Distributed Synthesis of Fault-Tolerance

Borzoo Bonakdarpour, Sandeep S. Kulkarni, 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}

1 Introduction

Synthesis algorithms usually 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 verified programs (e.g., [1]). In order to overcome the space explosion problem, recently, an increasing interest in parallel and distributed techniques has emerged in the model checking community (e.g., [2, 3]). Such techniques parallelize the state space of a given model over a network or cluster of workstations and run a distributed model checking algorithm over the parallelized state space. On the other hand, the space explosion problem is still unaddressed in the context of automated program synthesis.

With this motivation, we concentrate on the problem of designing distributed algorithms for automated program synthesis. More specifically, we parallelize two synthesis algorithms (from [1]) for adding two levels of fault-tolerance, namely failsafe and masking, to existing fault-intolerant programs. We assume that programs are in the high atomicity model, where all processes can read and write all the program variables in one atomic step.

2 Algorithm Sketches

In this paper, we only focus on designing a distributed algorithm that runs over a distributed state space. 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 parallelization technique proposed by Garavel, Mateescu, and Smarandache [2] 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 use 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 system 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 a detailed parallelized explicit state space.

**Distributed Synthesis of Failsafe Fault-Tolerance.** The essence of adding failsafe fault-tolerance consists of three parts: (1) a smallest fixpoint calculation for identifying the set of states from where safety may be violated, (2) a largest fixpoint calculation for computing the invariant of the failsafe program, and (3) emptiness checking of the synthesized program (to declare failure). Our distributed algorithm consists of a set of processes each running on one machine across the network. Each process consists of two threads, namely, **Distributed Add failsafe** and **MessageHandler**. Briefly, the thread **Distributed Add failsafe** is in charge of initiating local fixpoint calculations and synchronizing with other processes across the network. The thread **MessageHandler** is responsible for handling messages sent by other processes and invoking appropriate procedures. These messages inform a process whether a local state belongs to a global state predicate. For instance, if (1) a state $s_0$ is stored in machine $i$, (2) a state $s_1$ is stored in machine $j$, (3) there exists a fault transition $(s_0, s_1)$, and (4) safety may be violated from $s_1$, then $j$ sends a message to $i$ indicating that $s_0$ belongs to a global state predicate from where safety of the program may be violated.

**Distributed Synthesis of Masking Fault-Tolerance.** Similar to the distributed addition of failsafe fault-tolerance, our algorithm for adding masking fault-tolerance consists of two threads. For adding masking fault-tolerance, we first generate a failsafe program and then add recovery paths from each state in the fault-span (the set states reachable by both program and fault transitions) to a state in the invariant (a state predicate which captures the normal behavior of the program). 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 (transitions whose source and target states reside in different machines). In particular, we identify layers of states in the local fault-span corresponding to the number of steps of recovery paths. Since we require that recovery to the invariant must happen in a bounded number of steps, we identify the mentioned layers of states such that recovery transitions form no cycles in the fault-span. In other words, we construct a distributed tree whose leaves are states in the invariant in a distributed bottom-up fashion.

**Implementation and performance.** Since our synthesis algorithms are multithreaded and one of the threads are expected to be mostly busy with local computations, the computation time complexity is expected to be evenly distributed across the network. We plan to implement the distributed algorithms as an extension of our tool FTSyn, which is currently capable of synthesizing fault-tolerant programs using a single machine.

**References**