Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 258 (2009) 3–15
www.elsevier.com/locate/entcs

Weakest Invariant Generation for Automated Addition of Fault-Tolerance

Fuad Abujarad [1,2]

Department of Computer Science and Engineering
Michigan State University
East Lansing, USA

Sandeep S. Kulkarni[3]

Department of Computer Science and Engineering
Michigan State University
East Lansing, USA

Abstract

One of the important tasks in evolving a fault-intolerant program into a fault-tolerant one is to identify the legitimate states (its invariant) from where the original program satisfies its specification. This allows us to ensure that the fault-tolerant program recovers to these legitimate states from where it satisfies its specification. It is desired that the invariant be the weakest possible so as to provide maximal options to the algorithm for adding fault-tolerance. Requiring the designer to specify such weak invariant increases the burden on the designer. In this paper, we present a novel approach for automating the generation of the weakest invariant from the program actions and specifications. Our algorithm is efficient and manages the state explosion problem with the use of BDDs. We demonstrate our approach through two case studies and we show that generating such invariants is extremely fast. For example, for a mutual exclusion program with a state space of size 10^{84} states, it took 0.9 of a second.

Keywords: Invariant Generation, Fault-Tolerance

1 Introduction

The problem in this paper is motivated by the need for revising existing program and/or models to deal with new types of faults. Since the set of all faults that a program may be subject to is often unknown during the initial design, existing program and/or models need to be revised to deal with new faults. One requirement for such revision is that the existing program requirements continue to be satisfied.

[1] This work was partially sponsored by NSF CAREER CCR-0092724 and ONR Grant N00014-01-1-0744.
[2] Email: abujarad@cse.msu.edu
[3] Email: sandeep@cse.msu.edu

doi:10.1016/j.entcs.2009.12.010

An approach to gain assurance in such model revision is by *incremental synthesis*, which guarantees that the synthesized program is correct-by-construction.

One approach for providing such fault-tolerance is to ensure that after the occurrence of faults, the revised program eventually recovers to the legitimate states of the original program. Since the original program met its original specification from these legitimate states, we can ascertain that eventually a revised program reaches states from where subsequent computation is correct. This allows us to capture two levels of fault-tolerance: *masking* and *nonmasking*. In the former, safety requirements are preserved during recovery, whereas in the latter they may not be preserved.

One of the problems in providing recovery to legitimate states, however, is that these legitimate states are not always easy to determine. Existing synthesis approaches (e.g., [5]) have required the designer to specify these legitimate states explicitly. It is straightforward to observe that if these legitimate states could be derived automatically, then it would reduce the burden put on the designer, thereby making it easier to apply these techniques in revision of existing programs.

One method for identifying legitimate states is to use initial states as legitimate states. While identifying these initial states is easy for the designer, this approach is very limiting. A variation of this approach is to define the legitimate states to be those states that are reachable from the initial states. While less limiting, this approach fails to identify all states from where the existing program is correct, although such states are not reached in the fault-free execution. Although the knowledge of these states is irrelevant for fault-free execution, it is potentially useful in adding fault-tolerance. In particular, if faults perturb the program to one of these states, no recovery may be needed. Furthermore, recovery could be added to these states so that subsequent computation is correct. Based on this motivation, in this paper we focus on identifying the largest set of states from where the existing program is correct. We use the term *invariant* to denote the legitimate states of the program. (If we view a set of states as a predicate that is true only in those states then this corresponds to the weakest state predicate.) Of course, an enumerative approach, where we consider each state as a potential initial state, is impractical. Our goal in this paper is to identify efficient techniques for identifying the largest set of legitimate states for a given program.

Our algorithm for computing the largest set of legitimate states takes two inputs: the program (specified in terms of its transitions) and its specification. The program specifications consists of: (1) a safety specification, which is specified in terms of (bad) states that the program should not reach and (bad) transitions that the program should not execute, and (2) zero or more liveness specifications of the form $\mathcal{F}$ *leads to* $\mathcal{T}$ (written as $\mathcal{F} \rightsquigarrow \mathcal{T}$), which states that if the program ever reaches a state where $\mathcal{F}$ is true then in its subsequent computation it reaches a state where $\mathcal{T}$ is true.

Contributions of the paper.

- We show that our algorithms for finding the largest set of legitimate states is sound.
- With a BDD based implementation, we show that our algorithm manages the state explosion problem.
- We illustrate our algorithm in the context of two case studies: token ring, and tree based mutual exclusion program [18]. The invariants computed in these examples are identical to those in [13,5]. In particular, the set of legitimate states computed in this paper for mutual exclusion is used in [1] for adding nonmasking fault-tolerance. It follows that by combining our algorithm with that in [13] for adding fault-tolerance, it would be possible to permit synthesis of fault-tolerant programs without requiring the designer to specify the invariants explicitly.

Organization of the paper.

The rest of the paper is organized as follows: first, we define the problem statement in Section 2. Then, we present our algorithms in Section 3. Next, we illustrate our algorithm in Section 4. We discuss related work in Section 5 and the conclusion is in Section 6.

2 Programs and Specifications

A program p is a tuple $\langle S_p, \delta_p \rangle$ where S_p is a finite set of states, and δ_p is a subset of $\{(s_0, s_1) : s_0, s_1 \in S_p\}$. A state predicate of $p(= \langle S_p, \delta_p \rangle)$ is any subset of S_p. Since a state predicate can be characterized by the set of all states in which its Boolean expression is true, we use sets of states and state predicates interchangeably. Thus, conjunction, disjunction and negation of sets are the same as the conjunction, disjunction and negation of the respective state predicates.

A sequence of states, $\langle s_0, s_1, ... \rangle$ (denoted by σ), is a computation of $p(= \langle S_p, \delta_p \rangle)$ iff the following two conditions are satisfied:

- $\forall j : 0 < j < len(\sigma) : (s_{j-1}, s_j) \in \delta_p$,
- if σ is finite and the last state in σ is s_l then there does not exist state s such that $(s_l, s) \in \delta_p$.

Notation. We call δ_p as the transitions of p. When it is clear from context, we use p and δ_p interchangeably.

The safety specification for program p is specified in terms of bad states, b_s, and bad transitions b_t. A sequence $\langle s_0, s_1, ... \rangle$ (denoted by σ) satisfies the safety specification of p iff the following two conditions are satisfied.

- $\forall j : 0 \leq j < len(\sigma) : s_j \notin b_s$, and
- $\forall j : 0 < j < len(\sigma) : (s_{j-1}, s_j) \notin b_t$.

The liveness specification of program p is specified in terms of one or more leads-to properties of the form $\mathcal{F} \rightsquigarrow \mathcal{T}$. A sequence $\sigma = \langle s_0, s_1, ... \rangle$ satisfies $\mathcal{F} \rightsquigarrow \mathcal{T}$ iff

$\forall j : (\mathcal{F}$ is true in $s_j \Rightarrow \exists k : j \leq k < len(\sigma) : \mathcal{T}$ is true in $s_k)$. We assume that $\mathcal{F} \cap \mathcal{T} = \{\}$. If not, we can replace the property by $((\mathcal{F} - \mathcal{T}) \rightsquigarrow \mathcal{T})$

A program p satisfies the (safety and/or liveness) specifications from I iff every computation of p that starts from a state in I satisfies that specification.

The goal of the weakest invariant generation algorithm in this paper is to identify the weakest state predicate I such that every computation of p that starts in a state in I satisfies its safety specification and its liveness specification.

3 Weakest Invariant Generator Algorithm

In this section, we present our algorithm to automatically generate the weakest invariant using the program transitions and its specification. The goal of our algorithm is to generate the weakest possible invariant from where the program satisfy its safety and liveness specification. Our algorithm consists of three main parts: the invariant generator, the safety checker, and the liveness checker. We will describe each of the three algorithms in subsection 3.1-3.3.

3.1 Invariant Generator

The input to InvariantGenerator consists of the program transitions, $SPEC_{bs}$ (the states that should not be reached), $SPEC_{bt}$ (the transitions that should not be executed), and the liveness properties. It returns the weakest invariant from where the program satisfies its specification. First, the algorithm initializes the invariant I to be the whole state space (Line 1). Then, the algorithm computes the weakest-invariant by calling the function SafetyChecker (Line 4). At this point, I includes the set of states from where the program satisfies the given safety specification. Later, the algorithm satisfies the liveness properties one after another by calling the function LivenessChecker that removes states that violate the given liveness property (Lines 5-7). Removal of states due to liveness properties may require recomputation of I. Hence, this computation is in a loop and terminates when a fixpoint is reached.

Algorithm 1 InvariantGenerator

Input: program transitions p, $SPEC_{bs}$ (states that should not be reached), $SPEC_{bt}$ (transitions that should not be executed), $\mathcal{F}[]$, and $\mathcal{T}[]$ state predicates describing leads-to properties .
Output: weakest-invariant I.

```
    // Initially I equals S_p, the program states space.
1:  I = S_p
2:  repeat
3:          tmp = I
4:          I:= SafetyChecker(I, p, SPEC_bs, SPEC_bt);
             //check the i^th liveness properties
5:          for i := 0 to NoOfLivenessProperties  do
6:                  I:= LivenessChecker(I, p, F[i], T[i]);
7:          end for
8:  until tmp = I
    // return the weakest invariant.
9:  return  I;
```

3.2 Safety Checker

The input of the SafetyChecker algorithm consists of the initial invariant, the program transitions, the $SPEC_{bs}$, and the $SPEC_{bt}$. The output is the computed weakest invariant, I_{sf}, for the given safety specification.

First, the algorithm initializes the invariant I_{sf} to be the initial invariant I_{inp} excluding the states in $SPEC_{bs}$ (Line 1). Then, the algorithm starts a fixpoint computation that removes undesired states from the initial invariant. If I_{sf} contains a state s_0 such that the program can execute the transition $(s_0,\ s_1)$, which violates safety, then s_0 cannot be in I_{sf}. Hence, we remove s_0 from I_{sf} (Line 4). Note that a state is removed from I_{sf} only if the given program violates safety from that state. Thus , if I_{sf} contains a state s_0, p contains a transition $(s_0,\ s_1)$, and s_1 has been removed from I_{sf}, then s_0 must also be removed from I_{sf} (Line 5). This process continues until a fixpoint is reached. At this point, it exits the loop and returns the desired invariants I_{sf}.

Algorithm 2 SafetyChecker

Input: initial invariant I_{inp}, program transitions p, $SPEC_{bs}$ (states that should not be reached), $SPEC_{bt}$ (transitions that should not be executed)
Output: weakest-invariant I_{sf}.

// S_P is the state space of p
1: $I_{sf} := I_{inp} - SPEC_{bs}$;
2: **repeat**
3: $tmpI := I_{sf}$;
4: $I_{sf} := I_{sf} - \{s_0 : (s_0, s_1) \in p \cap SPEC_{bt}\ \}$;
5: $I_{sf} := I_{sf} - \{s_0 : (s_0, s_1) \in p \wedge s_0 \in I_{sf} \wedge s_1 \notin I_{sf}\}$;
6: **until** $tmpI = I_{sf}$
// return the set of states from where the program satisfies safety properties.
7: **return** I_{sf};

3.3 Liveness Checker

The input of the LivenessChecker algorithm consists of the initial invariant, I_{inp}, the program transitions, the $\mathcal{F}$ and $\mathcal{T}$ where $\mathcal{F} \rightsquigarrow \mathcal{T}$ is a given state predicates describing leads-to properties. The output is the largest set of states that is a subset of I_{inp} from where the given program satisfies $\mathcal{F} \rightsquigarrow \mathcal{T}$.

First, the algorithm creates a program $tmpP$ where we add a self-loop to all the deadlock states where the program p has no outgoing transitions from s_0 and $s_0 \notin \mathcal{T}$ (Line 1). Thus all computations of $tmpP$ are infinite or terminate in a state in $\mathcal{T}$. Now we remove all transitions in $tmpP$ that reach $\mathcal{T}$ (Line 2). If p satisfies $(\mathcal{F} \rightsquigarrow \mathcal{T})$ then it follows that $tmpP$ cannot include any infinite computation that includes a state in $\mathcal{F}$. Hence, the algorithm iteratively removes deadlock states in $tmpP$ (Lines 5-7). If some states in $\mathcal{F}$ still remain then it implies that there are infinite computations of $tmpP$ that begin in a state in $\mathcal{F}$ but do not reach a state in $\mathcal{T}$. Hence, we remove such states from I_{inp} and iteratively compute the invariant I_{inp}. fdf

Algorithm 3 LivenessChecker

Input: initial invariant I_{inp}, program transitions p, $\mathcal{F}$, and $\mathcal{T}$ state predicates describing leads-to properties.
Output: weakest-invariant I_{inp}.
// **ASSUMPTION:** $\mathcal{F} \cap \mathcal{T} = \{\}$. If not, change $\mathcal{F}$ to $(\mathcal{F} - \mathcal{T})$.

// let $ds(p) = \{s_0 : \forall s_1, (s_0, s_1) \notin p\}$ be the set of deadlock states.
// add self-loop to the states in $ds(p)$.
1: $tmpP := p \cup \{(s_0, s_0) : s_0 \notin \mathcal{T} \wedge s_0 \in ds(p)\}$;
2: $tmpP := \{(s_0, s_1) : (s_0, s_1) \in tmpP \wedge s_1 \notin \mathcal{T}\}$;

3: **repeat**
4: $invF := I_{inp}$;
5: **while** $(invF \cap ds(tmpP)) \neq \{\}$ **do**
6: $invF := invF - ds(tmpP)$;
7: **end while**
8: **if** $\mathcal{F} \cap invF \neq \{\}$ **then**
9: $I_{inp} := I_{inp} - (\mathcal{F} \cap invF)$;
10: **end if**
11: **until** $\mathcal{F} \cap invF = \{\}$
// return the set of states from where the program satisfies liveness properties.
12: **return** I_{inp};

Extension.

In some cases, the program actions are partitioned in terms of *system actions* and *environment actions*. It is expected that the environment actions will eventually stop (for a long enough time) so that the system actions can make progress (and satisfy liveness property). In such cases, we can apply the above algorithm as follows: The program actions used in SafetyChecker will consist of both the system actions and the environment actions. The program actions used for LivenessChecker will consist of only the system actions.

Theorem 3.1 *The Algorithm* InvariantGenerator *is sound (i.e the generated invariant is the weakest invariant that identifies the set of legitimate states for the program p).*

Proof. The proof consists of two parts: (1) if state, say s_0, is not included in the output of InvariantGenerator then the program does not satisfy its specification from s_0, and (2) if a state, say s_0, is included in the output of InvariantGenerator then the program satisfies its specification from s_0.

We now prove the first part by considering all parts of the code where some state is removed from the output.

- Line 1 of SafetyChecker : Clearly, states in $SPEC_{bs}$ cannot be included in the final invariant.
- Line 4 of SafetyChecker : If (s_0, s_1) is a transition of the program that violates safety then there is a computation of the program that starts from s_0 and violates the specification.
- Line 5 of SafetyChecker : If s_1 is a state already removed from the final invariant, i.e., there is a program computation that starts from s_1 and violates the specification, and (s_0, s_1) is a program transition then there exists a computation that starts from s_0 and violates the specification.
- Line 9 of LivenessChecker : Observe that in $tmpP$, transitions that reach $\mathcal{T}$ are

removed. Now, the loop on lines 5-7 removes all deadlock states in $invF$. If any state, say s_0, in $\mathcal{F}$ is not removed then that implies that there are infinite computations of $tmpP$ that start from s_0. For instance, this happens if a cycle is reachable from s_0. By construction, this computation cannot reach $\mathcal{T}$. Thus, if a state s_0 is removed on Line 9 of LivenessChecker then there is a computation from s_0 that violates the specification.

We use proof by contradiction for the second part. Suppose s_0 is included in the output of InvariantGenerator and there is a computation, say $\langle s_0, s_1, ...\rangle$ that violates the specification from s_0. We consider two cases depending upon whether this computation violates the safety specification or the liveness specification.

- **Safety specification.** Consider the first state where safety violation is detected, e.g., because a state, say s_j, in $SPEC_{bs}$ is reached or a transition, say (s_{j-1}, s_j) in $SPEC_{bt}$ is executed.
  - Case 1: $s_j \in SPEC_{bs}$. By Line 1 of SafetyChecker , $j \neq 0$. Also, from Line 5 of SafetyChecker , s_{j-1} would be removed from the final invariant. Likewise, s_{j-2} would be removed and so on. Thus, s_0 cannot be in the output of InvariantGenerator . This is a contradiction.
  - Case 2: $(s_{j-1}, s_j) \in SPEC_{bt}$. By the same argument as in Case 1, we can show that s_0 cannot be in the output of InvariantGenerator . This is a contradiction.
- **Liveness specification.** If this computation does not satisfy the liveness specification then this implies that it has a suffix where $\mathcal{F}$ is true in some state, say s_j, but $\mathcal{T}$ is false in all states. Now, we define a computation σ that starts from s_j. If the computation $\langle s_0, s_1, ...\rangle$ is infinite then σ is the suffix that starts from s_j. If not, i.e., it ends in a state, say s_l, where p has no outgoing transitions then σ is obtained by concatenating the suffix starting from s_j and an infinite stuttering of state s_l. By construction, σ is also a computation of $tmpP$ (Line 2 from LivenessChecker). Hence, s_j is removed from the output of InvariantGenerator . Again, by an argument similar to the case of safety specification, we can conclude that s_0 cannot be in the output of InvariantGenerator . This is a contradiction.

□

4 Case Studies

In Subsections 4.1-4.2, we describe and analyze case studies, namely the token ring program [5], and the Mutual Exclusion program [18]. We chose these classical examples from the literature of distributed computing to illustrates the feasibility and applicability of our algorithm in generating the weakest invariant.

Since we focus on the design of distributed programs, for brevity, we specify the state space of the program in terms of its variables. Each variable is associated with its domain. A state of the program is obtained by assigning each of its variables a value from the respective domain. The state space of the program is the set of all states.

To concisely describe the transitions of the program we use guarded command

notation: $\langle guard \rangle \rightarrow \langle statement \rangle$, where guard is a Boolean expression over program variables and the statement describes how program variables are updated and it always terminates. A guarded command of the form $g \rightarrow st$ corresponds to transitions of the form $\{(s_0, s_1) |\ g$ evaluates to true in s_0 and s_1 is obtained by executing st from $s_0\}$.

Throughout this section, all case studies are run on a MacBook Pro with 2.6 Ghz Intel Core 2 Duo processor and 4 GB RAM. The OBDD representation of the Boolean formula has been done using the C++ interface to the CUDD package developed at the University of Colorado [19].

4.1 Case Study 1: Token Ring

In this section, we illustrate our algorithm in the context of the token ring program. First, we specify the fault-intolerant program. Then, we provide its specification. Finally, we identify the invariant generated by the algorithm from Section 3.

Program. The token ring program consists of n processes organized in a ring. A token is circulated among the processes in a fixed direction. When a process gets the token it can access the critical section. Each process j, where $j \in \{0..n\}$, has a variable $x.j$ with the domain $\{0, 1, \perp\}$, where $\perp$ denotes that the process is in an illegitimate state. A process 0 has the token iff $x.n$ is equal to $x.0$ and a process j, where $1 \leq j \leq n$, has the token iff $x.j \neq x.(j-1)$.

The actions of the token ring program are as follows:

$$\begin{array}{lll} 1 :: x.j \neq x.(j-1) & \longrightarrow & x.j := x.(j-1); \\ 2 :: x.0 = x.n & \longrightarrow & x.0 := x.n +_2 1; \end{array}$$

where $+_2$ denotes modulo 2 addition.

Specification.

The safety specification of the token ring requires that the value of x at any process is either 0 or 1 and that no two processes have a token simultaneously. Thus, the safety specifications of the token ring program can be identified using the following set of bad states (i.e. states that should not be reached by normal program execution).

$$\begin{array}{l} SPEC_{TR_{bs}} = \\ \quad (\ \exists j, k : j \neq k\ \wedge\ j, k \in \{1..n\}\ ::\ ((x.(j-1) \neq x.j) \wedge (x.(k-1) \neq x.k))\ )\ \vee \\ \quad (\ \exists j : j \in \{1..n\}\ ::\ ((x.(j-1) \neq x.j) \wedge (x.0 = x.n))\ )\ \vee \\ \quad (\ \exists j\ :\ j \in \{0..n\}\ ::\ (x.j = \perp)\ ) \end{array}$$

The liveness specification of the token ring requires that eventually every process gets the token. The requirement that process 0 eventually gets the token can be specified as:

$$true \rightsquigarrow (x.0 = x.n).$$

Application of our algorithm.

After applying our algorithm with the above inputs, the generated weakest invariant can be represented using the following regular expression:

$$\langle x.0, x.1, x.2....x.n\rangle \in (0^l 1^{(n+1-l)} \cup 1^l 0^{(n+1-l)}), \text{ where } 0 \leq l \leq n+1.$$

Thus, the above invariant states that the sequence of $\langle x.0, x.1, x.2....x.n\rangle$ is a sequence of zeros followed by ones or ones followed by zeros. The value of $l+1$ in the above sequence identifies the process with the token.

We note that this is the exact same invariant used in [5] for adding fault-tolerance to the fault where up to n processes are detectably corrupted. Furthermore, the time for computing this invariant for different values of n is as shown in Table 1. As we can see, the time for generating the invariant is very small.

No. of Process	**Reachable States**	**Invariant Generation Time(Sec)**
10	10^4	0.1
20	10^9	0.2
30	10^{14}	0.3
40	10^{19}	0.4
50	10^{23}	0.6
100	10^{47}	0.19

Table 1
Invariant generation time for token ring program.

4.2 Case Study 2: Mutual Exclusion

In this section, we illustrate our algorithm in the context of the Raymond's tree-based mutual exclusion program [18]. Mutual exclusion is one of the fundamental problems in distributed/concurrent programs. One of the classical solutions to this problem is the token based solution due to Raymond[18]. In this solution, the processes form a directed rooted tree, *holder tree*, in which there is a unique token held at the tree root. If a process wants to access the critical section, it must first acquire the token. Our goal in this case study is to automatically generate the weakest invariant for the program in [1].

We start by specifying the fault-intolerant program. Then, we provide the program specification. Finally, we identify the invariant generated by our algorithm.

Program.

In the mutual exclusion program each process, say j where $j \in \{0..n\}$, has a variable $h.j$. If $h.j = j$ then j has the token. Otherwise, $h.j$ contains the process number of one of j's neighbors. In this program, a process can send the token to one of its neighbors. In particular, if j and k are adjacent, then the action by which k sends the token to j is as follows:

$$1 :: (h.k = k \ \wedge \ j \in Adj.k) \ \wedge \ (h.j = k) \longrightarrow h.k := j, \ \ h.j := j;$$

Where $Adj.k$ denote one of the neighbors of k.

Specification.

Since the goal of Raymond's mutual exclusion algorithm is to maintain a tree rooted at the token, it requires that the holder of any process is one of its tree neighbors. It also requires that there should be no cycles in the holder relation.

We formally describe the safety specifications in the following predicate:

$$\begin{aligned} SPEC_{ME_{bs}} \ = \ & (\ \exists j \in \{0..n\} :: ((h.j \neq j) \vee (h.j \neq p.j) \vee (h.j \neq ch.j)) \ ) \vee \\ & (\ \exists j, k \in \{0..n\} : j \neq k :: ((h.j = k) \wedge (h.k = j) \ )) \vee \\ & (\ \exists j, k \in \{0..n\} : j \neq k :: ((h.j = j) \wedge (h.k = k) \ )) \end{aligned}$$

Where $ch.j$ denote one of the children of j.

Application of our algorithm.

The generated weakest invariant predicate of the mutual exclusion program computed by our algorithm is as follows. The invariant predicate requires that j's holder can either be j's parent, j itself, or one of j's children. It also requires that the holder tree conforms to the parent tree and there are no cycles in the holder relation.

$$\begin{aligned} I_{ME} \ = \ & (\ \forall j \in \{0..n\} :: (h.j = P.j) \vee (h.j = j) \vee (\exists k : (P.k = j) \wedge (h.j = k)) \ ) \ \wedge \\ & (\ \forall j \in \{0..n\} :: (P.j \neq j) \Rightarrow (h.j = P.j) \vee (h.(P.j) = j) \ ) \ \wedge \\ & (\ \forall j \in \{0..n\} :: (P.j \neq j) \Rightarrow \neg((h.j = P.j) \wedge (h.(P.j) = j)) \ ) \end{aligned}$$

Where $P.j$ denote the parent of j.

The amount of time required for computing this invariant for a different number of processes is as shown in Table 2.

5 Related Work

Several techniques have been developed to verify program correctness [6,16,9,7,10,8]. For most of those methods, the program is translated into a logical formula that describes the program behavior and properties. Then, tools are used to verify the correctness of the program. For many of these tools identifying the program invariant is an essential step. Several approaches have been proposed to improve the automatic invariant generation[15,2,17,3,4]. These methods can be widely classified as either *top-down* or *bottom-up* approaches. The top-down approach starts with the weakest possible invariant and uses program specification to strengthen that

No. of Process	Reachable States	Invariant Generation Time(Sec)
10	10^9	0.01
20	10^{26}	0.1
30	10^{44}	0.2
40	10^{64}	0.5
50	10^{84}	0.9
100	10^{200}	0.43

Table 2
Invariant generation time for mutual exclusion program.

invariant. The bottom-up approach performs forward propagations of the program actions to derive the invariant. Our algorithm is a top-down approach since it starts by initializing the invariant to be the whole state space and later removes states that violate the predefined safety and liveness specifications.

Rustan, Leino, and Barnett [15,2] presented methods for forming an efficient weakest precondition to enhance the performance of the verification tools like ESC/Java and ESC/Modula3. Their goal is to simplify the presentation of the weakest pre-condition to avoid redundancy and to avoid exponential growth of the condition size. Our definition of invariant is equivalent to their definition of the weakest conservative preconditions in which the execution of a program statement does not go wrong and it terminates. However, in their work they address the problem of redundancy in describing such conditions while we focus on the automatic generation of such conditions from the program specification.

Jeffords and Heitmeyer [17,11] described an algorithm to automate the generation of the invariant. Their technique is based on deriving the invariant based on propositional formulas derived from the SCR tables. Their algorithm is intended for detecting errors at early stages of program design. By contrast, our algorithm is intended to discover the invariants of programs assumed to be correct for the purpose of adding fault-tolerance to such programs.

The accurate and complete identification of the invariant states is an essential step that enables the designers to apply the algorithms and tools to synthesize fault-tolerant programs from a fault-intolerant programs[12,14,5]. Our algorithm automates the generation of the invariants from program transitions and specification, which will have a significant improvement on simplifying the process of automated addition of fault-tolerance. Hence, this algorithm can be integrated with current

synthesis tools or it can be run in preliminary step to automatically discover the invariant.

6 Conclusion and Future Work

In this paper, we presented our algorithm for automated discovery of the weakest invariant of the given program. This problem was motivated by the need for finding such an invariant during the addition of fault-tolerance. In particular, one approach for adding fault-tolerance is to ensure that after the occurrence of faults, the program recovers to its legitimate states and subsequently satisfies its specification. Our approach allows the designer to discover the required legitimate states automatically.

Our algorithm uses the program actions and specification to automatically generate the weakest invariant. Our algorithm consists of two main steps. The first step is to generate the initial invariant to be the set of the states the given program dose not violate the safety specification. Then, the algorithm ensures that the generated invariant satisfies the liveness properties by removing any state that violates such properties. In the two case studies we used to demonstrate our algorithm, we generated the exact same invariant used in the automation of fault-tolerance for each of those cases. Furthermore, we note that the time needed to generate such invariants was very small.

In [12], authors presented algorithms for adding different levels of fault-tolerance. These algorithms are relatively complete with respect to the invariant specified by the designer. And, the authors argue that the invariant specified by the designer should be the weakest possible for improving success in synthesizing fault-tolerant programs. The result in the paper shows that this task can be fully automated. Moreover, the time required for identifying the weakest invariant is very small. For this reason, one future work is that we intend to combine our techniques for generating the weakest invariant with our tools for the automated synthesis of fault-tolerant programs.

References

[1] A. Arora and S. S. Kulkarni. Designing masking fault-tolerance via nonmasking fault-tolerance. *Proceedings of the Fourteenth Symposium on Reliable Distributed Systems, Bad Neuenahr, 1995, pages 174–185*, 14:174–185, 1995.

[2] M. Barnett and K.R.M. Leino. Weakest-precondition of unstructured programs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 82–87. ACM New York, NY, USA, 2005.

[3] S. BensMem, Y. Lakhnech, and H. Saidi. Powerful techniques for the automatic generation of invariants. In *Proc. 8th Int. Conf. on Computer-Aided Verification, to appear in Lect. Notes in Comput. Sci.* Springer, 1996.

[4] N. Bjørner, A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, 1997.

[5] 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.

[6] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[7] J.R. Burch, E.M. Clarke, and D.E. Long. Symbolic model checking with partitioned transition relations. In *International Conference on Very Large Scale Integration*, pages 49–58, 1991.

[8] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent system using temporal logic specifications: a practical approach. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 117–126, New York, NY, USA, 1983. ACM.

[9] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

[10] G. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 1997.

[11] R.D. Jeffords and C.L. Heitmeyer. An algorithm for strengthening state invariants generated from requirements specifications. In *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering (RE'01)*. IEEE Computer Society Washington, DC, USA, 2001.

[12] 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.

[13] 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.

[14] 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.

[15] K.R.M. Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6):281–288, 2005.

[16] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[17] J.W. Nimmer and M.D. Ernst. Automatic generation of program specifications. *ACM SIGSOFT Software Engineering Notes*, 27(4):229–239, 2002.

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

[19] F. Somenzi. CUDD: Colorado University Decision Diagram Package. http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html.