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 processing brings a high level of computing power as well as robustness to computer systems. However, distribution of resources is often subject to unanticipated events called *faults* (e.g., message loss, processor crash, etc.) caused by the environment that a 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 [5] 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.

* This is an extended version of a paper appeared in the proceedings of CONCUR'08: International Conference on Concurrency Theory, LNCS, Springer-Verlag, 2008. This work was partially sponsored by NSF CAREER CCR-0092724 and ONR Grant N00014-01-1-0744.

In order to overcome the complexity issues, in a previous work [3], we developed a set of symbolic heuristics for synthesizing fault-tolerant distributed programs with state space of size 10^{50} and beyond. 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[1]. The source code of SYCRAFT is available freely and can be downloaded from `http://www.cse.msu.edu/~borzoo/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 [4]) do not address the issue of distribution.

2 The Tool SYCRAFT

The tool SYCRAFT implements a set of symbolic heuristics for synthesizing fault-tolerant distributed programs. It has been tested using various case studies. These case studies include problems from the literature of fault-tolerant computing in distributed systems (e.g., Byzantine agreement, Byzantine agreement with fail-stop faults, Byzantine agreement with constrained specification, token ring, triple modular redundancy, alternating bit protocol) as well as examples from research and industrial institutions (e.g., an altitude switch controller [2], and a data dissemination protocol in sensor networks [6]). The tool has been tested on Sun Solaris, Mac OS, and various distributions of Linux (e.g., Debian, Ubuntu, and Fedora).

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 [7] (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

[1] Details about Colorado University Decision Diagram Package is available at `http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html`.

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, where process j cannot read $b.k$ for $k \neq j$. In Sycraft, a distributed fault-intolerant program comprises of:

- **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 $g \to st$, 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).

```
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.

– **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 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. We note that the BDD-based implementation of the aforementioned fixedpoint computations does not apply dynamic variable reordering in the current version of SYCRAFT.

Figure 2 demonstrates (1) steps of heuristics implemented in SYCRAFT, (2) verification of corresponding concerns such as satisfaction of safety, closure of invariant and *fault-span* (i.e., the set of states reachable by program and fault actions), and nonexistence of deadlock states for each step (denoted by *OK* and *FAILED*), and (3) satisfaction of fixedpoint computations. In particular, the tool has a solution to the synthesis problem when the answer to verification of all above concerns is affirmative.

As mentioned in the introduction, the problem of synthesizing distributed fault-tolerant programs is known to be NP-complete and SYCRAFT implements a set of heuristics [3] to deal with this complexity. Obviously, the heuristics are incomplete in the sense that they may be unable to synthesize a solution while there exists one. Although we have not observed this incompleteness in our case studies, the incompleteness of heuristics may be observed where recovery through a state outside the fault-span is possible.

The performance of the symbolic algorithms that SYCRAFT implements is discussed in detail in [3] using case studies including the Byzantine agreement and token ring problems with various number of processes. In particular, by applying more advanced techniques than those introduced in [3] (e.g., exploiting symmetry in distributed processes and state space partitioning) we have been able to synthesize Byzantine agreement with 3 processes in 0.07s and up to 40 processes (reachable states of size 10^{50}) in less than 100 minutes using a regular personal computer.

2.3 Output Format

The output of SYCRAFT is a fault-tolerant program in terms of its actions. Figure 3 shows the fault-tolerant version of Byzantine agreement with respect to process $\mathsf{j} = \mathsf{0}$. SYCRAFT organizes these actions in three categories. *Unchanged actions* entirely exist in the input program (e.g., Line 6 in Figure 1 and action 1 in Figure 3). *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 3). SYCRAFT adds *Recovery actions* to 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

```
$ bin/sycraft examples/ByzantineAgreement.fin

Initializing MDD manager Glu2.1...
Compiling the input fault-intolerant program...
Creating the symbol table...
Creating intolerant program MDDs...
Input program compiled successfully

Computing ms....
Computing mt....
Running the synthesis algorithms for 3 processes

SAFETY.........................OK.
DEADLOCKS......................OK.
INVARIANT NONEMPTINESS.........OK.
INVARIANT CLOSURE..............OK.
F-SPAN CLOSURE.............FAILED.

Computing the fault-span...
Removing unsafe transitions...
Unsafe transitions removed...
Computing the fault-span...
Step 3-4 Fixpoint reached ...

SAFETY.........................OK.
DEADLOCKS..................FAILED.
INVARIANT NONEMPTINESS.........OK.
INVARIANT CLOSURE..............OK.
F-SPAN CLOSURE.................OK.

Resolving deadlocks by adding recovery paths
Removing unsafe transitions...
Resolving deadlocks by adding recovery paths
Removing unsafe transitions...
Cycle(s) detected/removed from the fault-span...

SOME RECOVERY ACTION ADDED

Eliminating remaining deadlock states...
Eliminating remaining deadlock states...
Eliminating remaining deadlock states...
Eliminating remaining deadlock states...
Eliminating remaining deadlock states...
Eliminating remaining deadlock states...
Eliminating remaining deadlock states...
Something was eliminated.

SAFETY.........................OK.
DEADLOCKS..................FAILED.
INVARIANT NONEMPTINESS.........OK.
INVARIANT CLOSURE..............OK.
F-SPAN CLOSURE.................OK.

Computing the fault-span...
Removing unsafe transitions...
Step 3-4 Fixpoint reached ...

SAFETY.........................OK.
DEADLOCKS..................FAILED.
INVARIANT NONEMPTINESS.........OK.
INVARIANT CLOSURE..............OK.
F-SPAN CLOSURE.............FAILED.
Resolving deadlocks by adding recovery paths
Removing unsafe transitions...
Resolving deadlocks by adding recovery paths
Removing unsafe transitions...
SOME RECOVERY ACTION ADDED

Eliminating remaining deadlock states...
No program transitions removed.

SAFETY.........................OK.
DEADLOCKS......................OK.
INVARIANT NONEMPTINESS.........OK.
INVARIANT CLOSURE..............OK.
F-SPAN CLOSURE.............FAILED.

Computing the fault-span...
Removing unsafe transitions...
Step 3-4 Fixpoint reached ...

SAFETY.....................FAILED.
DEADLOCKS..................FAILED.
INVARIANT NONEMPTINESS.........OK.
INVARIANT CLOSURE..............OK.
F-SPAN CLOSURE.............FAILED.

Resolving deadlocks by adding recovery paths
Removing unsafe transitions...
Resolving deadlocks by adding recovery paths
Removing unsafe transitions...
SOME RECOVERY ACTION ADDED

Eliminating remaining deadlock states...
No program transitions removed.

SAFETY.....................FAILED.
DEADLOCKS......................OK.
INVARIANT NONEMPTINESS.........OK.
INVARIANT CLOSURE..............OK.
F-SPAN CLOSURE.............FAILED.

Computing the fault-span...
Removing unsafe transitions...
Step 3-4 Fixpoint reached ...
Step 3-5 Fixpoint reached ...
Step 3-7 Fixpoint reached ...
SAFETY.........................OK.
DEADLOCKS......................OK.
INVARIANT NONEMPTINESS.........OK.
INVARIANT CLOSURE..............OK.
F-SPAN CLOSURE.................OK.

Total synthesis time = 0.125s

Select the process you wish to see its
fault-tolerant version:
(1)  j:0..2      (example: "j 1")
(2) quit

Output process:

Destroying MDD manager Glu2.1...
$
```

Fig. 2. A sample run snapshot of SYCRAFT .

```
---------------------------------------------------------------------------
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)
7-(b0==0)& (d0==1)& (d1==0)& (d2==0)& (f0==0)   -->   (d0 := 0)
8-(b0==0)& (d0==0)& (d1==1)& (d2==1)& (f0==0)   -->   (d0 := 1), (f0 := 1)
9-(b0==0)& (d0==1)& (d1==0)& (d2==0)& (f0==0)   -->   (d0 := 0), (f0 := 1)
---------------------------------------------------------------------------
```

Fig. 3. Fault-tolerant Byzantine agreement.

also prohibits the program to reach deadlock states from where safe recovery is not possible.

3 Conclusion

In addition to the obvious benefits of automated program synthesis, we have observed that SYCRAFT can be potentially used to debug specifications as well. In particular, since the algorithms in SYCRAFT tend to add minimal restrictions on the synthesized program, it is expected that if a requirement is missing in a specification then the synthesized program would exhibit behaviors that violate that requirement. Thus, testing approaches can be used to evaluate behaviors of the synthesized programs to identify missing requirements. Future extensions to SYCRAFT include:

- **Parallel synthesis.** We are currently adding multi-core features to SYCRAFT for further performance improvement. Our preliminary experiments show significant improvements.
- **The issue of invariant predicate.** Currently, the user is required to manually specify the invariant of the given fault-intolerant program. As mentioned in the paper, the invariant is a state predicate from where the execution of program in the absence of faults does not violate the specification. We plan to incorporate invariant generation techniques, so that the user only specifies a reachability goal for recovery. One trivial way is by computing the set of reachable states from a set of initial states. Note, however, that having

weakest possible invariant is often desirable, as it increases the chances of successful synthesis.

- **Supervised synthesis.** It is often desirable for users to be able to observe and analyze intermediate programs and predicates during the course of synthesis. The old (enumerative) version of Sycraft provides supervised synthesis and we plan to add this feature to the symbolic version as well.
- **Input language.** Guarded commands provide a concise and expressive way to specify models. However, it is often desirable to have access to wrappers that make the input language more readable and easy-to-understand. We plan to incorporate such wrappers for the input language. We have already designed one such wrapper for NRL's SCR which will be available along with Sycraft soon.

Concluding, 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. Our experiments [3] show encouraging results paving the path for using program synthesis in practice.

References

1. B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.
2. R. Bharadwaj and C. Heitmeyer. Developing high assurance avionics systems with the SCR requirements method. In *Digital Avionics Systems Conference*, 2000.
3. 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.
4. B. Finkbeiner and S. Schewe. SMT-based synthesis of distributed systems. In *Automated Formal Methods (AFM)*, 2007.
5. 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.
6. S. S. Kulkarni and M. Arumugam. Infuse: A TDMA based data dissemination protocol for sensor networks. *International Journal on Distributed Sensor Networks (IJDSN)*, 2(1):55–78, 2006.
7. L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.