Distributed Synthesis of Fault-Tolerant Programs in the High Atomicity Model ⋆

Borzoo Bonakdarpour, Sandeep S. Kulkarni, and Fuad Abujarad

Department of Computer Science and Engineering,
Michigan State University,
East Lansing, MI 48824, USA
Email: {borzoo, sandeep, abujarad}@cse.msu.edu
http://www.cse.msu.edu/˜{borzoo,sandeep,abujarad}

Abstract. In this paper, we concentrate on distributed algorithms for automated synthesis of fault-tolerant programs in the high atomicity model, where all processes can read and write all program variables in one atomic step. Although there has recently been an increasing interest in using parallel and distributed techniques in the model checking community, these technique have not been investigated in program synthesis. Developing such techniques is crucial as a means to cope with the state explosion problem in the context of program synthesis and transformation as well. We propose two distributed multithreaded algorithms for adding two levels of fault-tolerance, namely *failsafe* and *masking*, to existing fault-intolerant programs whose state space is distributed over a network or cluster of workstations.

Keywords: Program transformation, Program synthesis, Distributed algorithms, Fault-tolerance, Parallel synthesis.

1 Introduction

Automated program synthesis is the problem of designing an algorithmic method to find a program that satisfies a set of required behaviors. Such automated method is desirable, as it ensures that the synthesized program is *correct-by-construction*. Similar to verification algorithms, synthesis algorithms often suffer from two factors of time and space complexity. In order to overcome the time complexity problem, several approaches have been proposed in the literature to incrementally *add* properties to existing programs [1–7]. These approaches (called *local redesign*) make it possible to start from an existing program and, hence, *reusing* the previous efforts made for synthesizing them effectively. As opposed to local redesign, the traditional synthesis algorithms (called *comprehensive redesign*) [8,9] start from specification. Hence, for adding a newly identified property, one should synthesize a new program by starting from the conjunction of the new property and the existing properties from scratch.

In order to overcome the space explosion problem, recently, an increasing interest in parallel and distributed techniques has emerged in the model checking community

⋆ This work was partially sponsored by NSF CAREER CCR-0092724 and ONR Grant N00014-01-1-0744.

(e.g., [10–16]). Such techniques parallelize *enumerative* or *symbolic* state space of a given model over a network or cluster of workstations and run a distributed verification algorithm over the parallelized state space. On the other hand, the space explosion problem remains unaddressed in the context of automated program synthesis.

With this motivation, in this paper, we concentrate on the problem of designing distributed algorithms for automated program synthesis. More specifically, we parallelize two synthesis algorithms (from [7]) for adding two levels of fault-tolerance, namely failsafe and masking, to existing fault-intolerant programs. Intuitively, in the presence of faults, a *failsafe* fault-tolerant program satisfies only its safety specification, but a *masking* fault-tolerant program satisfies both its safety and liveness specifications. We assume that programs are in the *high atomicity model*, where all processes can read and write all program variables in one atomic step. We note that the aforementioned synthesis algorithms *solely* add fault-tolerance to a fault-intolerant program in the sense that they add no new behaviors to the input program in the *absence* of faults.

Similar to distributed model checking techniques, developing distributed synthesis algorithms consists of two phases: (1) parallelizing the state space over a network of workstations, and (2) designing a distributed algorithm that runs on each partition of the state space. In this paper, we only focus on the second phase. In particular, we assume that parallelization of state space is already done using one of the known enumerative techniques in the literature. Precisely, we use the state space parallelization technique proposed by Garavel, Mateescu, and Smarandache [10] with some modifications tailored for the purpose of synthesis rather than model checking. Although there exist more efficient ways for parallel construction of state space (e.g., using abstract interpretation), we cannot trivially apply them as a means for synthesizing programs. This is due to the fact that in synthesis (unlike model checking), we usually require full information about the program being synthesized, as we need to manipulate a program by removing or adding computations. Thus, we conservatively choose to develop distributed algorithms that run over the detailed parallelized enumerative state space.

Since the essence of the proposed algorithm in [7] for synthesizing failsafe fault-tolerant programs is calculating fixpoint of formulas, in this paper, we propose a distributed multithreaded algorithm for calculating smallest and largest fixpoints. Furthermore, since a masking fault-tolerant program recovers to its normal behavior after the occurrence of faults, we also propose a distributed algorithm for synthesizing recovery paths.

Contributions of the paper[1]. The main results of this paper are as follows. We propose (i) a distributed multithreaded synthesis algorithm for adding failsafe fault-tolerance, and (ii) a distributed multithreaded synthesis algorithm for adding masking fault-tolerance to existing fault-intolerant programs. These algorithms involve designing distributed techniques for fixpoint calculations and adding recovery computations to a program. To the best of our knowledge, this paper is the first work that addresses challenges and proposes solutions for designing distributed algorithms in the context

[1] In this paper, since we focus on challenges of designing distributed synthesis algorithms, we refer the reader to [1, 2, 5–7, 17] for examples and analysis of different aspects of addition of fault-tolerance to fault-intolerant programs.

of program synthesis and transformation. We believe that this study paves the way for further research on designing distributed synthesis algorithms.
Organization of the paper. In Section 2, we present the preliminary concepts. In Section 3, we formally state the problem of addition of fault-tolerance to existing fault-intolerant programs. Then, we present our distributed synthesis algorithms for adding failsafe and masking fault-tolerance in Section 4. Finally, we make the concluding remarks and discuss future work in Section 5.

2 Preliminaries

In this section, we present formal definitions of programs, specifications, faults, and fault-tolerance. We specify *programs* in terms of their state space and their transitions. The definition of *specifications* is adapted from Alpern and Schneider [18]. The definition of *faults* and *fault-tolerance* is adapted from Arora and Gouda [19].

2.1 Programs and Specifications

A *program* p is specified by a tuple $\langle S_p, \delta_p \rangle$, where S_p is the finite *state space* of p and δ_p is a finite set of *transitions* (i.e., a subset of $S_p \times S_p$). A sequence of states, $\sigma = \langle s_0, s_1, ... \rangle$, is a *computation* of p where $s_i \in S_p$ for all $i \in \mathbb{Z}_{\geq 0}$ iff the following two conditions are satisfied: (1) if σ is infinite then $\forall j > 0 : (s_{j-1}, s_j) \in \delta_p$, and (2) if σ is finite and terminates in state s_n then there does not exist state s such that $(s_n, s) \in \delta_p$, and the condition $\forall j \mid 0 < j \leq n : (s_{j-1}, s_j) \in \delta_p$ holds. A *computation prefix* of p is a finite sequence of states $\langle s_0, s_1, ..., s_k \rangle$, where k is a positive integer and $\forall j \mid 0 < j \leq k : (s_{j-1}, s_j) \in \delta_p$. Note that when it is clear from the context, we use p and δ_p interchangeably.

A *state predicate* of p is any subset of S_p. A state predicate S is *closed* in program p iff $\forall (s_0, s_1) \in p \; : \; (s_0 \in S \Rightarrow s_1 \in S)$. The *projection* of program p on a state predicate S (denoted $p \mid S$) consists of transitions $\{(s_0, s_1) \mid (s_0, s_1) \in p \wedge s_0, s_1 \in S\}$, i.e., transitions of p that start in S and end in S.

A *specification* Σ is a set of infinite sequences of states. Given a program p, a state predicate S, and a specification Σ, we say that p *satisfies* Σ *from* S (denoted $p \models_S \Sigma$) iff (1) S is closed in p, and (2) every computation of p that starts in a state where S is true is in Σ. If $p \models_S \Sigma$ and $S \neq \{\}$, we say that S is an *invariant* of p for Σ.

We say that a finite computation α *maintains* Σ iff there exists a computation suffix β such that $\alpha\beta$ is in Σ. We say that program p *maintains* (does not *violate*) Σ from S iff (1) S is closed in p, and (2) every computation prefix α of p that starts in a state in S maintains Σ. Note that the definition of *maintains* focuses on finite sequences of states, whereas the definition of *satisfies* characterizes infinite sequences of states.
Notation. Whenever the specification is clear from the context, we will omit it; thus, "S is an invariant of p" abbreviates "S is an invariant of p for Σ".

In this paper, we only consider suffix-closed and fusion-closed specifications. *Suffix closure* of a set means that if a state sequence σ is in that set then so are all the suffixes of σ. *Fusion closure* of a set means that if state sequences $\langle \alpha, s, \gamma \rangle$ and $\langle \beta, s, \delta \rangle$ are in that set then so are the state sequences $\langle \alpha, s, \delta \rangle$ and $\langle \beta, s, \gamma \rangle$, where α and β are finite prefixes of state sequences, γ and δ are suffixes of state sequences, and s is a program state. Intuitively, fusion closure of the specification means that an implementation of the specification must execute its next transition only based on its current state, i.e., the history of a computation does not affect the next move of the program.

Furthermore, following Alpern and Schneider [18], we let the specification be a conjunction of a *safety specification* and a *liveness specification*. For a suffix-closed and fusion-closed specification, the safety specification can be represented as a set Σ_{bt} of *bad transitions* [20] that must not occur in program computations (i.e., the safety specification is a subset of $S_p \times S_p$). Now, let Σ be a specification. Throughout the paper, we let $\Sigma_{\overline{bt}}$ be the specification whose computations, say $\sigma = \langle s_0, s_1, \cdots \rangle$, is in Σ and for all $i \geq 0$, $(s_i, s_{i+1}) \notin \Sigma_{bt}$, i.e., the specification in which safety is never violated. In our algorithms, we do not explicitly specify the liveness specification; the transformed fault-tolerant program satisfies the liveness specification iff the input fault-intolerant program satisfies the liveness specification.

2.2 Faults and Fault-Tolerance

The *faults* that a program p is subject to are systematically represented by a set f of transitions, i.e., a subset of $S_p \times S_p$ where S_p is the state space of p. A sequence of states $\langle s_0, s_1, ... \rangle$ is a *computation of* p *in the presence of* f iff (1) $\forall j > 0 \;:\; ((s_{j-1}, s_j) \in \delta_p \cup f)$, (2) if the sequence is finite and terminates in s_l then there exists no program transition originating at s_l, and (3) $\exists n \geq 0 : (\forall j > n : (s_{j-1}, s_j) \in \delta_p)$. We note that the last condition (bounded fault model) is only necessary for *masking* fault-tolerance (defined below) where recovery to the invariant is required. This constraint is not necessary for *failsafe* fault-tolerance.

We use $p[]f$ to denote the transitions obtained by taking the union of the transitions in p and the transitions in f. 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) $S \subseteq T$, 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.

We now describe what we mean by *levels of fault-tolerance*. We identify the fault-tolerance level of a program based on its behavior in the presence of faults. We say that p is *failsafe* f-tolerant to Σ_{bt} from S iff (i) $p \models_S \Sigma_{\overline{bt}}$, and (ii) there exists a state predicate T such that T is an f-span of p from S and $p[]f$ maintains $\Sigma_{\overline{bt}}$ from T. We say that p is *masking* f-tolerant to Σ_{bt} from S iff (i) $p \models_S \Sigma_{\overline{bt}}$, and (ii) there exists a state predicate T such that (1) T is an f-span of p from S, (2) $p[]f$ maintains $\Sigma_{\overline{bt}}$ from T, and (3) every computation of $p[]f$ that starts from a state in T has a state in S.

3 Problem Statement

In this section, we reiterate the problem statement from [7]. However, it is important to note that in this paper, we solve the same problem in a distributed fashion. Given are a program p with invariant S, a set of faults f, and safety specification Σ_{bt} such that $p \models_S \Sigma_{\overline{bt}}$. Our goal is to find a program p' with invariant S' such that p' is f-tolerant to Σ_{bt} from S'.

As mentioned in the introduction, our synthesis methods obtain p' from p by adding fault-tolerance alone to p, i.e., p does not introduce new behaviors to p when no faults have occurred. Observe that if S' (respectively, $p' \mid S'$) contains states (respectively, transitions) that are not in S (respectively, $p' \mid S$) then, in the absence of faults, p' may include computations that start outside S (respectively, $p' \mid S$). Since we require that $p' \models_{S'} \Sigma_{\overline{bt}}$, it would imply that p' is using a new way to satisfy $\Sigma_{\overline{bt}}$ in the absence of faults. Therefore, we require that $S' \subseteq S$ and $(p' \mid S') \subseteq (p \mid S')$. Thus, the synthesis problem is as follows (we instantiate this problem for failsafe and masking f-tolerance in the obvious way):

Problem Statement 3.1. Given p, S, f, and Σ_{bt} such that $p \models_S \Sigma_{\overline{bt}}$. Identify p' and S' such that:

$(C1)$ $S^{'} \subseteq S$,
$(C2)$ $(p' \mid S') \subseteq (p \mid S')$, and
$(C3)$ p' is f-tolerant to Σ_{bt} from S'. □

In [7], Kulkarni and Arora introduce centralized polynomial time (in the state space of p) sound and complete synthesis algorithms, where the program can atomically read and write all program variables. Based on these algorithms, in Section 4, we present distributed algorithms for synthesizing failsafe and masking fault-tolerant programs.

4 Distributed Automated Addition of Fault-Tolerance

In this section, we present our distributed algorithms for adding fault-tolerance to existing fault-intolerant programs. Similar to distributed model checking techniques, developing distributed synthesis algorithms consists of two phases: (1) parallelizing the state space over a network of workstations, and (2) designing a distributed algorithm that runs on each portion of the state space. In this paper, we only focus on the second phase. In particular, we assume that parallelization of state space is already done using the construction technique due to Garavel, Mateescu, and Smarandache [10]. However, we make some modifications tailored for the purpose of synthesis rather than model checking.

4.1 Parallel Construction of State Space

In order to represent a program p with state space S_p and invariant S on N machines (numbered from 0 to $N-1$), we use the notion of *partitioned* programs. More specifically, the state space S_p is partitioned to $S_p^0 \cdots S_p^{N-1}$, where $S_p = \cup_{i=0}^{N-1} S_p^i$ and $S_p^i \cap S_p^j = \{\}$ for all $0 \leq i \neq j < N$ (i.e., the state space is partitioned into N classes, one class per machine). Likewise, state predicates are partitioned in the same fashion. For instance, machine i contains S^i and T^i partitions of the invariant S and the fault-span T. From now on, we call S the *global invariant* and each S^i the *local invariant* with respect to machine i. The same concept applies to any other state predicate such as the fault-span T, i.e., T is the global fault-span and T^i is the local fault-span with respect to machine i.

The set p of transitions is partitioned to $p^0 \cdots p^{N-1}$, where $p = \cup_{i=0}^{N-1} p^i$, and $(s_0, s_1) \in p^i$ iff $(s_0 \in S_p^i \vee s_1 \in S_p^i)$ for all $0 \leq i < N$ (i.e., if the source and target of a transition belong to different machines, the transition is stored in both the source and target machines). We call such transitions *cross transitions*. Likewise, f and Σ_{bt} are partitioned in the same fashion. From now on, we call p the *global set of program transitions* and each p^i the *local set of program transitions* with respect to machine i. The same concept applies to any other set of transitions such as the set of faults f and the set of bad transitions Σ_{bt}.

Remark 4.1. We choose to store cross transitions in both source and target machines due to two reasons: (1) as we shall see in Subsections 4.2 and 4.3, such duplication decreases the number of potential broadcast messages considerably, and (2) it allows us to efficiently do both forward and backward reachability analysis at the same time. In fact, this deviation from distributed model checking techniques is due to the nature synthesis as compared to verification.

Assumption 4.2. In our synthesis algorithms, we assume that the input fault-intolerant program is already partitioned over a network using a reasonable static partition function $h : S_p \to [0, N-1]$ using the above parallelization method. In other words, machine i contains a state s iff $h(s) = i$. We also assume that all the synthesis processes over the network have a replica of h.

Revised problem statement. With this setting, we revise the Problem Statement 3.1 as follows. Given are a partition function h, a partitioned program $p^0 \cdots p^{N-1}$ with state space $S_p^0 \cdots S_p^{N-1}$, local invariants $S^0 \cdots S^{N-1}$, a partitioned class of faults $f^0 \cdots f^{N-1}$, and safety specification $\Sigma_{bt}^0 \cdots \Sigma_{bt}^{N-1}$ such that $p \models_S \Sigma_{\overline{bt}}$. Our goal is to design distributed algorithms that synthesize a program p' with invariant S' such that p' is failsafe/masking f-tolerant to Σ_{bt} from S'.

4.2 Distributed Addition of Failsafe Fault-Tolerance

In order to synthesize a failsafe fault-tolerant program, we transform p into p' such that transitions of Σ_{bt} occur in no computation prefixes of p'. Towards this end, we parallelize the proposed centralized algorithm in [7] for adding failsafe fault-tolerance.

Algorithm sketch. The essence of adding failsafe fault-tolerance consists of two parts: (1) a smallest fixpoint calculation for identifying the set of states from where safety may be violated, and (2) a largest fixpoint calculation for computing the invariant of the failsafe program. Our algorithm consists of a set of distributed processes each running on one machine across the network. Each process consists of two threads, namely, Distributed_Add_failsafe (cf. Figure 1) and MessageHandler (cf. Figure 2). Briefly, the thread Distributed_Add_failsafe is in charge of initiating local fixpoint calculations and managing synchronization points of the algorithm. The thread MessageHandler is responsible for handling messages sent by other synthesis processes across the network and invoking appropriate procedures. The thread Distributed_Add_failsafe consists of three main parts, namely, Lines 1-4 which is a smallest fixpoint computation, Lines 5-8 which is a largest fixpoint computation, and Lines 9-10 where we check the emptiness of the synthesized program (to declare failure or success). It also invokes three procedures, namely, FindLocalUnsafeStates, RemoveLocalDeadlocks, and EnsureClosure.

Assumption 4.3. Throughout the paper, we assume that procedure invocations are *atomic*.

thread Distributed_Add_failsafe(p^i, f^i, Σ^i_{bt} : set of transitions,
S^i_p, S^i : state predicate, N: **int**, h: partition function, $bLeader^i$: **Boolean**)
{
$cbSnt^i$, $cbRcvd^i := 0$; $ns^i := \{\}$; (1)
$ms^i := \{s_0 \mid \exists s_1 \in S^i_p : (s_0, s_1) \in f^i \ \wedge \ (s_0, s_1) \in \Sigma^i_{bt}\}$; (2)
$ms^i :=$ FindLocalUnsafeStates(S^i_p, ms^i, f^i); (3)
$\rightarrow$ BLKRECEIVE (`Trm_dtct`); $cbSnt^i$, $cbRcvd^i := 0$; (4)
$mt^i := \{(s_0, s_1) \mid s_1 \in (ms^i \cup ns^i) \ \vee \ (s_0, s_1) \in \Sigma^i_{bt}\}$; (5)
$S^i := S^i - ms^i$; $p^i := p^i - mt^i$; (6)
$S'^i, p'^i :=$ RemoveLocalDeadlocks(S^i, p^i); (7)
$\rightarrow$ BLKRECEIVE (`Trm_dtct`); (8)
if $(S'^i \neq \{\})$ **then return** $p' := \cup_{i=0}^{N-1} p'^i$, $S' := \cup_{i=0}^{N-1} S'^i$; (9)
elseif $(bLeader^i)$ **then** SEND($(i+1) \mod N$, $\texttt{Empt\_inv}(0)$); (10)
}
procedure FindLocalUnsafeStates(S^i_p, ms^i: state predicate, f^i: set of transitions)
// Returns the set of states from where safety may be violated by faults alone
{
while $(\exists s_0, s_1 : (s_1 \in ms^i \ \wedge \ (s_0, s_1) \in f^i))$ (11)
if $h(s_0) = i$ **then** $ms^i := ms^i \cup \{s_0\}$; (12)
else SEND($h(s_0)$, $\texttt{New\_ms}(s_0, s_1)$); $cbSnt^i := cbSnt^i + 1$; (13)
return ms^i; (14)
}
procedure RemoveLocalDeadlocks(S^i : state predicate, p^i : set of transitions)
// Returns the largest subset of S^i such that computations of p within that subset are infinite
{
while $(\exists s_1 \in S^i : (\forall s_2 \mid (\exists s_0 \mid (s_0, s_2) \in p^i) \ : \ (s_1, s_2) \notin p^i))$ (15)
$S^i := S^i - \{s_1\}$; (16)
$p^i :=$ EnsureClosure(p^i, S^i, s_1); (17)
return S^i, p^i (18)
}
procedure EnsureClosure(p^i: set of transitions, S^i: state predicate, s_1: state)
{
while $(\exists s_0 \ : \ ((s_0, s_1) \in p^i \ \wedge \ h(s_0) \neq i))$ (19)
SEND($h(s_0)$, $\texttt{New\_ds}(s_0, s_1)$); $cbSnt^i := cbSnt^i + 1$; (20)
$p^i := p^i - \{(s_0, s_1)\}$; (21)
return $p^i - \{(s_0, s_1) \mid s_0 \in S^i\}$ (22)
}

Fig. 1. Distributed algorithm for adding failsafe fault-tolerance.

We now describe our algorithm in detail. First, the thread Distributed_Add_failsafe finds the set ms^i of states from where a single fault transition violates the safety (Line 2). Next, we invoke the procedure FindLocalUnsafeStates where we find the set of states from where faults alone may violate the safety (Line 3). We find this set by calculating the smallest fixpoint of backward reachable states, given the initial set ms^i (Lines 11-12). In this calculation, if we find a fault transition, say (s_0, s_1), where $s_1 \in ms^i$, but s_0 resides in a machine other than i (i.e., $h(s_0) \neq i$), we send a `New_ms` message to process $h(s_0)$ indicating that s_0 is a state from where faults alone may violate the safety specification (Line 13).

Notation: At the receiver's side, we denote messages by $msg_j(params)$, where msg is the name of message, j is the sender process, and $params$ is a list of parameters sent along with the message. All messages (except `Trm_dtct`) are handled in the thread MessageHandler. At the sender's side, we omit the sender's subscript.

The receiver of a `New_ms` message (cf. Lines 1-2 in Figure 2) adds s_0 to its local ms^i (Line 1) and invokes the procedure FindLocalUnsafeStates (Line 2) so that by taking s_0 into account, new states from where faults alone may violate the safety specification are explored. The set ns^i consists of states that are in ms^j. Notice that every time a process sends (respectively, receives) such messages, it increments the variable $cbSnt^i$ (respectively, $cbRcvd^i$). We shall use these variables for termination detection as a means to synchronize processes at certain points.

The next phase of the algorithm is removing the states of global ms from the global invariant. To this end, we need to have a synchronization mechanism to ensure that calculation of ms^i is completed for all $i \in [0..N-1]$. In particular, we use the termination detection technique proposed by Mattern [21]. More specifically, in Line 4, the thread Distributed_Add_failsafe waits to receive a `Trm_dtct` message indicating that all processes are finished by calculating their local ms^i and all communication channels are empty. The arrows ($\rightarrow$) in Figure 1 mark the synchronization barriers. We will describe the termination detection technique later in this subsection.

After calculating the global set ms, we remove this set from the invariant to ensure that no computation of p' that starts from a state in S' violates the safety specification. We also remove the transitions of the set mt^i from p^i, where mt^i consists of transitions whose target states are in ms^i or directly violate the safety specification (Line 6). Notice that this removal may create *deadlock* states (i.e., states from where there exist no outgoing transitions). Thus, the thread Distributed_Add_failsafe invokes the procedure RemoveLocalDeadlocks (Line 7) to remove deadlock states which is in turn calculating the largest fixpoint of backward reachable states, given the initial set S^i. In other words, it keeps removing deadlock states until it reaches a fixpoint (Lines 15-16). In this calculation, since removal of a deadlock state, say s_1, may create transitions, say (s_0, s_1), such that (s_0, s_1) violates the closure of invariant, we invoke the procedure EnsureClosure (Line 17) to ensure that no such transitions exist in the final synthesized program. Furthermore, if we encounter a program transition, say (s_0, s_1), where s_1 is a deadlock state and s_0 resides in a machine other than i (i.e., $h(s_0) \neq i$), then we send a `New_ds` message to process $h(s_0)$ indicating that s_0 *might* be a deadlock state (Line 20). Upon receipt of such a message (cf. Line 3 in Figure 2), the receiver removes the transition (s_0, s_1) to maintain consistency of transitions and then invokes the procedure RemoveLocalDeadlocks (cf. Line 4 in Figure 2) to remove possible new deadlock states due to removal of (s_0, s_1). Similar to the calculation of ms, our algorithm ensures completion of calculation of the largest fixpoint S' using the same termination detection technique (cf. Line 8 in Figure 1).

At this point, each process has synthesized a local set of program transitions p'^i with a local invariant S'^i. The union of these portions is the final synthesized program, i.e., $p' = \cup_{i=0}^{N-1} p'^i$ and $S' = \cup_{i=0}^{N-1} S'^i$. However, since invariant predicates cannot be empty, if S' turns out to be equal to the empty set, the algorithm declares failure. To test the emptiness of S', a pre-specified *leader* process identified by the variable *bLeader* initiates an emptiness polling of the global invariant S' as follows. For this polling (and also termination detection), we consider a unidirectional virtual ring which connects every machine i to its successor machine $(i+1) \mod N$. Note that this virtual ring is independent of the fully connected topology of the network. Now, if the local invariant of the leader is equal to the empty set then it sends an `Empt_inv`(0) message to its first neighbor on the virtual ring (process $(i+1) \mod N$) indicating that its own local invariant is equal to the empty set (cf. Line 10 in Figure 1). If the local invariant of the

$((i+1) \mod N)^{\text{th}}$ process is equal to the empty set as well, it increments the value of k (the integer received along with the message `Empt_inv`) by one and sends the same message to the next process on the ring (cf. Line 5 in Figure 2). Otherwise, it does not change the value of k and sends an $\texttt{Empt\_inv}(k)$ message to the next process (Line 6). Upon the completion of one round of sending the `Empt_inv` messages, the leader finally finds out whether the global invariant S' is equal to the empty set or not (Lines 8-10). If the global invariant S' is indeed equal to the empty set then the leader declares failure (Line 8). Otherwise, it calculates and returns p' and S' (Line 10). Notice that Lines 9 and 10 in Figures 1 and 2 respectively *describes* that the output of the distributed algorithm is indeed a program which is the union of all local sets of transitions and local invariants.

Termination detection. In order to detect the termination of the fixpoint calculations, we use a virtual ring-based algorithm inspired by Mattern [21]. According to the general definition, global termination is reached when all local computations are complete (i.e., each machine i has calculated a local fixpoint) and all communication channels are empty (i.e., all sent transitions have been received). The core of the termination detection algorithm is as follows. Every time the leader process finishes its local fixpoint calculations, it checks whether global termination has been reached by generating two successive waves of `Report_rcv` (respectively, `Report_snd`) messages on the virtual ring to collect the number of messages received (respectively, sent) by all machines. A message $\texttt{Report\_rcv}_j(k)$ (respectively, $\texttt{Report\_snd}_j(k)$) received by machine i indicates that k messages have been received (respectively, sent) by the machines from the leader up to $j = (i-1) \mod N$. Each machine i counts the messages it has received and sent using two integer variables $cbRcvd^i$ and $cbSnt^i$, and adds their values to the numbers carried by `Report_rcv` and `Report_snd` messages (Lines 11, 13, and 14). Upon receipt of the $\texttt{Report\_snd}_j(k)$ message ending the second wave, the leader machine checks whether the total number k of messages sent is equal to the total number $nbTotal$ of messages received (the result of the `Report_rcv` wave). If this is the case, it informs the other machines that termination has been reached, by sending a broadcast `Trm_dtct` message. Otherwise, the leader concludes that termination has not been reached yet and will generate a new termination detection wave later (Line 15).

Theorem 4.4. The algorithm Distributed_Add_failsafe is sound and complete.
Proof. Since the output of our algorithm is identical to the output of the centralized algorithm by Kulkarni and Arora [7], the proof of soundness and completeness immediately follows. □

Performance of parallelized addition of failsafe. Distributing the synthesis algorithm is aimed at reducing the space complexity and time complexity. Of these, similar to the goals for distributed model checking, reducing the space complexity is a higher priority. We expect that our approach would assist in this case. In particular, if N machines are used to perform synthesis then each of them is expected to have $(1/N)^{\text{th}}$ number of states and at most $(2/N)^{\text{th}}$ number of transitions (because a transition may be stored in up to two machines). Regarding time complexity, in each phase, a machine performs some local computation that results a set of queries (e.g., `New_ms`, `New_ds`, etc.) for other machines. Now, consider the role of the two threads Dis-

thread MessageHandler()
{
  $msg :=$ RECEIVE();
  case msg **is**
    $\texttt{New\_ms}_j(s_0, s_1)$: $ms^i := ms^i \cup \{s_0\}$; $ns^i := ns^i \cup \{s_1\}$; $cbRcvd^i := cbRcvd^i + 1$; (1)
      $ms^i :=$ FindLocalUnsafeStates(S_p^i, ms^i, f^i); (2)
    $\texttt{New\_ds}_j(s_0, s_1)$: $p^i := p^i - \{(s_0, s_1)\}$; $cbRcvd^i := cbRcvd^i + 1$; (3)
      $S'^i, p'^i :=$ RemoveLocalDeadlocks(S^i, p^i); (4)
    $\texttt{Empt\_inv}_j(k)$: **if** $(\neg bLeader \ \wedge \ S^i = \{\})$ **then**
      SEND$((i+1) \mod N, \texttt{Empt_inv}(k+1))$; **return** $\{\}$; (5)
      elseif $(\neg bLeader \ \wedge \ S^i \neq \{\})$ **then**
        SEND$((i+1) \mod N, \texttt{Empt_inv}(k))$; (6)
        return p'^i, S'^i; (7)
      if $(bLeader \ \wedge \ (k = N-1))$ **then**
        `declare no failsafe program` p' `exists`; (8)
        exit; (9)
      else return $p' := \cup_{i=0}^{N-1} p'^i$, $S' := \cup_{i=0}^{N-1} S'^i$; (10)
    $\texttt{Report\_rcv}_j(k)$: **if** $(\neg bLeader^i)$ **then**
        SEND$((i+1) \mod N, \texttt{Report_rcv}(k + cbRcvd^i))$; (11)
      else $nbTotal := k$; (12)
        SEND$((i+1) \mod N, \texttt{Report_snd}(cbSnt)^i)$; (13)
    $\texttt{Report\_snd}_j(k)$: **if** $(\neg bLeader^i)$ **then**
        SEND$((i+1) \mod N, \texttt{Report_snd}(k + cbSnt^i))$; (14)
      elseif $(nbTotal = k)$ **then**
        SEND$([(i+1) \mod N..(i+N-1) \mod N], \texttt{Trm_dtct})$; (15)
    $\texttt{New\_fs}(s_0)$: $T_1^i := T_1^i - \{s_0\}$; $cbRcvd^i := cbRcvd^i + 1$; (16)
      $T_1^i :=$ConstructLocalFaultSpan$(T_1^i, T_2^i - T_1^i, f^i)$; (17)
    $\texttt{Search\_path}_j(X)$: $r^i := \{\}$; $cbRcvd^i := cbRcvd^i + 1$; (18)
      For each $s_0 \in X$:
        if $(\exists s_1 \ : \ (\mathrm{Rank}(s_1) \neq \infty)) \wedge$
          $(s_0, s_1) \notin mt^i)$ **then**
            $r^i := r^i \cup \{(s_0, s_1, \mathrm{Rank}(s_1) + 1)\}$; (19)
      SEND$(j, \texttt{New_path}(r^i))$; $cbSnt^i := cbSnt^i + 1$; (20)
    $\texttt{New\_path}_j(r^i)$: $q^i := \{\}$; $cbRcvd^i := cbRcvd^i + 1$; (21)
      For each (s_0, s_1, a) s.t. $((s_0, s_1, a) \in r^i \ \wedge \ s_0 \in (T_2^i - T_1^i)$:
        if $(s_0, s_1) \notin p^i$ **then**
          $p^i := p^i \cup (s_0, s_1)$; $q^i := q^i \cup (s_0, s_1)$; (22)
          $\mathrm{Rank}(s_0) := a$; (23)
          $T_1^i, T_2^i := T_1^i \cup \{s_0\}, T_2^i - \{s_0\}$; (24)
          $p_1^i, T_1^i :=$ ConstructLocalRecoveryPaths$(S_1^i, T_2^i, p_1^i, mt^i)$; (25)
          SEND$(j, \texttt{Confirm_trns}(q^i))$; $cbSnt^i := cbSnt^i + 1$; (26)
    $\texttt{Confirm\_trns}_j(q^i)$: $p^i := p^i \cup q^i$; $cbRcvd^i := cbRcvd^i + 1$; (27)
      SEND$(j, \texttt{Commit})$; $cbSnt^i := cbSnt^i + 1$; (28)
    $\texttt{Commit}_j$: $cbRcvd^i := cbRcvd^i + 1$; (29)
      Wait to receive `Commit` message from all providers; (30)
      SEND$(i + 1 \mod N, \texttt{Token})$; $cbSnt^i := cbSnt^i + 1$; (31)
    $\texttt{Token}_j$: $cbRcvd^i := cbRcvd^i + 1$; (32)
      SEND$([(i+1) \mod N..(i+N-1) \mod N]$,
        $\texttt{Search\_path}(T_2^i - T_1^i))$; $cbSnt^i := cbSnt^i + 1$; (33)
}

Fig. 2. The message handler thread.

tributed_Add_failsafe and MessageHandler. The thread MessageHandler provides a new list of tasks (received from other machines) that should be performed by Distributed_Add_failsafe. Since Distributed_Add_failsafe begins with a list of tasks (based on its local states and transitions) and MessageHandler continues to provide new tasks based on requests received from others, we expect that the list of tasks that Distributed_Add_failsafe needs to perform will typically be nonempty at all times. In other words, communication cost will not be in the critical path for the synthesis. Therefore, we expect that the distributed synthesis algorithm will be able to provide significant benefits regarding time complexity as well.

4.3 Distributed Addition of Masking Fault-Tolerance

In order to synthesize a masking program, we should generate a program p' with invariant S' and fault-span T', such that p' never violates its safety specification and if faults perturb the state of p' to a state in T', it recovers to S' within a finite number of recovery steps. Similar to the distributed algorithm for adding failsafe fault-tolerance, our algorithm for adding masking fault-tolerance consists of two threads Distributed_Add_masking (cf. Figure 3) and MessageHandler (cf. Figure 2).

Our first estimate of a masking program is a failsafe program. Hence, we let our first estimate S_1^i be the local invariant of its failsafe fault-tolerant program (cf. Line 2 in Figure 3). Likewise, we estimate the local fault-span to be T_1^i where T_1^i includes all the states in the local state space minus the states from where safety of p' may be violated (Line 3). Next, we compute the local set of transitions p_1^i, local fault-span T_1^i, and local invariant S_1^i in a loop (Lines 5-15). This loop consists of three main steps for constructing recovery paths, calculating fault-span, and calculating invariant as follows:

1. In order to compute the local set of transitions p_1^i, we construct recovery paths from each state in the fault-span to a state in the invariant. To this end, we identify two types of recovery paths: (1) recovery paths consist of only local program transitions, and (2) recovery paths consist of both local program transitions as well as cross transitions. We note that since these transitions originate outside the invariant, they do not violate the second constraint of the problem statement (i.e., in the absence of faults, no new computation is introduced to fault-tolerant program).
   Recovery paths through local transitions. The thread Distributed_Add_masking invokes the procedure ConstructLocalRecoveryPaths (Line 6), which identifies layers of states in the local fault-span corresponding to the number of steps of recovery paths, in a loop (Lines 21-25). In the beginning of the loop it assigns a rank to each state which is equal to the number of recovery steps from that state to a state in the local invariant. In this setting, the rank of states in the local invariant are zero. In the first iteration of the loop, we identify the set of states from where one-step recovery to the local invariant is possible while maintaining the safety, i.e., $X_1^i = \{s_0 \mid s_0 \in (T_1^i - S_1^i) \ \wedge \ \exists s_1 \in S_1^i \ : \ (s_0, s_1) \notin mt^i\}$. Thus, we add the transitions, say (s_0, s_1) where $s_0 \in X_1^i$ and $s_1 \in S^i$, to the set of local program transitions. In the second iteration of the loop, we identify the set of states from where two-step recovery is possible. Indeed, this is equivalent to identifying the set of states from where one-step recovery is possible from $T_1^i - X_1^i$ to the set $X_1^i \cup S_1^i$. Continuing thus inductively, we identify layers of states from where multi-step recovery is possible. Finally, we reach a point where we identify the set X_1^i of states from where recovery to the local invariant using local transitions is possible and the set $T_1^i - X_1^i$ of states from where such recovery is not possible.
   Recovery paths through cross transitions. After constructing local recovery paths, the leader process initiates a wave of communication among all processes

thread Distributed_Add_masking(p^i, f^i, Σ^i_{bt} : set of transitions,
S^i_p, S^i : state predicate, N: **int**, h: partition function, $bLeader^i$: **Boolean**)
{
Compute ms^i and mt^i as in Distributed_Add_failsafe; (1)
Let S^i_1 be the local invariant of the failsafe version of p; (2)
$T^i_1 := S^i_p - ms^i; \forall s \in (T^i_1 - S^i_1) \ : \ \mathrm{Rank}(s) := \infty$; (3)
$p^i_1 := p^i$; (4)
repeat
$T^i_2, S^i_2 := T^i_1, S^i_1$; $cbSnt^i$, $cbRcvd^i := 0$ (5)
$p^i_1, T^i_1 :=$ ConstructLocalRecoveryPaths(S^i_1, T^i_1, p^i, mt^i); (6)
$\rightarrow$ BLKRECEIVE (Trm_dtct); $cbSnt^i$, $cbRcvd^i := 0$; (7)
if ($bLeader$) **then**
SEND($[(i+1) \mod N..(i+N-1) \mod N]$, Search_path($T^i_2 - T^i_1$)); (8)
$cbSnt^i := cbSnt^i + 1$; (9)
$\rightarrow$ BLKRECEIVE (Trm_dtct); $cbSnt^i$, $cbRcvd^i := 0$; (10)
$T^i_1 :=$ ConstructLocalFaultSpan($T^i_1, T^i_2 - T^i_1, f^i$); (11)
$\rightarrow$ BLKRECEIVE (Trm_dtct); $cbSnt^i$, $cbRcvd^i := 0$; (12)
$S^i_1 :=$ RemoveLocalDeadlocks($S^i_1 \cap T^i_1, p^i_1$); (13)
$\rightarrow$ BLKRECEIVE (Trm_dtct); (14)
if ($S^i_1 = \{\} \ \vee \ T^i_1 = \{\}$) **then break**; (15)
until ($T^i_1 = T^i_2 \ \wedge \ S^i_1 = S^i_2$)
$\rightarrow$ BLKRECEIVE (Trm_dtct); (16)
$T'^i, S'^i := T^i_1, S'^i$; (17)
if ($bLeader^i \ \wedge \ (S'^i = \{\})$) **then**
SEND($(i+1) \mod N$, Empt_inv(0)); (18)
if ($bLeader^i \ \wedge \ (T'^i = \{\})$) **then**
SEND($(i+1) \mod N$, Empt_fs(0)); (19)
}
procedure ConstructLocalRecoveryPaths(S^i, T^i : state predicate, p^i, mt^i: set of transitions)
$X^i_1 := S^i; j = 0; X^i_2 := \{\}$; (20)
repeat
$\forall s \in (X^i_1 - X^i_2) \ : \ \mathrm{Rank}(s) := j$; (21)
$X^i_2 := X^i_1; j := j + 1$; (22)
$r^i := \{(s_0, s_1) \mid s_0 \in (T^i_1 - X^i_1) \ \wedge \ s_1 \in X^i_1\} - mt^i$; (23)
$p^i := p^i \mid S^i \cup r^i$; (24)
$X^i_1 := X^i_1 \cup \{s_0 \mid \exists s_1 \ : \ (s_0, s_1) \in r_i\}$; (25)
until ($X^i_1 = X^i_2$)
return p^i, X^i; (26)
}
procedure ConstructLocalFaultSpan(T^i_1, T^i_2 : state predicate, f^i : set of transitions)
// Returns the largest subset of T^i_1 that is closed in f
{
while ($\exists s_0, s_1 \ : \ ((s_0 \in T^i_1) \ \wedge \ (s_1 \in T^i_2) \ \wedge \ (s_0, s_1) \in f^i)$)
$T^i_1 := T^i_1 - \{s_0\}$; (27)
For each $s_1 \in T^i_2$:
if ($\exists s_0 \ : \ ((s_0, s_1) \in f^i \ \wedge \ h(s_0) \neq i)$)
SEND($h(s_0)$, New_fs(s_0)); $cbSnt^i := cbSnt^i + 1$; (28)
return T^i_1; (29)
}

Fig. 3. Distributed algorithm for adding masking tolerance.

to identify the set of states from where local recovery is not possible, but recovery through cross transitions is possible. More specifically, the leader process sends a Search_path message to all other processes (Line 8 in Figure 3). Let us call the process which sends a Search_path the *requester* process. Upon receipt of this message along with the set X of states from where local recovery is not possible (Line 18 in Figure 2), each process offers a recovery cross transition, say (s_0, s_1), provided $(s_0, s_1) \notin mt^i$ and there exists a recovery path from s_1 (i.e., $\mathrm{Rank}(s_1) \neq$

∞), for each state $s_0 \in X$ (Line 19). Let us call such processes the *providers*. Each provider sends a `New_path` message carrying the set r^i of cross recovery transitions along with the rank of state s_0 to the requester (Line 20). Obviously, if the requester accepts the provider's transitions, the rank of s_0 will be Rank$(s_1) + 1$. Upon receipt of this message (Line 21 in Figure 2), the requester adds the new recovery cross transitions, say (s_0, s_1), to its set of local program transitions (Line 22) and sets the rank of source states s_0 (Line 23). These states should be added to the local fault-span (Line 24). Next, it invokes the procedure ConstructLocalRecoveryPaths to add new possible local recovery transitions by taking the newly added recovery cross transitions into account (Line 25). Then, it sends a `Confirm_trns` message to the providers of the cross transitions so that the set of cross transitions of providers and the requester processes are consistent (Line 26). Obviously, if the requester receives other offers for a cross transition originated at s_0, say (s_0, s_1) with rank a, where the current rank of s_0 is greater than a, then the requester can replace its current cross transition with (s_0, s_1). However, we do not illustrate such implementation details in the algorithms.

Finally, upon the receipt of a `Confirm_trns` message, the providers add the set q^i of cross transitions (selected by the requester) to their set of local program transitions as well (Line 27). At this point, providers send a `Commit` message to the requester (Line 28) indicating that the changes are committed. Upon receipt of `Commit` message from all providers (Lines 29-30), the requester sends a `Token` message to the next process on the virtual ring (Line 31) so that it starts identifying the cross recovery transitions in the same fashion (Line 33). We continue doing this until no cross transition is added across the network.

Notice that in both types of recovery paths, we do not introduce cycles to the fault-span, as we do not add transitions from a state with a lower rank to a state with higher rank. Hence, after occurrence of faults, recovery within a finite number of steps is guaranteed. We synchronize the completion of construction of recovery paths in Line 7.

2. Since there may exist states from where recovery to the invariant is not possible, we need to recompute the local fault-span by removing the states from where closure of fault-span is violated through fault transitions. To this end, we invoke the procedure ConstructFaultspan which is a largest fixpoint calculation (Line 11 in Figure 3) to calculate the largest fault-span which is closed in $p[]f$. Since this removal may cause other states in the local fault-span of other processes to violate the closure of the global fault-span, we send a `New_fs` message to such processes to indicate this fact (Line 28). Note that in order to synchronize the completion of calculation of local fault-spans, here as well, we need a barrier synchronization (Line 12).
3. Due to the removal of some states in step 2, we recompute the local invariant by invoking the procedure RemoveLocalDeadlocks. Notice that since S_1^i must be a subset of T_1^i, this invocation is parameterized by $S_1^i \cap T_1^i$ (Line 13). At this point, if both S_1^i and T_1^i are nonempty, we jump back to step 1 and we keep repeating the loop until a fixpoint is reached, i.e., $(T_1^i = T_2^i \wedge S_1^i = S_2^i)$.

Upon the termination of the repeat-until loop, recovery without violation of the safety specification from T_1' to S_1' is provided. At this point, if there exist processes

i and j such that S'^i and T'^j are both nonempty then we have a solution to the synthesis problem. Thus, similar to addition of failsafe, we run an emptiness poll among the processes (Lines 18-21). To this end, we send a `Empt_fs`(0), which is similar to `Empt_inv`, except the message handler tests the emptiness of the local fault-span rather than local invariant. We skip including this message in Figure 2.

We note that, in this paper, we have modified the recovery mechanism of the centralized algorithm in [7]. This is due to the fact that in that algorithm the authors add all possible transitions, i.e., the set $p_1|S_1 \cup \{(s_0, s_1) \mid s_0 \in T_1 - S_1 \ \wedge \ s_1 \in T_1\}$, and then remove non-progress cycles. However, since the size of this set in worst case is in the square order of the size of the state space, it implies that in worst case, each machine i must store a set whose size is in the square order of the state space which obviously does not make sense. Hence, instead of adding all possible transitions and removing cycles, we construct recovery paths in a more space-efficient way in a stepwise manner using the notion of layered fault-span (cf. the Procedure ConstructLocalRecoveryPaths).

Theorem 4.5. The algorithm Distributed_Add_masking is sound. □

5 Conclusion and Future Work

In this paper, we focused on the problem of automated addition of fault-tolerance to existing fault-intolerant programs where the state space of the fault-intolerant program is distributed over a network or cluster of workstations. We addressed this problem in the high atomicity model where all processes of the program are able to read and write all program variables in one atomic step. We presented two distributed multithreaded algorithms for adding failsafe and masking fault-tolerance to a given fault-intolerant program. To this end, we parallelized calculation of smallest and largest fixpoints of a given formula and also addition of safe recovery paths.

As future work, we plan to implement the algorithms proposed in this paper in our tool FTSyn. This implementation will enable us to synthesize fault-tolerant programs with large state space. We also plan to study the problem of designing distributed algorithms for adding fault-tolerance to distributed [6] and real-time [2] programs.

We are currently investigating the possibility of reducing the number of synchronization points in our algorithms. Such synchronization barriers (implemented by termination detection mechanisms) decrease the level of parallelism and, hence, efficiency of distributed algorithms. This study seems to be more crucial for distributed addition of masking fault-tolerance, as it needs four synchronization points in each iteration of its main loop. There are also open questions on how to reduce the number of cross transitions and, hence, message passing overhead through designing static partition functions tailored for synthesis purposes rather than reusing the ones designed for model checking techniques.

References

1. B. Bonakdarpour and S. S. Kulkarni. Exploiting symbolic techniques in automated synthesis of distributed programs with large state space. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 3–10, 2007.

2. B. Bonakdarpour and S. S. Kulkarni. Incremental synthesis of fault-tolerant real-time programs. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 122–136, 2006.
3. B. Bonakdarpour and S. S. Kulkarni. Automated incremental synthesis of timed automata. In *International Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, pages 261–276, 2006.
4. 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.
5. S. S. Kulkarni and A. Ebnenasir. Automated synthesis of multitolerance. In *International Conference on Dependable Systems and Networks (DSN)*, pages 209–219, 2004.
6. S. S. Kulkarni, A. Arora, and A. Chippada. Polynomial time synthesis of Byzantine agreement. In *Symposium on Reliable Distributed Systems (SRDS)*, pages 130–140, 2001.
7. S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. In *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*, pages 82–93, 2000.
8. 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.
9. Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(1):68–93, 1984.
10. H. Garavel, R. Mateescu, and I. Smarandache. Parallel state space construction for model-checking. In *8th International SPIN Workshop on Model Checking of Software*, pages 217–234, 2001.
11. U. Stern and D. L. Dill. Parallelizing the murφ verifier. In *Computer Aided Verification (CAV)*, pages 256–278, 1997.
12. T. Heyman, D. Geist, O. Grumberg, and A. Schuster. Achieving scalability in parallel reachability analysis of very large circuits. In *Computer-Aided Verification (CAV)*, pages 20–35, 2000.
13. M. Leucker, R. Somla, and M. Weber. Parallel model checking for LTL, CTL*, and L_2^μ. In *International Workshop on Parallel and Distributed Model Checking (PDMC)*, 2003.
14. R. Kumar and E. G. Mercer. Load balancing parallel explicit state model checking. In *Parallel and Distributed Model Checking (PDMC)*, 2004.
15. M.-Y. Chung and G. Ciardo. A dynamic firing speculation to speedup distributed symbolic state-space generation. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.
16. J. Barnat, L. Brim, I. Černá, and P. Šimeček. DiVinE the distributed verification environment. In M. Leucker and J. van de Pol, editors, *International Workshop on Parallel and Distributed Methods in verifiCation (PDMC)*, Lisbon, Portuga, 2005.
17. S. S. Kulkarni, B. Bonakdarpour, and A. Ebnenasir. Mechanical verification of automatic synthesis of fault-tolerant programs. In *International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR)*, pages 36–52, 2004.
18. B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.
19. 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.
20. S. S. Kulkarni. *Component-based design of fault-tolerance*. PhD thesis, Ohio State University, 1999.
21. F. Mattern. Algorithms for distributed termination detection. *Journal of Distributed Computing*, 2(3):161–175, 1987.