzantine. A Byzantine process may arbitrarily change its d and/or f values. (We include similar fault actions for k, l and g.)

$$\begin{array}{lll} F_1: \; \neg b.g \wedge \neg b.j \wedge \neg b.k \wedge \neg b.l & \longrightarrow & b.j := true \\ F_2: \; b.j & \longrightarrow & d.j, f.j := 0|1, false|true \end{array}$$ [4]

In Section 5, we shall illustrate how to calculate the fault-span in a DiConic manner.

Failsafe fault-tolerance. A program p is *failsafe* F*-tolerant* from an invariant $\mathcal{I}$ for its specification *spec* (i.e., fault-tolerant against a fault-type F) iff there exists an F-span $\mathcal{FS}$ such that: (1) in the absence of fault F, p satisfies *spec* from $\mathcal{I}$, and (2) in the presence of fault F, the actions of p and F satisfy the safety of *spec* (i.e., $\mathcal{S}$) from $\mathcal{FS}$.

[4] $d.j := 0|1$ means that $d.j$ could be assigned either 0 or 1.

3. PROBLEM STATEMENT

In this section, we reiterate the problem of adding failsafe fault-tolerance to programs from [18]. Specifically, to formulate our DiConic approach in terms of the satisfiability problem, we represent the addition problem in terms of state/transition predicates.

In order to separate fault-tolerance from functional concerns, the problem of adding fault-tolerance stipulates that no new behaviors are added to programs in the *absence* of faults. More precisely, given a program $p = \langle V_p, \Pi_p \rangle$, its invariant $\mathcal{I}$, its specification $spec = \mathcal{S} \cap \mathcal{L}$ and a fault-type F, we aim to generate a revised version of p, denoted $p^f = \langle V_p, \Pi_p^f \rangle$ with a new invariant $\mathcal{I}^f$ such that p^f is failsafe F-tolerant from $\mathcal{I}^f$ for *spec*. If $\mathcal{I}^f$ includes a state s that does not belong to $\mathcal{I}$, then the execution of p^f in the absence of F from s may generate new computations. Hence, we require that $\mathcal{I}^f \Rightarrow \mathcal{I}$. Moreover, starting in $\mathcal{I}^f$, the actions of p^f should not include new transitions. Otherwise, p^f may exhibit new behaviors in the absence of faults. Thus, for each action $grd^f \rightarrow stmt^f$ in p^f , we require that there exists an action $grd \rightarrow stmt$ in p such that the transition predicate $\mathcal{I}^f \wedge grd^f \wedge Primed(stmt^f Expr) \wedge rwRest$ implies the transition predicate $\mathcal{I}^f \wedge grd \wedge Primed(stmtExpr) \wedge rwRest$. Therefore, we formally state the problem of adding failsafe fault-tolerance as follows:

The Problem of Adding Failsafe Fault-Tolerance.
Given p, $\mathcal{I}$, *spec*, and F, identify p^f and $\mathcal{I}^f$ such that

(1) $\mathcal{I}^f \Rightarrow \mathcal{I}$ and $\mathcal{I}^f \not\equiv false$,

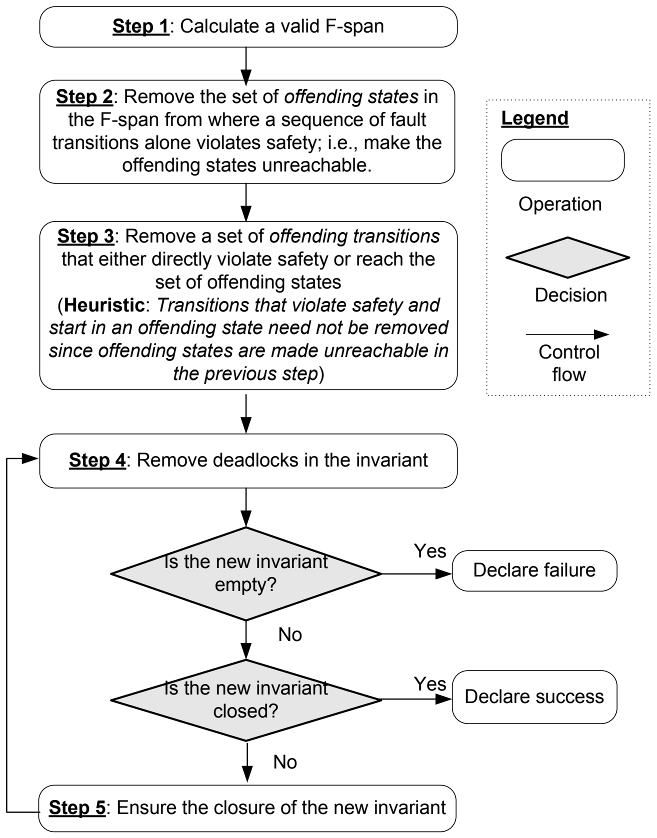
(2) For each action $grd^f \rightarrow stmt^f$ in p^f, there exists some action $AC : grd \rightarrow stmt$ in p such that $(\mathcal{I}^f \wedge grd^f \wedge Primed(stmt^f Expr) \wedge rwRest) \Rightarrow$ $(\mathcal{I}^f \wedge grd \wedge Primed(stmtExpr) \wedge rwRest)$ where $rwRest$ captures the read/write restrictions of AC.

(3) p^f is failsafe F-tolerant from $\mathcal{I}^f$ for *spec*. $\square$

4. ADDING FAILSAFE FAULT-TOLERANCE

In order to simplify the presentation of our DiConic approach, in this section, we present an intuitive description of an algorithm presented in previous work [19, 13] for the addition of fault-tolerance to integrated models of distributed programs. This algorithm (see Figure 1) takes a non-deterministic finite-state machine representing the fault-intolerant program and generates a state machine representing the failsafe fault-tolerant program.

The algorithm in Figure 1 first identifies a *valid* fault-span of the program that is a superset of the invariant and is closed in program and fault transitions. Subsequently, the algorithm computes and removes the set of *offending states* from where a sequence of fault transitions can directly violate safety. The removal of offending states is accomplished by making them unreachable. Then the algorithm identifies and removes a set of offending transitions that either directly violate safety or reach an offending state. However, since the



Step 1: Calculate a valid F-span
Step 2: Remove the set of offending states in the F-span from where a sequence of fault transitions alone violates safety; i.e., make the offending states unreachable.
Step 3: Remove a set of offending transitions that either directly violate safety or reach the set of offending states
(Heuristic: Transitions that violate safety and start in an offending state need not be removed since offending states are made unreachable in the previous step)
Step 4: Remove deadlocks in the invariant
Is the new invariant empty?
Yes
Declare failure
No
Is the new invariant closed?
Yes
Declare success
No
Step 5: Ensure the closure of the new invariant
Legend
Operation
Decision
Control flow

Figure 1: Adding failsafe fault-tolerance to integrated models.

offending states are made unreachable in Step 2, offending transitions that start in an offending state need not be removed. While removing offending states/transitions, the addition algorithm may introduce states that have no outgoing transitions; i.e., *deadlock* states. The exclusion of deadlock states from the invariant may leave transitions that start in the new invariant and reach a recently removed state, thereby violating the closure of the new invariant. These transitions are removed in Step 5. The algorithm iterates through Steps 4 and 5 until either all invariant states are removed or a closed non-empty invariant is found. Existing implementations [19, 13, 7] of the algorithm in Figure 1 execute all steps on an integrated model represented either as a non-deterministic finite-state machine [13] or as a set of symbolic entities (e.g., BDDs) [7], thereby making it difficult to scale these algorithms as the number of program processes increases.

5. DICONIC ADDITION

In this section, we present our DiConic algorithm. Specifically, in Section 5.1, we illustrate how to compute the fault-span and the set of offending states (Steps 1 and 2 in Figure 1). In Section 5.2, we present a DiConic method for removing offending transitions (Step 3 in Figure 1). Finally, in Section 5.3, we calculate a new invariant for the synthesized failsafe program in a DiConic fashion. This step corresponds to the loop that includes Steps 4 and 5 in Figure 1.

5.1 Computing Fault-Span and Offending States

In this section, we decompose the problem of calculating a fault-span and the set of offending states by introducing a distributed forward/backward reachability algorithm. Specifically, in calculating the fault-span using forward reachability, the integrated algorithms [19, 13, 7] implement a fixpoint computation that explicitly explores the set of states reachable from the invariant of the fault-intolerant program by all program and fault transitions. In each iteration, the forward reachability algorithm computes a new set of states reachable from previously calculated states. Continuing thus, the forward reachability algorithm reaches a point where no more states are reachable. The union of the sets of all states computed in all iterations comprises the fault-span of the fault-intolerant program. Instead of calculating a set of states reachable from a given state predicate X, in each iteration, the backward reachability algorithm computes a set of states from where X is reached.

Our proposed DiConic approach for the calculation of the fault-span (respectively, the set of offending states) is a divide-and-conquer fixpoint computation with two kinds of components, namely the *coordinator* and the *reachability* nodes. A reachability node contains a single program action and computes the set of states reached by the execution of that action from a state predicate X (respectively, the set of states from where the execution of that action reaches a state in X). The coordinator node manages the reachability nodes. Figures 2 and 3 illustrate the two components of our DiConic approach for forward reachability. (We omit the DiConic backward reachability algorithm as it is similar to the forward reachability.) The coordinator starts by sending out a $Base$ state predicate to all nodes. Each node computes the set of states outside $Base$ that are reachable from $Base$ by that node's action (see Figure 2). The coordinator takes the union of all reachable states, denoted $reachedStates$, calculated by all nodes. If this union is empty, then the fixpoint computation is terminated. Otherwise, another iteration starts with a new $Base$, which is the union of the old $Base$ and the state predicate $reachedStates$. For the calculation of the fault-span, the $Base$ predicate is the invariant of the fault-intolerant program, and corresponding to each program and fault action, a reachability node is instantiated. Note that for the calculation of the set of offending states using DiConic backward reachability, only fault actions are considered since we want to compute the set of states from where safety is violated by a sequence of fault transitions *alone*. Specifically, for each fault action, we first identify the set of states from where that action directly violates safety. The union of all such states comprises the base set for a backward reachability computation using only fault actions. Therefore, in the rest of this section, we present our DiConic approach assuming that the fault-span, denoted FS, and the set of offending states, denoted OS, have already been calculated.

BG example. In order to calculate the fault-span of the canonical BG program, denoted $\mathcal{FS}_{BG}$, we instantiate 14 forward reachability nodes as we have 2 program actions and 2 fault actions corresponding to each non-general process and 2 fault actions corresponding to the general process. We also create a reachability coordinator. The base predicate is equal to the invariant $\mathcal{I}_{BG}$. We consider only the program/fault actions of process P_j as they are structurally similar to other

```
ForwardReachability_Node($grd \rightarrow stmt$: action; $X$: state predicate;
                                        $rwRest$: transition predicate)
{
  - Wait for $X$ from the coordinator;
  - repeat {
      - $reachedStates := \emptyset$; // set of states reached outside $X$ by
                              // the execution of $grd \rightarrow stmt$ from $X$;
      - $transPred := (grd \wedge X) \wedge$ Primed($stmtExpr$) $\wedge$
                                     $rwRest \wedge$ Primed($\neg X$);
      - If ($transPred$ is satisfiable) then
          - $reachedStates :=$ getPrimed($transPred$);
      - Send $reachedStates$ to coordinator;
      - Wait for $X$ from the coordinator;
  - } until(termination signal is received);
}
```

Figure 2: Reachability node in DiConic forward reachability.

```
Reachability_Coordinator($Base$: state predicate)
{ // $N$ denotes the total number of program and fault actions.
  - $X := Base$;
  - repeat {
      - Send $X$ to all nodes;
      - Wait for $reachedStates_i$ from Node$_i$;
      - $reachedStates := reachedStates_1 \vee \cdots \vee reachedStates_N$;
      - $X := X \vee reachedStates$;
  - } until($reachedStates = \emptyset$);
  - Send termination signal to all nodes;
  - return $X$;
}
```

Figure 3: Reachability coordinator in DiConic forward reachability.

non-general processes. In the first iteration of the fixpoint computation, starting from the invariant, program actions BG_1 and BG_2 would reach to a state in the invariant (due to the closure of the invariant in program actions).

The reachability node corresponding to the fault action F_1 returns $\mathcal{I}_1 \vee (b.g \wedge \neg b.j \wedge \neg b.k \wedge \neg b.l)$ as the set of states reachable from the invariant by F_1. Specifically, starting from the state predicate $\mathcal{I}_1$ (see Section 2) in the invariant, if the fault action F_1 does not cause the general process P_g to become Byzantine, then the set of states reached from $\mathcal{I}_1$ is the same as $\mathcal{I}_1$. Nonetheless, if P_g becomes Byzantine, then the b values of non-general processes would remain false as at most one process could be Byzantine. Thus, the set of states reached from $\mathcal{I}_1$ by faults is $(\mathcal{I}_1 \vee (b.g \wedge \neg b.j \wedge \neg b.k \wedge \neg b.l))$. If the BG program is in a state in $\mathcal{I}_2$, then the fault action F_1 will not be executed as P_g is already Byzantine.

The reachability node corresponding to the fault action F_2 returns the same state predicate as that of F_1. If P_j is Byzantine, then the execution of the action F_2 from $\mathcal{I}_1$ would not violate the closure of $\mathcal{I}_1$ since the second and the third conjuncts of $\mathcal{I}_1$ are specified for non-Byzantine processes. Moreover, F_2 will not be enabled from $\mathcal{I}_2$. If P_g is Byzantine, then F_2 will not be enabled from $\mathcal{I}_1$, thereby preserving the closure of $\mathcal{I}_1$. In this case, a similar reasoning as that for F_1 would yield $\mathcal{I}_2$ as the set of states reachable from $\mathcal{I}_2$ by F_2.

Since the state predicate $\mathcal{I}_1 \vee (b.g \wedge \neg b.j \wedge \neg b.k \wedge \neg b.l)$ is closed in all program and fault actions, the fault-span of the BG program is equal to $\mathcal{FS}_{BG}$, where

$$\mathcal{FS}_{BG} = \mathcal{I}_1 \vee (b.g \wedge \neg b.j \wedge \neg b.k \wedge \neg b.l).$$

Both fault actions may violate safety if they start in a state in $UnPrimed(\mathcal{R}_1)$ (see Section 2); i.e., the base set for backward reachability is $\mathcal{R}_1$. Subsequently, we compute the set of all states from where the execution of fault actions would yield a state in $\mathcal{R}_1$. The fault actions F_1 and F_2 yield a state in $\mathcal{R}_1$ only if they are enabled in $UnPrimed(\mathcal{R}_1)$. Therefore, the second iteration of the fixpoint computation is the last iteration, returning $UnPrimed(\mathcal{R}_1)$ as the set of offending states.

5.2 Removing Offending Transitions

We decompose the removal of the offending transitions into the elimination of such transitions in each action without actually creating an integrated model of the program. Specifically, each synthesis node investigates two cases for the existence of offending transitions in the set of transitions represented by its associated action. Steps 1 and 2 in Figure 4 identify and exclude a subclass of the offending transitions that start outside the set of offending states and reach an offending state. The motivation behind these steps is that such transitions take the program to states from where the occurrence of faults may violate safety. Thus, in order to preserve safety even when faults occur, the program must not take such transitions. Steps 3 and 4 in Figure 4 determine and remove another subclass of the offending transitions that start outside the invariant in the fault-span $\mathcal{FS}$ and directly violate the safety specification. In the next subsection, we describe how to divide the task of resolving deadlock states that may be created due to the elimination of offending transitions.

BG example. We create two synthesis nodes corresponding to the actions BG_1 and BG_2. The execution of Steps 1 and 2 in Figure 4 for the action BG_1 returns an empty set since the only way the execution of BG_1 generates a state in the set of offending states is that BG_1 is enabled in an offending state. In other words, the action BG_1 may violate safety if it is enabled in a state in the fault-span where the general is Byzantine and the other two non-generals have finalized with different decisions. However, since such states are included in the set of offending states and are removed from the fault-span, we do not have to exclude such transition from the action BG_1. Therefore, the first two steps of the Synthesis_Node algorithm do not change action BG_1.

The synthesis node corresponding to BG_2 revises BG_2 in the following way. Specifically, if the action BG_2 is enabled from a state outside the set of offending states where a non-general non-Byzantine process has finalized with a different decision from that of P_j, then the safety of specification will be violated if P_j finalizes its decision. Thus, a set of transitions starting in $(d.k \neq \bot \wedge d.j \neq d.k) \vee (d.l \neq \bot \wedge d.j \neq d.l)$ would be excluded from BG_2 in Steps 1 and 2 of Synthesis_Node. Steps 3 and 4 do not exclude any additional transitions from BG_2 as the only transitions that directly violate safety upon finalization are the same transitions excluded in Steps 1 and 2. Therefore, the revised action BG_2 is as follows:

$$\begin{aligned} BG_2:\quad & (d.j \neq \bot \;\wedge\; f.j = 0) \;\wedge \\ & (\mathbf{d.k} = \bot \vee \mathbf{d.j} = \mathbf{d.k}) \wedge (\mathbf{d.l} = \bot \vee \mathbf{d.j} = \mathbf{d.l}) \\ & \qquad\qquad \longrightarrow\; f.j := true \end{aligned}$$

In the case of the BG program, the synthesis nodes associated with the actions of P_k and P_l perform revisions similar to that of synthesis nodes associated with BG_1 and BG_2.

```
Synthesis_Node($grd \rightarrow stmt$: action; $OS, Inv, FS$: state predicate;
                                   $\mathcal{S}, rwRest$: transition predicate)
/* $\mathcal{S}$ denotes the safety specification and $FS$ denotes the fault-span. */
/* $OS$ denotes the set of offending states. */
{
  /* Does $grd \rightarrow stmt$ start outside OS and reach OS? */
  - $transPred := (FS \wedge grd \wedge \neg OS) \wedge$
                     (Primed($stmtExpr \wedge OS$)) $\wedge rwRest$;   (1)
  - If ($transPred$ is satisfiable) then
      - Exclude($transPred$, $grd \rightarrow stmt$);                  (2)

  /* Does $grd \rightarrow stmt$ directly violate safety in $FS - Inv$? */
  - $transPred := (FS \wedge \neg Inv \wedge grd \wedge$ Primed($stmtExpr$) $\wedge$
                                          $rwRest \wedge \mathcal{S}$);  (3)
  - If ($transPred$ is satisfiable) then
      - Exclude($transPred$, $grd \rightarrow stmt$);                  (4)

  /* Synchronize this node with the synthesis coordinator. */
  repeat {
      - Send $grd$ to the synthesis coordinator;                       (5)
      - If ($grd$ is unsatisfiable) then declare that
                          $grd \rightarrow stmt$ is removed; exit();   (6)
      - Wait to receive a new invariant $Inv_{new}$ and $DeadlockStates$;   (7)

  /* Does the revised action contain closure-violating transitions? */
      - $closureViolatingTrans := grd \wedge Inv_{new} \wedge$ Primed($stmtExpr$) $\wedge$
                          Primed($\neg Inv_{new}$) $\wedge rwRest$;   (8)
      - If ($closureViolatingTrans$ is satisfiable) then
          - Exclude($closureViolatingTrans$, $grd \rightarrow stmt$);  (9)
  until (($Inv_{new}$ is unsatisfiable) $\vee$ ($DeadlockStates$ is unsatisfiable));
  - If ($Inv_{new}$ is satisfiable) then return $grd \rightarrow stmt$;   (10)
  - declare failure in synthesizing a failsafe program;                (11)
}

ExcludeTransitions($transPred$: transition predicate, $grd \rightarrow stmt$: action)
// exclude the set of transitions represented by $transPred$
// from the action $grd \rightarrow stmt$.
{    - $transPred := grd \wedge Primed(stmtExpr) \wedge \neg transPred$;
     - $grd$ := getUnPrimed($transPred$); }
```

Figure 4: Synthesis node.

5.3 Computing a New Invariant

In this section, we illustrate how our DiConic approach decomposes the problem of calculating a new invariant for the failsafe program by the collaboration between the Synthesis_Node and the Synthesis_Coordinator algorithms (see Figures 4 and 5). First, the Synthesis_Coordinator algorithm ensures that the removal of offending states does not lead to the removal of all invariant states (see Steps 1 and 2 in Figure 5). Then the Synthesis_Coordinator enters a loop which synchronizes the activities of all synthesis nodes once they start participating in the calculation of the new invariant (see the loop in Steps 5-9 in Figure 4). These two loops are in fact a DiConic implementation of the loop that includes Steps 4 and 5 in Figure 1. The Synthesis_Coordinator waits to receive the revised guards from all synthesis nodes after the removal of offending transitions in Steps 1-4 of the Synthesis_Node algorithm. The synthesis nodes enter a waiting state after sending their revised guard to the coordinator. The Synthesis_Coordinator calculates the set of all invariant states from where no action is enabled (see Steps 4 and 5 in Figure 5); i.e., deadlock states. Afterwards, the coordinator computes a new invariant by removing the deadlock states. The Synthesis_Coordinator sends the new invariant and a state predicate representing deadlock states to all synthesis nodes. If the new invariant becomes empty, then Synthesis_Coordinator declares failure in the addition of failsafe fault-tolerance. After receiving the new invariant, each synthesis node determines whether or not its associated action includes transitions that reach a state outside the invariant Inv_{new} (Steps 8 and 9 in Figure 4); i.e., violates the closure of the invariant. An action violates the closure of the invariant if it contains transitions that reach states that have been excluded by other actions in other synthesis nodes. Afterwards, each synthesis node starts another iteration by sending its revised guard grd to the coordinator. The collaboration between the synthesis coordinator and synthesis nodes continues until either a valid new invariant is found or the synthesis fails.

```
Synthesis_Coordinator($Inv, OS$: state predicate)
{   /* $OS$ denotes the set of offending states. */
    /* $n$ denotes the number of program actions. */
    - $Inv_{new} := Inv \wedge \neg OS$;                                          (1)
    - If ($Inv_{new}$ is unsatisfiable) then
        - declare that a failsafe program cannot be synthesized; exit();   (2)
    repeat {
      - Wait for the revised guards $grd_1, \cdots, grd_n$
                          from synthesis nodes $Node_1, \cdots, Node_n$;   (3)
      - $Guards := grd_1 \vee \cdots \vee grd_n$;                              (4)
      - $DeadlockStates := (Inv_{new} \wedge \neg Guards)$;                    (5)
      - $Inv_{new} := Inv_{new} \wedge \neg DeadlockStates$;                   (6)
      - Send $Inv_{new}$ and $DeadlockStates$ to nodes $Node_1, \cdots, Node_n$;   (7)
    } until (($Inv_{new}$ is unsatisfiable) $\vee$ ($DeadlockStates$ is unsatisfiable));
    - If ($Inv_{new}$ is unsatisfiable) then
        - declare failure in synthesizing a failsafe program; exit();      (8)
    - Notify nodes of successful termination;
}
```

Figure 5: Synthesis coordinator.

BG example. Since the invariant $\mathcal{I}_{BG}$ does not include any offending states (i.e., $\mathcal{I}_{BG} \cap UnPrimed(\mathcal{R}_1) = \emptyset$), the revision of the action BG_2 does not create new deadlock states in the invariant. Thus, the predicate $DeadlockStates$ computed in Step 5 of Figure 5 is equal to the empty set. As a result, the synthesis nodes exit the repeat-until loop in the first iteration. Therefore, the invariant of the failsafe program is equal to $\mathcal{I}_{BG}$.

Theorem 5.1 DiConic addition of failsafe fault-tolerance is sound. (see [12] for proof) □

Comment on the completeness of our approach. The elimination of closure-violating transitions in each synthesis node may create new deadlock states in the invariant. The number of such deadlock states depends on how appropriate closure-violating transitions are selected for elimination among all program actions. The current strategy for eliminating closure-violating transitions may result in the removal of an inappropriate set of such transitions, thereby leading to the failure of the addition of fault-tolerance to the program at hand; i.e., our algorithm may fail to find a failsafe program while there exists one. Thus far, none of the case studies we have conducted failed due to the incompleteness of our approach.

Complexity analysis with respect to integrated approaches. We analyze the degree of improvement that our DiConic approach (algorithms in Figures 4 and 5) provides with respect to the time complexity of the integrated algorithm presented in Figure 1. Let n denote the number of program action. In the integrated approach, Steps 1 to 4, and Steps 8 and 9 in Figure 4 are executed for all program actions, whereas in each synthesis node those steps are performed for only one action. Thus, as far as these steps are concerned, $1/n$ of the total workload is assigned to each node. There is no improvement for the Steps 4 to 6 in Figure 5 as the current version of the synthesis coordinator performs exactly the same steps that the integrated algorithm does.

The worst case message complexity of our DiConic approach depends on the size of the program invariant. In principle, in the worst case, the loops in both Figures 4 and 5 are executed until all invariant states are removed. Thus the worst case message complexity is $O(n \cdot |Inv|)$, where $|Inv|$ represents the number of states in the invariant.

6. EXAMPLE: ALTITUDE SWITCH CONTROLLER

In this section, we demonstrate the application of our DiConic approach in adding failsafe fault-tolerance to a simplified version of an altitude switch (ASW) controller program (adapted from [6]).

The fault-intolerant altitude switch (ASW). The ASW program monitors a set of input variables and generates an output. Also, the program has a set of internal variables as follows: (i) *AltBelow* is equal to 1 if the altitude is below a specific threshold, otherwise, it is equal to 0; (ii) *ActuatorStatus* is equal to 1 if the actuator is powered on, otherwise, it is equal to 0; (iii) *Inhibit* is equal to 1 when the actuator power-on is inhibited, otherwise, it is equal to 0, and (iv) *Reset* is equal to 1 if the system is being reset.

The ASW program has a mode variable $Mode$ and can be in three different modes: (i) the *Initialization* mode when the ASW system is initializing, denoted $Mode = I$; (ii) the *Await-Actuator* mode if the system is waiting for the actuator to power on, denoted $Mode = AA$, and (iii) the *Standby* mode, denoted $Mode = SB$.

Moreover, we model the signals that come from the input sensors to indicate the occurrence of faults using the following variables: (i) if the system fails in the Initialization mode, then the variable $InitFail$ will be set to 1, otherwise, $InitFail$ remains 0; (ii) if the altitude sensors fail and do not recover in a certain number of built-in reset attempts, then the variable $AltFail$ will be equal to 1, otherwise, $AltFail$ remains 0, and (iii) if the Actuator fails in the Await-Actuator mode, then the variable $ActFail$ will be equal to 1, otherwise, $ActFail$ remains 0. The domain of all variables except $Mode$ is equal to $\{0, 1\}$. The fault-intolerant program consists of only one process with the following actions:

$$\begin{array}{lll}
A_1: & (Mode = I) & \longrightarrow \; Mode := SB; \\
A_2: & (Mode = SB) \wedge (Reset = 1) & \longrightarrow \; Mode := I; \\
 & & \qquad Reset := 0; \\
A_3: & (Mode = SB) \wedge (AltBelow = 1) \wedge (Inhibit = 0) \wedge & \\
 & (ActuatorStatus = 0) & \longrightarrow \; Mode := AA; \\
A_4: & (Mode = AA) \wedge (ActuatorStatus = 0) \wedge (Inhibit = 0) & \\
 & \qquad \longrightarrow Mode := SB;\ ActuatorStatus := 1; & \\
A_5: & (Mode = AA) \wedge (Reset = 1) & \longrightarrow \; Mode := I; \\
 & & \qquad Reset := 0;
\end{array}$$

The program changes its mode from Initialization to Standby. The program goes to the Initialization mode when it is either in Standby or in Await-Actuator mode and the reset signal is received. If the program is in the Standby mode, the altitude is below a pre-determined threshold, the actuator power-on is not inhibited and the actuator is not powered on, then the program changes its mode to Await-Actuator. In the Await-Actuator mode, the program either powers on the actuator and goes to the standby mode, or enters the initialization mode upon receiving the reset signal.

Read/Write restrictions. All variables can be read. However, the program cannot write the variables $InitFail$, $AltFail$, $ActFail$, $AltBelow$ and $Inhibit$.

Faults. If the altitude sensors incur malfunction then the state of the program will be perturbed to a faulty state. We represent the fault actions as follows:

$$\begin{array}{lll}
F_1: & (InitFail = 0) & \longrightarrow \; InitFail := 1; \\
F_2: & (AltFail = 0) & \longrightarrow \; AltFail = 1; \\
F_3: & (ActFail = 0) & \longrightarrow \; ActFail = 1;
\end{array}$$

The guards of the above actions represent conditions under which the program *detects* the occurrence of faults.

Safety specification. The problem specification requires that the program does not change its mode from Standby to Await-Actuator if the altitude sensors are failed; i.e., $AltFail$ is equal to 1. Moreover, if the initialization has failed, then the ASW program must not enter the Standby mode. Finally, if the actuator fails in the AA mode, then the ASW program should not power on the actuator. Thus, we represent the safety specification of the ASW program by the transition predicate $\mathcal{S}_{ASW}$, where

$$\begin{array}{l}
\mathcal{S}_{ASW} = \\
((AltFail = 1) \wedge (Mode = SB) \wedge (Mode' = AA)) \vee \\
((InitFail = 1) \wedge (Mode = I) \wedge (Mode' = SB)) \vee \\
((ActFail = 1) \wedge (Mode = AA) \wedge \\
\qquad (ActuatorStatus = 0) \wedge (ActuatorStatus' = 1))
\end{array}$$

This transition predicate specifies the set of transitions that must not appear in any computation of the ASW program even when faults occur.

Invariant. If the $Inhibit$ is activated, then the actuators should not be powered on. Moreover, the program should not be in a faulty mode. Thus, the invariant of the ASW program is equal to $\mathcal{I}_{ASW}$, where

$$\begin{array}{l}
\mathcal{I}_{ASW} = \\
((Inhibit = 1) \Rightarrow (ActuatorStatus = 0)) \wedge \\
((InitFail = 0) \wedge (AltFail = 0) \wedge (ActFail = 0))
\end{array}$$

DiConic calculation of the fault-span and offending states. In DiConic forward reachability on fault actions, in the first iteration, the fault action F_1 reaches the state predicate $reachedStates_1 \equiv ((Inhibit = 1) \Rightarrow (ActuatorStatus = 0)) \wedge (InitFail = 1)$ from the invariant $\mathcal{I}_{ASW}$. Likewise, the reachability nodes corresponding to F_2 and F_3 respectively send the following set of reached states to the reachability coordinator: $reachedStates_2 \equiv ((Inhibit = 1) \Rightarrow (ActuatorStatus = 0)) \wedge (AltFail = 1)$ and $reachedStates_3 \equiv ((Inhibit = 1) \Rightarrow (ActuatorStatus = 0)) \wedge (ActFail = 1)$. Thus, since program actions are closed in the invariant, the set of states reachable by fault and program actions in the first iteration of the repeat-until loop in Figure 3 is equal to $\mathcal{FS}_{ASW} = ((Inhibit = 1) \Rightarrow (ActuatorStatus = 0)) \wedge ((AltFail = 1) \vee (InitFail = 1) \vee (ActFail = 1))$. The set of states reachable from $\mathcal{FS}_{ASW}$ by fault and program actions would yield no new states outside $\mathcal{FS}_{ASW}$, thereby returning $\mathcal{FS}_{ASW}$ as the fault-span of the ASW program for fault actions F_1, F_2

and F_3. Since fault actions do not directly violate safety specification, the base set of states for backward reachability is empty. Therefore, the set of offending states is empty. **Removing the set of offending transitions.** We create five synthesis nodes corresponding to the actions A_1-A_5. In addition to an action A_i $(1 \leq i \leq 5)$, each synthesis node $Node_i$ takes the invariant $\mathcal{I}_{ASW}$, the fault-span $\mathcal{FS}_{ASW}$, the safety specification $\mathcal{S}_{ASW}$, the set of offending states OS and the write restrictions as its inputs. Since $OS = \emptyset$, the first two steps in Figure 4 result in an unsatisfiable transition predicate. Next, we describe the results of executing Steps 3 and 4 in each $Node_i$.

- **Action** A_1**.** Step 3 calculates the transition predicate $(Mode = I) \wedge (InitFail = 1) \wedge (Mode' = SB)$ as the set of transitions that violate the safety specification and should be excluded from action A_1. Therefore, action A_1 would be revised as follows:

$$(Mode = I) \wedge (\mathbf{InitFail = 0}) \qquad \longrightarrow \; Mode := SB;$$

- **Action** A_2**.** Since the safety specification $\mathcal{S}_{ASW}$ does not stipulate anything about changing the program mode to initialization, this action does not include any offending transitions. Therefore, A_2 remains unchanged.

- **Action** A_3**.** The safety specification $\mathcal{S}_{ASW}$ states that the ASW program must not enter the AA mode if the altitude sensors have failed (i.e., $AltFail = 1$). Thus, the synthesis node corresponding to action A_3 revises A_3 as follows:

$$(Mode = SB) \wedge (AltBelow = 1) \wedge (Inhibit = 0) \wedge (ActuatorStatus = 0) \wedge (\mathbf{AltFail = 0}) \longrightarrow \; Mode := AA;$$

- **Action** A_4**.** If faults cause the actuator to fail, then the program should not power on the actuator. Thus, due to the execution of the fault action F_3, action A_4 may be enabled in states where $ActFail = 1$ holds, thereby leading to the violation of safety. The elimination of such transitions originated in the fault-span outside the invariant results in the following revised action:

$$(Mode = AA) \wedge (ActuatorStatus = 0) \wedge (Inhibit = 0) \wedge (\mathbf{ActFail = 0}) \longrightarrow Mode := SB;\ ActuatorStatus := 1;$$

- **Action** A_5**.** This action remains unchanged for the same reason as that of action A_2.

Since the set of offending states is empty, no state is removed from the invariant. Therefore, the invariant of the failsafe program is equal to $\mathcal{I}_{ASW}$. We note that the failsafe ASW program synthesized in this section is similar to the ASW program that has been manually designed in [6].

7. SUMMARY OF CASE STUDIES

In this section, we discuss some of the results of our case studies. We would like to emphasize that our focus is on decomposing the problem of adding fault-tolerance. Thus instead of comparing the efficiency of different tools, techniques and data structures in the implementation of individual steps of our approach, we investigate the approximate speed up provided by our approach considering synthesis and communication times. Let ρ denote the ratio of the average time spent for communication with the coordinator over the average synthesis time spent in each synthesis node. We present the results of our experiments on the BG example as Byzantine faults are among the most difficult types of faults to tolerate.[5] For small programs (e.g., 3 to 9 non-generals), integrated algorithms outperform our approach as the average value of ρ is greater than or equal to 1, whereas for larger programs (10 to 21 non-generals) the synthesis time rises significantly, and accordingly ρ declines below 1. For instance, the largest program that we have so far synthesized on a cluster of two MS-Windows notebooks (each with 1 GB RAM and Pentium IV CPU) and one Linux PC (with 1 GB RAM and a Dual-Core CPU) is a failsafe BG program with 21 non-generals. In this case, the average synthesis time in each synthesis node is about 38 minutes while the average communication time in each synthesis node is about 13 minutes. We would like to mention that we could not synthesize the above program on either one of the above machines alone due to time/space complexity.

8. RELATED WORK

In this section, we discuss related work on existing divide-and-conquer synthesis methods [27, 26] and techniques for reducing the time/space complexity of automatic addition of fault-tolerance [19, 20, 21, 14, 7]. Specifically, Smith [27] presents a divide-and-conquer approach for synthesizing programs from their specifications, where he decomposes the program specification into the specifications of sub-problems and combines the results of synthesizing solutions for sub-problems. Puri and Gu [26] propose a divide-and-conquer synthesis method for asynchronous digital circuits, where they decompose circuit specifications and satisfy design constraints for each sub-specification. While the aforementioned approaches inspire our work, they are essentially specification-based approaches and do not directly focus on adding failsafe fault-tolerance to existing programs.

Previous work on automatic addition of fault-tolerance [19, 20, 21, 14, 7] mostly focuses on techniques for reducing time/space complexity of synthesis. For example, Kulkarni *et al.* [19] present a set of heuristics based on which they reduce the time complexity of adding fault-tolerance to integrated models of distributed programs. Kulkarni and Ebnenasir [20] present a technique for reusing the computations of an existing fault-tolerant program in order to enhance its level of tolerance. They also present a set of pre-synthesized fault-tolerance components [21] that can be reused during the addition of fault-tolerance to different programs. We have presented a SAT-based technique [14]

[5]Our implementation is in C++. Initially, we used Yices for the validation of our case studies, however, our recent experiments show that a combination of different data structures (e.g., decision diagrams) and tools (e.g., SAT solvers) would greatly enhance the efficiency of synthesis.

where we employ SAT solvers to solve some verification problems during the addition of fault-tolerance to *integrated* program models. Bonakdarpour and Kulkarni [7] present a symbolic implementation of the heuristics in [19] where they use BDDs to model distributed programs.

Our DiConic approach significantly differs from the previous work in that we present a new paradigm for decomposing the problem of adding failsafe fault-tolerance instead of adding failsafe fault-tolerance to an integrated model of the entire program. More specifically, time/space complexity issues are orthogonal to our DiConic approach since one can benefit from existing techniques to mitigate the complexity of revising individual program actions.

9. CONCLUSIONS AND FUTURE WORK

We presented a divide-and-conquer approach, called DiConic, for the addition of failsafe fault-tolerance to (distributed) programs, where in the presence of faults a failsafe program guarantees to satisfy at least its safety specification. We specifically focused on the following question: *Given a fault-intolerant program and a specific fault-type, how can we revise each program action separately so that the entire program becomes failsafe fault-tolerant?* To address this question, we decomposed the problem of adding failsafe fault-tolerance into sub-problems. In each sub-problem, we concentrated on one program action and determined how that action should be revised so that the entire program would be failsafe fault-tolerant. We validated our approach for the classic problem of Byzantine Generals and for a simplified version of an altitude switch controller.

As an extension of this work, we are investigating the DiConic addition of recovery to programs, where we revise program instructions in isolation so that the entire program would eventually recover to its invariant after faults stop occurring. Another extension of this work is to enhance the efficiency of the synthesis coordinator by developing its distributed version. DiConic addition of fault-tolerance is a special case of a more general problem in which new Temporal Logic [16] properties are incrementally added to an existing program (as defined and addressed in our previous work [15]). Towards this end, we plan to develop new DiConic algorithms for automatic addition of new properties to (multithreaded and distributed) programs.

10. REFERENCES

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

[2] 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.

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

[4] P. Attie and A. Emerson. Synthesis of concurrent programs for an atomic read/write model of computation. *ACM TOPLAS*, 23(2), March 2001.

[5] P. C. Attie, A. Arora, and E. A. Emerson. Synthesis of fault-tolerant concurrent programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(1):125 – 185, 2004.

[6] R. Bharadwaj and C. Heitmeyer. Developing high assurance avionics systems with the SCR requirements method. *In Proceedings of the 19th Digital Avionics Systems Conference, Philadelphia, PA*, October 2000.

[7] B. Bonakdarpour and S. S. Kulkarni. Exploiting symbolic techniques in automated synthesis of distributed programs. In *IEEE International Conference on Distributed Computing Systems*, pages 3–10, 2007.

[8] K. Cho and J. 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, April 1998.

[9] E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[10] B. Dutertre and L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *Proceedings of the 18th Computer-Aided Verification conference*, volume 4144 of *LNCS*, pages 81–94. Springer-Verlag, 2006.

[11] A. Ebnenasir. *Automatic Synthesis of Fault-Tolerance.* PhD thesis, Michigan State University, May 2005.

[12] A. Ebnenasir. Diconic addition of failsafe fault-tolerance. Technical Report CS-TR-07-03, Department of Computer Science, Michigan Technological University, Houghton, Michigan, June 2007.

[13] A. Ebnenasir and S. S. Kulkarni. FTSyn: A framework for automatic synthesis of fault-tolerance. `http://www.cs.mtu.edu/~aebnenas/research/tools/ftsyn.htm`.

[14] A. Ebnenasir and S. S. Kulkarni. SAT-based synthesis of fault-tolerance. *Fast Abstracts of the International Conference on Dependable Systems and Networks, Palazzo dei Congressi, Florence, Italy, June 28 - July 1*, 2004.

[15] A. Ebnenasir, S. S. Kulkarni, and B. Bonakdarpour. Revising UNITY programs: Possibilities and limitations. In *International Conference on Principles of Distributed Systems (OPODIS)*, pages 275–290, 2005.

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

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

[18] S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. In *Proceedings of the 6th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 82–93, 2000.

[19] S. S. Kulkarni, A. Arora, and A. Chippada. Polynomial time synthesis of Byzantine agreement. In *Symposium on Reliable Distributed Systems*, pages 130 – 139, 2001.

[20] S. S. Kulkarni and A. Ebnenasir. Enhancing the fault-tolerance of nonmasking programs. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 441–449, 2003.

[21] S. S. Kulkarni and A. Ebnenasir. Adding fault-tolerance using pre-synthesized components. *Fifth European Dependable Computing Conference (EDCC-5), LNCS, Vol. 3463, p. 72*, 2005.

[22] S. S. Kulkarni and A. Ebnenasir. Complexity issues in automated synthesis of failsafe fault-tolerance. *IEEE Transactions on Dependable and Secure Computing*, 2(3):201–215, 2005.

[23] S. Lafortune and F. Lin. On tolerable and desirable behaviors in supervisory control of discrete event systems. *Discrete Event Dynamic Systems: Theory and Applications*, 1(1):61–92, 1992.

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

[25] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *ACM Symposium on Principles of Programming Languages*, pages 179–190, Austin, Texas, 1989.

[26] R. Puri and J. Gu. A divide-and-conquer approach for asynchronous interface synthesis. In *ISSS '94: Proceedings of the 7th international symposium on High-level synthesis*, pages 118–125, 1994.

[27] D. Smith. A problem reduction approach to program synthesis. *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, pages 32–36, 1983.

[28] W. Thomas. On the synthesis of strategies in infinite games. In *STACS*, pages 1–13, 1995.

[29] N. Wallmeier, P. Hütten, and W. Thomas. Symbolic synthesis of finite-state controllers for request-response specifications. In *CIAA, LNCS, Vol. 2759*, pages 11–22, 2003.