SYCRAFT: A Tool for Synthesizing Distributed Fault-Tolerant Programs*

Borzoo Bonakdarpour and Sandeep S. Kulkarni

Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48823, U.S.A.
{borzoo,sandeep}@cse.msu.edu

Abstract. We present the tool Sycraft (*SYmboliC synthesizeR and Adder of Fault-Tolerance*). In Sycraft, a distributed fault-intolerant program is specified in terms of a set of processes and an invariant. Each process is specified as a set of actions in a guarded command language, a set of variables that the process can read, and a set of variables that the process can write. Given a set of fault actions and a specification, the tool transforms the input distributed fault-intolerant program into a distributed fault-tolerant program via a symbolic implementation of respective algorithms.

1 Introduction

Distributed programs are often subject to unanticipated events called *faults* (e.g., message loss, processor crash, etc.) caused by the environment that the program is running in. Since identifying the set of all possible faults that a distributed system is subject to is almost unfeasible at design time, it is desirable for designers of *fault-tolerant* distributed programs to have access to synthesis methods that transform existing fault-intolerant distributed programs to a fault-tolerant version.

Kulkarni and Arora [4] provide solutions for automatic addition of fault-tolerance to fault-intolerant programs. Given an existing program, say p, a set f of faults, a safety condition, and a reachability constraint, their solution synthesizes a fault-tolerant program, say p', such that (1) in the absence of faults, the set of computations of p' is a subset of the set of computations of p, and (2) in the presence of faults, p' never violates its safety condition, and, starting from any state reachable by program and fault transitions, p' satisfies its reachability condition in a finite number of steps. In particular, they show that the synthesis problem in the context of distributed programs is NP-complete in the state space of the given intolerant program.

To cope with this complexity, in a previous work [2], we developed a set of symbolic heuristics for synthesizing moderate-sized fault-tolerant distributed

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

programs. The tool Sycraft implements these symbolic heuristics. It is written in C++ and the symbolic algorithms are implemented using the Glu/CUDD 2.1 package. The source code of Sycraft is available freely and can be downloaded from `http://www.cse.msu.edu/~sandeep/sycraft`

Related work. Our synthesis problem is in sprit close to controller synthesis and game theory problems. However, there exist several distinctions in theories and, hence, the corresponding tools. In particular, in controller synthesis and game theory, the notion of addition of *recovery* computations does not exist, which is a crucial concept in fault-tolerant systems. Moreover, we model *distribution* by specifying read-write restrictions, whereas related tools and methods (e.g., GASt, Supremica, and the SMT-based method in [3]) do not address the issue of distribution.

2 The Tool SYCRAFT

2.1 Input Language

We illustrate the input language and output format of Sycraft using a classic example from the literature of fault-tolerant distributed computing, the Byzantine agreement problem [5] (Figure 1). In Byzantine agreement, the program consists of a *general* `g` and three (or more) *non-general* processes: `0`, `1`, and `2`. Since, the non-general processes have the same structure, we model them as a process `j` that ranges over 0..2. The general is not modeled as a process, but by two global variables bg and dg. Each process maintains a decision d; for the general, the decision can be either 0 or 1, and for the non-general processes, the decision can be 0, 1 or 2, where the value 2 denotes that the corresponding process has not yet received the value from the general. Each non-general process also maintains a Boolean variable f that denotes whether that process has finalized its decision. To represent a Byzantine process, we introduce a variable b for each process; if $b.j$ is true then process `j` is Byzantine. In Sycraft, a distributed fault-intolerant program comprises of the following sections:

Process sections. Each process includes (1) a set of *process actions* given in *guarded commands*, (2) a *fault section* which accommodates fault actions that the process is subject to, (3) a *prohibited* section which defines a set of transitions that the process is not allowed to execute, (4) a set of variables that the process is allowed to *read*, and (5) a set of variables that the process is allowed to *write*. The syntax of actions is of the form $g \rightarrow st$, where the guard g is a Boolean expression and statement st is a set of (possibly non-deterministic) assignments. The semantics of actions is such that at each time, one of the actions whose guard is evaluated as *true* is chosen to be executed non-deterministically. The read-write restrictions model the issue of distribution in the input program. Note that in Sycraft, fault actions are able to read and write all the program variables. Prohibited transitions are specified as a conjunction of an (unprimed) source predicate and a (primed) target predicate.

In the context of Byzantine agreement, each non-general process copies the decision value from the general (Line 6) and then finalizes that value (Line 8). A fault action may turn a process to a Byzantine process, if no other process is Byzantine (Line 10). Faults also change the decision (i.e., variables *d* and *f*) of a Byzantine process arbitrarily and non-deterministically (Line 12). In Sycraft, one can specify faults that are not associated with a particular process. This feature is useful for cases where faults affect global variables, e.g., the decision of the general (Lines 18-22). In the *prohibited* section, we specify transitions that violate safety by process (and not fault) actions. For instance, it is unacceptable for a process to change its final decision (Line 14). Finally, a non-general process is allowed to read its own and other processes' *d* values and update its own *d* and *f* values (Lines 15-16).

Invariant section. The invariant predicate has a triple role: it (1) is a set of states from where execution of the program satisfies its safety specification (described below) in the absence of faults, (2) must be closed in the execution of the input program and, (3) specifies the *reachability* condition of the program, i.e., if occurrence of faults results in reaching a state outside the invariant, the (synthesized) fault-tolerant program must *safely* reach the invariant predicate in a finite number of steps. In order to increase the chances of successful synthesis, it is desired that Sycraft is given the weakest possible invariant. In the context of our example, the following states define the invariant: (1) at most one process may be Byzantine (Line 24), (2) the *d* value of a non-Byzantine non-general process is either 2 or equal to *dg* (Line 25), and (3) a non-Byzantine undecided process cannot finalize its decision (Line 26), or, if the general is Byzantine and other processes are non-Byzantine their decisions must be identical and not equal to 2 (Line 28).

Specification section. Our notion of specification is based on the one defined by Alpern and Schneider [1]. The specification section describes the *safety* specification as a set of *bad prefixes* that should not be executed neither by a process nor a fault action. Currently, the size of such prefixes in Sycraft is two (i.e., a set of transitions). The syntax of specification section is the same as prohibited section described above. In the context of our example, *agreement* requires that the final decision of two non-Byzantine processes cannot be different (Lines 30-31). And, *validity* requires that if the general is non-Byzantine then the final decision of a non-Byzantine process must be the same as that of the general (Lines 32). Notice that in the presence of a Byzantine process, the program may violate the safety specification.

2.2 Internal Functionality

Sycraft implements three nested symbolic fixedpoint computations. The inner fixedpoint deals with computing the set of states reachable by the input intolerant program and fault transitions. The second fixedpoint computation identifies the set *ms* of states from where the safety condition may be violated by fault transitions. It makes *ms* unreachable by removing program transitions that end

```
1)  program Byzantine_Agreement;
2)  const N = 2;
3)  var boolean bg, b.0..N, f.0..N;
4)     int dg: domain 0..1, d.0..N: domain 0..2;
    {------------------------------------------------------------------------}
5)  process j: 0..N
6)        ((d.j == 2) & !f.j & !b.j) --> d.j := dg;
7)     []
8)        ((d.j != 2) & !f.j & !b.j) --> f.j := true;
9)     fault Byzantine_NonGeneral
10)             (!bg) & (forall p in 0..N::(!b.p)) --> b.j := true;
11)       []
12)             (b.j) --> (d.j := 1) [] (d.j := 0) []
                          (f.j := false) [] (f.j := true);
13)    endfault
14)    prohibited (!b.j)&(!b.j’)&(f.j)&((d.j!=d.j’) | (!f.j’))
15)    read: d.0..N, dg, f.j, b.j;
16)    write: d.j, f.j;
17) endprocess
    {------------------------------------------------------------------------}
18) fault Byzantine_General
19)       !bg & (forall p in 0..N:: (!b.p)) --> bg := true;
20)    []
21)       bg --> (dg := 1) [] (dg := 0);
22) endfault
    {------------------------------------------------------------------------}
23) invariant
24)    (!bg & (forall p, q in 0..N:(p != q):: (!b.p | !b.q))&
25)    (forall r in 0..N::(!b.r => ((d.r == 2) | (d.r == dg)))) &
26)    (forall s in 0..N:: ((!b.s & f.s) => (d.s != 2))))
27)    |
28)    (bg & (forall t in 0..N:: (!b.t)) &
             (forall a,b in 0..N::((d.a==d.b)&(d.a!=2))));
    {------------------------------------------------------------------------}
29) specification:
30)    (exists p, q in 0..N: (p != q) :: (!b.p’ & !b.q’ & (d.p’ != 2) &
31)       (d.q’ != 2) & (d.p’ != d.q’) & f.p’ & f.q’))  |
32)    (exists r in 0..N:: (!bg’ & !b.r’ & f.r’ & (d.r’ != 2) & (d.r’ != dg’)));
```

Fig. 1. The Byzantine agreement problem as input to Sycraft

in a state in *ms*. Note that in a distributed program, since processes cannot read and write all variables, each transition is associated with a *group* of transitions. Thus, removal or addition of a transition must be done along with its corresponding group. The outer fixedpoint computation deals with resolving *deadlock* states by either (if possible) adding safe recovery simple paths from deadlock states to the program's invariant predicate, or, making them unreachable via adding minimal restrictions on the program.

2.3 Output Format

The output of Sycraft is a fault-tolerant program in terms of its actions. Figure 2 shows the fault-tolerant version of Byzantine agreement with respect to process $\mathsf{j} = 0$. Sycraft organizes these actions in three categories. *Unchanged actions* entirely exist in the input program (e.g., action 1). *Revised actions* exist in the input program, but their guard or statement have been strengthened (e.g., Line 8 in Figure 1 and actions 2-5 in Figure 2). Sycraft adds *Recovery actions* to

```
--------------------------------------------------------------------------------
UNCHANGED ACTIONS:
--------------------------------------------------------------------------------
1-((d0==2) & !(f0==1)) & !(b0==1)           -->   (d0:=dg)
--------------------------------------------------------------------------------
REVISED ACTIONS:
--------------------------------------------------------------------------------
2-(b0==0) & (d0==1) & (d1==1) & (f0==0)     -->   (f0:=1)
3-(b0==0) & (d0==0) & (d2==0) & (f0==0)     -->   (f0:=1)
4-(b0==0) & (d0==0) & (d1==0) & (f0==0)     -->   (f0:=1)
5-(b0==0) & (d0==1) & (d2==1) & (f0==0)     -->   (f0:=1)
--------------------------------------------------------------------------------
NEW RECOVERY ACTIONS:
--------------------------------------------------------------------------------
6-(b0==0)&(d0==0)&(d1==1)&(d2==1)&(f0==0)   -->   (d0:=1)[]((d0:=1), (f0:=1))
7-(b0==0)&(d0==1)&(d1==0)&(d2==0)&(f0==0)   -->   (d0:=0)[]((d0:=1), (f0:=1))
--------------------------------------------------------------------------------
```

Fig. 2. Fault-tolerant Byzantine agreement

enable the program to safely converge to its invariant predicate. Notice that the strengthened actions prohibit the program to reach a state from where validity or agreement is violated in the presence of faults. It also prohibits the program to reach deadlock states from where safe recovery is not possible.

3 Conclusion

Sycraft allows for transformation of moderate-sized fault-intolerant distributed programs to their fault-tolerant version with respect to a set of uncontrollable faults, a safety specification, and a reachability constraint. In addition to the obvious benefits of automated program synthesis, we have observed that Sycraft can be potentially used to debug specifications as well, since the algorithms in Sycraft tend to add minimal restrictions on the synthesized program. Thus, testing approaches can be used to evaluate behaviors of the synthesized programs to identify missing requirements. To address this potential application of program synthesis, we plan to add *supervised synthesis* features to Sycraft.

References

1. Alpern, B., Schneider, F.B.: Defining liveness. Information Processing Letters 21, 181–185 (1985)
2. Bonakdarpour, B., Kulkarni, S.S.: Exploiting symbolic techniques in automated synthesis of distributed programs with large state space. In: IEEE International Conference on Distributed Computing Systems (ICDCS), pp. 3–10 (2007)
3. Finkbeiner, B., Schewe, S.: SMT-based synthesis of distributed systems. In: Automated Formal Methods (AFM) (2007)
4. Kulkarni, S.S., Arora, A.: Automating the addition of fault-tolerance. In: Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT), pp. 82–93 (2000)
5. Lamport, L., Shostak, R., Pease, M.: The Byzantine generals problem. ACM Transactions on Programming Languages and Systems (TOPLAS) 4(3), 382–401 (1982)