Algorithmic Synthesis of Fault-Tolerant Distributed Programs

Ali Ebnenasir
Department of Computer Science and Engineering
Michigan State University
East Lansing MI 48824 USA
ebnenasi@cse.msu.edu

Abstract

Synthesizing fault-tolerant distributed programs from their fault-intolerant version is NP-hard in the size of the state space of the fault-intolerant program. We present an approach for dealing with this complexity and creating a framework to facilitate the algorithmic synthesis of fault-tolerant distributed programs. In particular, we propose three different directions for tackling the complexity of synthesis. In the first direction, we design heuristics to reduce the complexity of algorithmic synthesis. In another direction, we explore the polynomial-time boundary of algorithmic synthesis in order to find programs (respectively, specifications) for which algorithmic synthesis can be achieved in polynomial time. Yet another direction proposes the behavioral and structural analysis of the given fault-intolerant program in order to acquire some information that will be helpful during the synthesis of its fault-tolerant version. We validate our approach by the design and implementation of a synthesis framework that helps the developers of fault-tolerance in the development of fault-tolerant distributed programs.
Keywords : Fault-tolerance, Automatic addition of fault-tolerance, Program synthesis, Program transformation, Distributed programs

1 Introduction

We concentrate on algorithmic synthesis of fault-tolerant *distributed* programs from their fault-intolerant versions. Since it is difficult to anticipate all the faults that perturb a program, we need to incrementally add fault-tolerance to fault-intolerant programs as we encounter new classes of faults. In an incremental addition of fault-tolerance, we can reuse those behaviors of the fault-intolerant program that are hard to specify (e.g., efficiency) in a specification-based synthesis approach [1]. More specifically, an incremental algorithmic approach for the synthesis of fault-tolerant programs (i) reuses the computations of the given fault-intolerant program in the design of its fault-tolerant version, and (ii) guarantees the correctness of the fault-tolerant program by construction. Our approach differs from specification-based synthesis approaches where one can synthesize a program from its specification (cf. [1] for a survey on automatic synthesis approaches).

In general, the complexity of algorithmic synthesis of fault-tolerant distributed programs is exponential [2, 3]. This complexity is due to distribution issues that impose some read/write restrictions on the processes of a distributed program. In order to deal with this complexity Kulkarni, Arora, and Chippada [4] present a *heuristic-based* approach for polynomial-time synthesis of fault-tolerant distributed programs. Also, in order to recognize the polynomial-time boundary of algorithmic synthesis of fault-tolerant programs, Kulkarni and Ebnenasir [3] introduce a class of programs and specifications for which the synthesis of a failsafe fault-tolerant program can be done in polynomial-time in the state space of the fault-intolerant program.

We propose a three-directional approach in order to deal with the problem of *algorithmic synthesis of fault-tolerant distributed programs* from their fault-intolerant versions. These three directions are as follows: (i) Designing new heuristics for reducing the complexity of synthesis. (ii) Exploring the boundary of polynomial-time synthesis of fault-tolerant programs. (iii) Analyzing the behavior and structure of the fault-intolerant program in order to use the result of the analysis during the synthesis. Also, in parallel, we design a synthesis framework that helps the developers of fault-tolerance in the synthesis of fault-tolerant programs. Moreover, the synthesis framework helps us to validate the applicability of new heuristics in reducing the complexity of synthesis.

The contribution of our research work is multi-fold: (1) We provide a synthesis framework for the developers of fault-tolerance in order to benefit from the automation in the design of fault-tolerant programs. (2) We design our framework so that it is appropriate for educational purposes. (3) We conjecture that our approach in dealing with fault-tolerance will be useful in dealing with other non-functional concerns of distributed programs.

The organization of the abstract. In Section 2, we present theoretical issues regarding the algorithmic synthesis of fault-tolerant distributed programs. In Section 3, we discuss our motivations for the development of a synthesis framework (for algorithmic synthesis of fault-tolerance). Finally, in Section 4, we make concluding remarks and discuss future work.

2 Theoretical Issues

In this section, we describe three directions for dealing with the problem of algorithmic synthesis of fault-tolerant distributed programs. Specifically, in Section 2.1, we illustrate the effect of heuristics in reducing the complexity of algorithmic synthesis. In Section 2.2, we describe the importance of recognizing programs (respectively, specifications) for which the synthesis can be done in polynomial-time (in the size of the state space of the fault-intolerant program). Finally, in Section 2.3, we present the influence of behavioral and structural analysis of fault-intolerant programs in reducing the complexity of synthesis.

2.1 Heuristics

The exponential complexity of algorithmic synthesis of fault-tolerant distributed programs confines the application of these algorithmic techniques to programs where state space is small. Thus, for programs with large state space, we need to develop heuristics in order to reduce complexity. However, if the heuristics are not applicable then the synthesis algorithm declares that it cannot synthesize a fault-tolerant program even though there exists a fault-tolerant version of the given fault-intolerant program (i.e., incompleteness of the synthesis).

The application of heuristics depends on the problem at hand. Thus, the developers of fault-tolerance should be able to (i) select different subsets of heuristics for different problems, and (ii) apply heuristics in different order depending on their own discretion. Hence, we need a framework where the developers of fault-tolerance can interactively synthesize a fault-tolerant program (cf. Section 3).

2.2 Polynomial-Time Boundary

The drawback of using heuristics is that there exist situations where the heuristics fail in the synthesis even though a fault-tolerant version of the fault-intolerant program exists. Thus, it is desirable to find fault-intolerant programs (respectively, specifications) for which a fault-tolerant version can be synthesized in polynomial time if such a fault-tolerant version exists. In other words, we look for the properties of distributed programs and specifications where the algorithmic synthesis of fault-tolerance is sound and complete and can be done in polynomial-time. For example, Kulkarni and Ebnenasir [3] have identified *monotonicity* property for programs and specifications. They show that given a monotonic fault-intolerant program (respectively, a monotonic specification), a failsafe fault-tolerant program can be synthesized in polynomial time in the state space of the fault-intolerant program.

The knowledge of properties that reduce the complexity of synthesis to polynomial time opens up another frontier in our research work. In other words, if we know that the synthesis of fault-tolerance for programs with a specific property, say *Prop*, can be done in polynomial time then we search for the answer of the following question. *Given a fault-intolerant program p that does not satisfy Prop, how can we transform p to another program p' in polynomial time so that p' preserves all its properties and also satisfies Prop?*

2.3 Analysis of Fault-Intolerant Programs

During the synthesis of a fault-tolerant program, there exist situations where the knowledge of the behavior and the structure of the fault-intolerant program helps us to reduce the complexity of the synthesis. For example, given a fault-intolerant program, p, and a class of faults, f, we would like to answer the following question. *From a state s, does there exist a path of execution in p that reaches a deadlock state in the presence of f?* Also, from the structural point of view, we find situations where we can reuse the structure of the fault-intolerant program to deal with the complexity of synthesis. For example, if program p reaches a deadlock state s_d in the presence of f then we want to know whether the structure of p provides a finite number, say k, recovery paths from s_d or not [5].

Thus, before the synthesis, if we perform a structural (respectively, behavioral) analysis on the fault-intolerant program then we will use the results of the analysis during the synthesis of the fault-tolerant program. In this regard, we take advantage of model checkers to gather required information for the synthesis. In the case of a program whose specification is available, we use the specification to provide the required behavioral information. Since in some cases the extraction of behavioral information about the given fault-intolerant program is by itself difficult, we perform this analysis as far as our computational resources allow us to do so.

3 Synthesis Framework

In parallel with the design of synthesis algorithms, we develop a synthesis framework that helps the developers of fault-tolerance to synthesize fault-tolerant distributed programs. Also, the developers of heuristics can validate their heuristics using this synthesis framework. Thus, our synthesis framework targets two categories of users: *the developers of fault-tolerance*, and *the developers of heuristics*.

We use Dijkstra's guarded commands [6] as the interaction language between the user and our synthesis framework. Thus, users prepare the input fault-intolerant program in terms of guarded commands and receive the fault-tolerant program in guarded commands as well. The faults are also modeled as a set of guarded commands that update program variables.

The synthesis framework can automatically synthesize the output fault-tolerant program. However, there exist situations where the fully automatic synthesis fails. In such cases, the developers of fault-tolerance can interact with the framework and apply their own strategy.

The developers of heuristics also need to determine how their heuristics reduce the complexity of the synthesis. Furthermore, they need to identify if a heuristic is so restrictive that its use will cause the synthesis algorithm to declare failure very often. To achieve this goal, we present an extensible design for our framework in the sense that the developers of heuristics can easily integrate their new heuristics in the framework. Then, the developers of fault-tolerance can use these new heuristics. Moreover, since we want our synthesis framework to be platform independent and appropriate for pedagogical purposes, we have implemented the current version of the framework using Java environment [7].

4 Concluding Remarks and Future Work

Since the complexity of synthesizing a fault-tolerant distributed program from its fault-intolerant version is exponential in the state space of the fault-intolerant program [2,3], we have developed some heuristics [5] to deal with this complexity. We have also identified monotonic programs and specification [3] for which the synthesis can be done in polynomial time.

We also have designed and implemented a framework for the synthesis of fault-tolerant distributed programs. Using our synthesis framework, we have synthesized a fault-tolerant version of the following programs: a canonical version of Byzantine agreement program, token ring program with four processes, and an agreement program that is simultaneously subject to Byzantine and failstop faults.

In the future, we plan to move in the three directions that we mentioned in the Introduction, and simultaneously, we integrate new heuristics and algorithmic techniques in our synthesis framework.

References

[1] Ali Ebnenasir. Automatic synthesis of distributed programs: A survey. `http://www.cse.msu.edu/~ebnenasi/survey.pdf`, 2002.

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

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

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

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

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

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