SAT-Based Synthesis of Fault-Tolerance [1]

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

Abstract

We present a technique where we use SAT solvers in automatic synthesis of fault-tolerant distributed programs from their fault-intolerant version. Since adding fault-tolerance to distributed programs is NP-complete, we use state-of-the-art SAT solvers to benefit from efficient heuristics integrated in SAT solvers to deal with the exponential complexity of adding fault-tolerance. Also, such SAT-based technique has the potential to use multiple instances of SAT solvers simultaneously so that independent sub-problems can be solved in parallel during synthesis.

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

1 Introduction

In the design of reliable systems, it is desirable to automatically synthesize a fault-tolerant program from its fault-intolerant version upon finding new classes of faults. Such automated approach generates a program that is correct by construction. However, the exponential complexity of synthesis is an obstacle in the synthesis of fault-tolerant *distributed* applications [1, 2]. In this paper, we propose a SAT-based synthesis approach for efficient synthesis of fault-tolerant programs.

To synthesize fault-tolerant distributed programs in polynomial time, Kulkarni, Arora, and Chippada [3] present a *heuristic-based* approach. Also, in [4], we present heuristics for enhancing the level of the fault-tolerance of distributed programs. Each heuristic identifies a deterministic order for the *verification* of synthesis requirements. Synthesis requirements are conditions that have to be met by program states and transitions during synthesis so that the synthesized fault-tolerant program is correct by construction. As a result, the efficiency of synthesis is directly affected by the efficiency of verifying synthesis requirements.

Previously, we developed a framework for the synthesis of fault-tolerant distributed programs [5]. In this framework, we exhaustively verify the synthesis requirements in different steps of the synthesis. Such exhaustive verification is too expensive when we synthesize programs with large state space. In this paper, we present a SAT-based approach where we use the state-of-the-art SAT solvers to verify the synthesis requirements. Specifically, to verify a particular synthesis requirement, we first transform the verification problem to a Boolean formula. Afterwards, we use SAT solvers to verify the satisfiability of the generated formula. As a result, instead of going through the states and the transitions of the program for the verification of a condition, we encode the verification problem as a Boolean formula, and then in one step query a SAT solver to solve the verification problem.

The impact of this technique is multi-fold: (i) We expect to improve the efficiency of synthesis since existing SAT solvers (e.g., zChaff [6]) have advanced heuristics to deal with the exponential nature of NP-complete problems; (ii) We use multiple instances of SAT solvers running in parallel to simultaneously solve independent sub-problems of synthesis on a distributed platform, and finally (iii) We adopt intelligent heuristics implemented in SAT solvers to design enhanced heuristics for the synthesis of fault-tolerant distributed programs.

The organization of the abstract. In Section 2, we present preliminary concepts. Then, in Section 3, we formally state the verification problem that the synthesis algorithm has to solve during the synthesis. Subsequently, in Section 4, we present our SAT-based synthesis method where we use SAT solvers to solve the verification problem. Finally, in Section 5, we make concluding remarks and discuss future work.

2 Preliminaries

In this section, we present basic concepts regarding modeling programs and the effect of distribution on the program processes. A comprehensive description of our modeling approach can be found in [1].

We specify a program p by a finite set of variables and a finite set of processes. Each variable is associated with a finite domain of values. A state of p is obtained by assigning each variable a value from its respective domain. The state space of p, S_p, is the set of all possible states of p. A state predicate of p is any subset of S_p. A transition predicate of p is any subset of $S_p \times S_p$. The set of transitions of each process is also a subset of $S_p \times S_p$.

In a distributed program p consisting of n processes $P_1, \cdots, P_n$, each process P_j may have read restrictions with respect to a private variable x of another process, say P_k ($1 \leq j, k \leq n$). As a result, this inability to read x creates an uncertain world for P_j where x can take different possible values of its finite domain. Thus, when P_j executes a transition, variable x can take $|d_x|$ different values, where $|d_x|$ is the size of the domain of x. This way, the effect of executing an individual transition of P_j is similar to the execution

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

of a group of $|d_x|$ different transitions according to $|d_x|$ different possibilities for the value of x. Hence, due to read restrictions, we model the set of transitions of each process P_j as a set of groups of transitions instead of a set of individual transitions.

In the synthesis of a fault-tolerant distributed program from its fault-intolerant version, we include groups of transitions (respectively, states) of the processes depending on the properties that they have. We specify a property $\mathcal{P}$ as a transition (respectively, state) predicate. For example, we verify if there exists a transition in a group of transitions whose execution causes the program to perform a bad action.

3 Problem Statement

In this section, we state the problem of verifying synthesis requirements. The synthesis algorithm specifies a synthesis requirement as a given property $\mathcal{P}$ that has to be verified for program transitions. Given a process P_j that consists of a set of groups of transitions $g_0, \cdots, g_m$ $(0 \leq m)$ and a property $\mathcal{P}$, we say P_j has the property $\mathcal{P}$ iff (if and only if) all groups of transitions $g_0, \cdots, g_m$ have the property $\mathcal{P}$. Also, a group of transition g_i $(0 \leq i \leq m)$ has the property $\mathcal{P}$ iff all transitions of g_i have the property $\mathcal{P}$. We present the verification problem as follows:

The verification problem

Given a group of transitions g, and a property $\mathcal{P}$:
Does g have the property $\mathcal{P}$? □

4 SAT-Based Synthesis

In this section, we present a SAT-based solution for verifying synthesis requirements. To verify synthesis requirements, we transform the problem of verifying a property $\mathcal{P}$ for a group of transitions g to a Boolean formula whose satisfiability can be verified by SAT solvers. Specifically, we define a function F that takes a group of transitions g and a property $\mathcal{P}$ and generates a Boolean formula. Since $\mathcal{P}$ is either a state predicate or a transition predicate, such transformation can be done in polynomial time in the state space of the program.

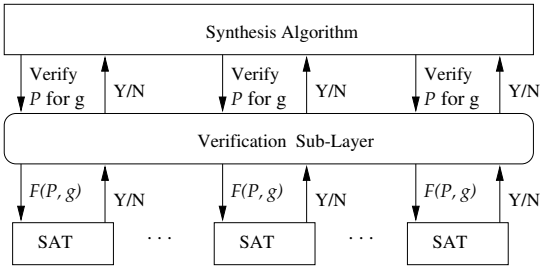

Figure 1. Using SAT solvers for the synthesis of fault-tolerant programs.

Using SAT solvers. Now, given the function F, we design a *verification sub-layer* that provides verification abilities to the synthesis algorithm (cf. Figure 1). Specifically, every time the synthesis algorithm needs to verify a property $\mathcal{P}$ of a group of transitions g, it sends $\mathcal{P}$ and g to the verification sub-layer. The verification sub-layer transforms the request of the synthesis algorithm to a Boolean formula $F(\mathcal{P}, g)$ and delivers it to the SAT solver. After the SAT solver provides the result of the satisfiability of $F(\mathcal{P}, g)$, the verification sub-layer forwards this result to the synthesis algorithm. Since the synthesis algorithm usually verifies a property $\mathcal{P}$ for all groups of transitions of the program being synthesized, the verification sub-layer creates multiple instances of the SAT solver to verify $\mathcal{P}$ for all groups of transitions in parallel.

Open research problems. To implement our SAT-based approach, we encounter some difficulties that we discuss next. First, the properties that we encounter during synthesis are not necessarily in normal form, whereas SAT solvers often require the input formula to be in conjunctive normal form (CNF). As a result, the Boolean formula $F(\mathcal{P}, g)$ must be in CNF. This issue creates a problem of *generating CNF formulas from arbitrary formulas*.

Second, we currently use SAT solvers as black boxes and every time the verification sub-layer needs to verify a formula, it invokes an instance of SAT solver. Such invocation is expensive since the current implementation of our framework is in Java and we invoke SAT solvers using native method calls. Hence, we would like to *identify the appropriate level of transparency* that SAT solvers should provide to the verification sub-layer.

Third, SAT solvers contain advanced heuristics for tackling the exponential complexity of SAT problem. Since the problem of synthesizing fault-tolerant distributed programs is also NP-complete [1,2], we expect that the heuristics used in SAT solvers can provide new insights for the design of enhanced heuristics applicable in the synthesis of fault-tolerant programs. Hence, a comprehensive *study of the advanced heuristics implemented in SAT solvers* will help us to develop more efficient heuristics.

5 Concluding Remarks and Future Work

We presented a technique for using SAT solvers in the synthesis of fault-tolerant distributed programs from their fault-intolerant version. We reduce the sub-problems that we encounter during the synthesis to the satisfiability problem and then invoke SAT solvers to solve those problems. This way, we reap the benefits of the efficiency of the state-of-the-art SAT solvers during the synthesis of fault-tolerant distributed programs. Currently, we have created a centralized implementation of our approach, however, we plan to extend this work for the cases where we deploy our synthesis algorithm on a distributed platform. Also, we plan to investigate the applicability of other decision procedures [7] in the synthesis of fault-tolerant distributed programs.

References

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

[2] S. S. Kulkarni and A. Ebnenasir. The complexity of adding failsafe fault-tolerance. *International Conference on Distributed Computing Systems*, page 337, 2002.

[3] S. S. Kulkarni, A. Arora, and A. Chippada. Polynomial time synthesis of byzantine agreement. *Symposium on Reliable Distributed Systems*, page 0130, 2001.

[4] S. S. Kulkarni and A. Ebnenasir. Enhancing the fault-tolerance of nonmasking programs. *International Conference on Distributed Computing Systems*, page 441, 2003.

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

[6] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. *39th Design Automation Conference, Las Vegas*, 2001.

[7] Jean-Christophe Filliâtre, Sam Owre, Harald Rueß, and N. Shankar. ICS: integrated canonizer and solver. *Proceedings of the 13th Conference on Computer-Aided Verification (CAV'01), volume 2102 of Lecture Notes in Computer Science. Springer-Verlag*, 2001.