

SYCRAFT User Manual

Borzoo Bonakdarpour

Last update: September 2008

1 Overview

This document is a user manual for 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.

The tool 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 [4]). It is written in C++ and the symbolic algorithms are implemented using the Glu/CUDD 2.1 package¹. The source code of SYCRAFT is available freely and can be downloaded from <http://www.cse.msu.edu/~borzoo/sycraft>. Installation instructions and other resources are also available in the web site. The tool has been tested on Sun Solaris, Mac OS, and various distributions of Linux (e.g., Debian, Ubuntu, and Fedora).

2 SYCRAFT Input Program Language and Grammar

Every input program to SYCRAFT has eight declaration sections. These sections are meant to declare program name, constants, variables, processes, faults, invariant, safety specification, and finally a transition predicate which the output program is prohibited to execute.

```
<prog> ::=      <progdecl> <constdecl> <vardecl> <procdecl> <faultdecl>  
              <invariantdecl> <specdecl> <prohibited>
```

```
<progdecl> ::= "program" <identifier> ";"
```

The rule <identifier> is reduced by SYCRAFT's lexical analyzer and returned as a token. An identifier can be any combination of alphanumeric characters. The only constraints are an identifier has to start with an alphabet and case matters. A typical structure of a SYCRAFT input program is illustrated bellow:

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

```

program MyProgram;

const
...
var
...
process p1
...
endprocess
process p2
...
endprocess
fault MyFaults
...
endfault
invariant
...
specification
...
prohibited
...

```

2.1 Program, Constant, and Variable Declaration

We now describe each declaration section by starting from constants. *Constants* are often used to facilitate parameterizing Booleans and integers such as domain of variables, number of processes, or range of quantifiers.

```

<constdecl> ::=      "const" <constlist> | <empty>
<constlist> ::=     <concreteconst> | <constlist> <concreteconst>
<concreteconst> ::= "int" <identifuer> "=" <guard> ";" |
                   "boolean" <identifier> "=" <guard> ";"
<empty> ::=

```

We explain the rule <guard> in detail in the next section. For now, a guard can be any combination of arithmetic and Boolean expressions, including a single integer or Boolean. For example, the following is a constant declaration:

```

const
  int N = 2;
  int bot = 2;
  int M = N - 1;

```

In SYCRAFT, *variables* are of two types: nonnegative integer and Boolean. Note that SYCRAFT does not have a perfect type checker and it is user’s responsibility to ensure correct type checking. Each integer must be declared along with its domain. The lower-bound of an integer domain must be zero, but its upper bound is optional. Variables can be defined as an array. The index of the first element of an array is always zero.

```

<vardecl> ::=      "var" <varlist>
<varlist> ::=      <concretevar> | <varlist> <concretevar>
<concretevar> ::=  <booldecl> | <intdecl>
<booldecl> ::=    "boolean" <identifuer> ";" |
                  "boolean" <identifier> "." <range> ";"
<range> ::=       <rangedelim> ".." <rangedelim>
<rangedelim> ::=  <guard>
<intdecl> ::=     "int" <indetifier> ": domain" <range> ";" |
                  "int" <identifier> "." <range> ": domain" <range> ";"

```

For example, the following is a variable declaration.

```

var
    boolean bg;
    boolean b.0..N;
    boolean f.0..N;
    int dg: domain 0..1;
    int d.0..N: domain 0..2;

```

One can refer to elements of array *d* declared above, by *d.0*, *d.1*, ..., *d.N*. One can also use expressions such as *d.(i+1)* where *i* is an integer ranges over 0..N-1. We explain parameterized variable elements in the next section.

2.2 Process Declaration and Structure

As mentioned earlier, a distributed program consists of a set of *processes*. In SYCRAFT, 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.

```
<procdecl> ::= <procstruct> | <procdecl> <procstruct>
<procstruct> ::= "process" <identifier> <procrange> <faultdecl>
                <prohibited> <rwrestrict> "endprocess"
<procrange> ::= ":" <range> | <empty>
```

For example, a process declarations template without range is the following:

```
process MyProcess

    guarded commands
    ...
    fault actions
    ...
    prohibited transitions
    ...
    read/write restrictions

endprocess
```

Likewise, an array of symmetric processes can be defined as follows:

```
process j:0..N
...
endprocess
```

We emphasize that fault actions and prohibited transitions sections are allowed to be empty in a process. The reason for providing the feature of defining an array of processes is due to the fact that many distributed programs consist of identical processes that can be modeled through parametrization. We now describe constituents of a process. First, the set of guarded commands which essentially model the behavior of a process.

```
<actions> ::= <actionlist>
<actionlist> ::= <action> | <actionlist> "[]" <action>
<action> ::= <guard> "- ->" <statement> ";
```

The following is an example of two guarded commands (actions) joined by "[]" operator:

<pre> (d.j == bot) & !f.j & !b.j --> d.j := dg; [] (d.j != bot) & !f.j & !b.j --> f.j := true; </pre>

We now describe each nonterminal of the above rules for defining a process. The first rule is `<guard>`. A guard is essentially a combination of quantified Boolean and arithmetic expressions.

```

<guard> ::=
    <guard> "&" <guard> |
    <guard> "|" <guard> |
    <guard> "=>" <guard> |
    <guard> "^" <guard> |
    "!" <guard> |
    "exists" <boundlist> "in" <range> <quantcond> ":: (" <guard> ")" |
    "forall" <boundlist> "in" <range> <quantcond> ":: (" <guard> ")" |
    <arithmeticexp> |
    <comparison>

```

```

<boundlist> ::= <identifier> | <boundlist> "," <identifier>
<quantcond> ::= ":" (" <guard> ")" | <empty>

```

The first five rules of "guard" are concerned with logical and, or, implication, xor, and negation operators, respectively. The next two rules are concerned with universal and existential quantifiers, respectively. A quantifier has a list of bound variables separated by comma. All variables must be from the same range. Moreover, a quantifier may have two conditions determined by the internal nonterminals `<quantcond>` and `<guard>`. The first condition reduced by rule `<quantcond>` is on bound variables. The second condition is on which SYCRAFT's compiler eliminates the quantifier. Although the above grammar allows nested quantifiers, we emphasize that SYCRAFT's code generator currently does not support nested quantifiers. In other words, an input program containing nested quantifiers may compile successfully, but the corresponding BDDs would encode a different expression. We describe the syntax of arithmetic and comparison expressions later in this section. The following is an example of a Boolean expression without arithmetic expression:

<pre>!bg & (forall p, q in 0..N: (p != q) :: (!b.p !b.q))</pre>

where `N` is a constant. We note that in SYCRAFT, equality of Boolean variables has to be expressed using Boolean operators only. For instance, the following expressions are not meaningful to SYCRAFT:

- `(x == true)`,
- `(x == false)`,
- `(x == y)`, and
- `(x != y)`.

The right way to write the above expressions in SYCRAFT is respectively as follows:

- x ,
- $!x$,
- $!(x \wedge y)$, and
- $(x \wedge y)$.

In addition to logical operators, expressions can contain arithmetic expressions and integer comparisons as well. Arithmetic expressions may include addition, subtraction, and modulo operators. Comparisons may involve equality, inequality, greater than, less than, greater than or equal, and less than or equal comparisons. As mentioned earlier, equality and inequality are not meant to be used for Boolean variables.

```

<arithmeticexp> ::= <arithmeticexp> "+" <arithmeticexp> |
                   <arithmeticexp> "-" <arithmeticexp> |
                   <arithmeticexp> "%" <arithmeticexp> |
                   "(" <guard> ")" |
                   <number> |
                   <id> |
                   "true" |
                   "false"

<comparison> ::= <arithmeticexp> "==" <arithmeticexp> |
                 <arithmeticexp> "!=" <arithmeticexp> |
                 <arithmeticexp> ">" <arithmeticexp> |
                 <arithmeticexp> "<" <arithmeticexp> |
                 <arithmeticexp> ">=" <arithmeticexp> |
                 <arithmeticexp> "<=" <arithmeticexp> |

<id> ::= <identifier> |
         <identifier> " '" |
         <identifier> "." <arithmeticexp> |
         <identifier> "." <arithmeticexp> " '"

```

The following is an example of an expression that involves both Boolean and arithmetic operators along with comparisons:

$$\text{forall } q \text{ in } 1..K :: ((s.q \leq r.(q-1) + 1) \ \& \ (s.q \leq r.(q+1) + 1))$$

where s and r are two arrays of integers and K is a constant.

In an expression a (Boolean or integer) variable can appear in four different forms (cf. the `<id>` rule). A variable can be either primed or unprimed. A primed variable is meant to express the *next-state* value of the variable. For example, if x is a Boolean variable, $(x \ \& \ !x')$ expresses a transition where x is true in the source state, but it becomes false in the target state. A variable

may also be indexed or unindexed. For instance, `bg` is unindexed. An indexed variable is one that is declared in an array and is used along with an arithmetic expression as parameter. For instance, `d.(i+1)` is an indexed variable which refers to element `(i+1)` in array `d` where `i` can be a process range, quantifier, or simply a constant. We note that if an indexed variable is primed, its index must appear in parentheses and the prime character must be placed after the right parenthesis. For instance, `b.i'` is a meaningless primed index variable. The right way to write this variable is `b.(i)'`. Another example of a primed indexed variable is `d.(i-1)'`.

Finally, a guarded command ends with a statement, which is a set of deterministic or non-deterministic assignments.

```
<statement> ::= "(" <statement> ")" |
              <statement> "[" <statement> |
              <statement> "," <statement> |
              <statementstruct>
```

```
<statementstruct> ::= <id> ":" <arithmeticexp>
```

Precisely, given an action, a non-deterministic assignment (i.e., `,` operator) stipulates that one assignment is chosen non-deterministically when the guard of the action holds. For example, a guarded command with non-deterministic assignment is as follows:

<code>(b.j) - -> (d.j := 1) [] (d.j := 0) [] (f.j := false) [] (f.j := true);</code>

To the contrary, in a deterministic assignment, all assignments are considered to determine the next state. The following guarded command contains deterministic assignments.

<code>(cs != 2) & (br == 0) - -> (cs := 2), (rr := true);</code>
--

We note that although the SYCRAFT grammar allows a combination of deterministic and non-deterministic assignments, such combinations are not handled during translation of guarded commands to BDDs.

A fault section inside a process expresses the set of fault transitions that the process is subject to. Such fault transitions are modeled using guarded commands. Thus, fault and process actions have exactly the same syntax.

```
<faultdecl> ::= "faults" <identifier> <actions> "endfaults" | <empty>
```

Recall that in addition to each process of a program, the program itself may have a fault section as well. These sections are allowed to be empty, but there must exist at least one nonempty fault section. Otherwise, it means that the program is subject to no faults and, hence, adding fault-tolerance becomes irrelevant.

Finally, we declare the read/restrictions of a process.

```

<rwrestrict> ::= "read:" <rwlist> ";"
                "write:" <rwlist> ";"

<rwlist> ::= <idrange> | <rwlist> "," <idrange>
<idrange> ::= <id> | <identifier> "." <range>

```

For example, the following is a legal declaration of read/write restrictions:

```

read: d.0..N, dg, f.j, b.j;
write: d.j, f.j;

```

2.3 Invariant, Safety Specification, and Prohibited Transitions Declaration

Invariant predicate, safety specification, and prohibited transitions are simply a combination of Boolean and arithmetic expressions. 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.

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). Since safety specification and prohibited transitions model a set of bad transitions that should not occur in any program computation, they typically involve primed variables to model states that should not be reached. The input program and its processes may or may not have prohibited transitions.

```

<invariantdecl> ::= "invariant" <guard> ";"

<specdecl> ::= "specification" <guard> ";" | <empty>

<prohibited> ::= "prohibited" <guard> ";" | <empty>

```

We emphasize that the main difference between transitions specified in **prohibited** and **specification** sections is that transitions in **prohibited** section should not be executed by the program only. In other words, it is perfectly legitimate for a fault action to execute a transition that is specified in the **prohibited** section. To the contrary, transitions specified in the **specification** are not allowed to be executed by both program and fault actions. We typically use prohibited transitions to specify transition that cannot be used in order to add recovery paths.

2.4 Operator Precedence

The precedence of operators and their associativity in SYCRAFT are as follows:

<code>:=</code>	left
<code>^</code>	left
<code>=></code>	left
<code> </code>	left
<code>&</code>	left
<code>==, !=</code>	left
<code>>, <, <=, >=</code>	left
<code>+, -</code>	left
<code>forall, exists</code>	left
<code>%</code>	left
<code>,</code>	left
<code>[]</code>	left
<code>!</code>	left
<code>.</code>	non-associative
<code>'</code>	non-associative

3 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 1 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.

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

```

$ 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...
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 ...
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...
$

```

Figure 1: A sample run snapshot of SYCRAFT.

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.

4 Output Format

The output of SYCRAFT is a fault-tolerant program in terms of its actions. SYCRAFT organizes these actions in three categories. *Unchanged actions* entirely exist in the input program. *Revised actions* exist in the input program, but their guard or statement have been strengthened. 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 also prohibits the program to reach deadlock states from

where safe recovery is not possible.

5 Example 1: Never 7

This example is adapted from M.Sc. thesis of Bastian Braun at University of Mannheim, Germany. Note that in Braun’s thesis the objective is to synthesize a failsafe program, where safe recovery to the program invariant is not required, whereas SYCRAFT synthesizes masking fault-tolerant programs where safe recovery is provided in order to ensure that the synthesized program does not deadlock in the presence of faults. Thus, the our program is slightly different from that developed by Braun.

The program’s state-transition graph is illustrated in Figure 2. This program has 8 states and the safety specification requires that state 7 should never be reached. The invariant predicate of the program is the set $\{0, 1, 2\}$. Program and fault transitions are clear in the figure. It is easy to observe that the program does not reach state 7 in the absence of faults, but it may reach state 7 due to the occurrence of faults. We use SYCRAFT to synthesize a fault-tolerant program, i.e., a program that never reaches state 7 in both absence and presence of faults. For the sake of illustration, we require that potential transitions $(6, 1)$, $(6, 2)$, and $(3, 0)$ should not be used for adding recovery computations.

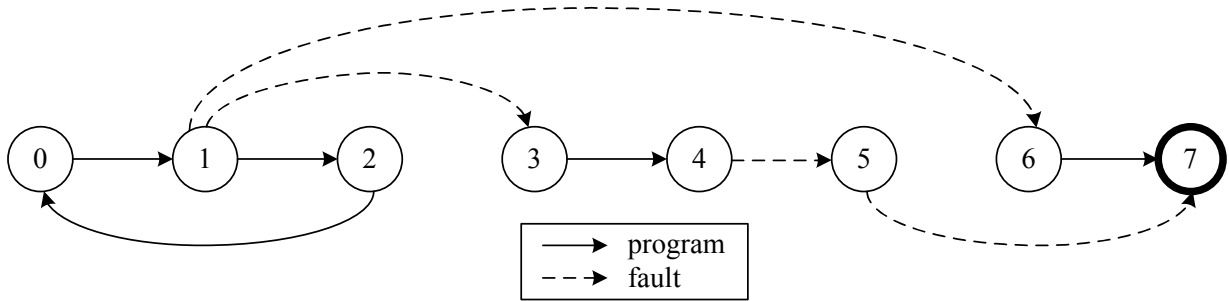


Figure 2: Never7 program state-transition graph.

We now model this program in SYCRAFT’s input language. We use a variable `state` with domain `0..7` which encodes each state shown in Figure 2. We declare and define only one process and this process can read and write the variable `state`, as there is no distribution involved. It is straightforward to see that the guarded commands in `process trans` of Figure 3 model the transitions in Figure 2. Obviously, `process trans` can read and write variable `state` which concludes the definition of `process trans`.

As mentioned earlier, the invariant predicate of `Never7` consists of state in the set $\{0, 1, 2\}$. It is easy to see that the invariant in Figure 3 defines the equivalent expression. The safety specification of our program consists of the simple expression $(state' == 7)$, which essentially includes any transition that ends in state where the value of variable `state` is 7.

Finally, the expression in the `prohibited` section of Figure 3, specifies the transitions that we ruled out for adding recovery, i.e., transitions $(6, 1)$, $(6, 2)$, and $(3, 0)$.

After running SYCRAFT on the input file in Figure 3, the output is a fault-tolerant version of `Never7` shown in Figure 4. As can be seen in this figure, SYCRAFT has removed transitions $(3, 4)$ and $(6, 7)$. Notice that existence of transition $(3, 4)$ may lead the program to state 5 from where occurrence of faults alone violate the safety. Moreover, if the program reaches state 6 due to the occurrence of faults, then transition $(6, 7)$ violates the safety.

```

program never7;

var int state: domain 0..7;
{-----}
process trans
  (state == 0)  -->  state := 1;
  []
  (state == 1)  -->  state := 2;
  []
  (state == 2)  -->  state := 0;
  []
  (state == 3)  -->  state := 4;
  []
  (state == 6)  -->  state := 7;

  fault Malfunction
    (state == 1)  -->  state := 3;
    []
    (state == 1)  -->  state := 6;
    []
    (state == 4)  -->  state := 5;
    []
    (state == 5)  -->  state := 7;
  endfault

  read: state;
  write: state;
endprocess
{-----}
invariant
  (state == 0) | (state == 1) | (state == 2);
{-----}
specification
  (state' == 7);
{-----}
prohibited
  ((state == 6) & ((state' == 1) | (state' == 2)))
  |
  ((state == 3) & (state' == 0));

```

Figure 3: Never 7 program as input to SYCRAFT.

One can also notice that SYCRAFT has added three recovery transitions in order to resolve deadlock states. Notice that these transitions do not intersect with transitions specified in the prohibited section. For the reader's convenience, Figure 5 shows the state-transition graph of the fault-tolerant version of Never7.

```

-----
UNCHANGED ACTIONS:
-----
1- (state == 0)  -->  (state := 1)
2- (state == 1)  -->  (state := 2)
3- (state == 2)  -->  (state := 0)
-----
REVISED ACTIONS:
-----
NEW RECOVERY ACTIONS:
-----
1- (state == 6)  -->  (state := 0)
2- (state == 3)  -->  (state := 2)
3- (state == 3)  -->  (state := 1)
-----

```

Figure 4: SYCRAFT output: fault-tolerant Never7.

6 Example 2: Token Ring

In a token ring program, the processes 0..N are organized in a ring and the token is circulated along the ring in a fixed direction. Each process, say j , maintains a variable x with the domain $\{0, 1, 2\}$, where the value 2 denotes a corrupted value. Process j , where j is greater than zero, has the token iff $x.j$ differs from its successor $x.(j + 1)$ and process N has the token iff $x.N$ is the same as its successor $x.0$. Each process can only write its local variable (i.e., $x.j$). Moreover, a process can only read its own local variable and the variable of its predecessor. In this program, faults can restart at most $N-1$ processes.

The invariant predicate of the token ring problem is determined as follows. Consider a state where a process j has the token. In this state, since no other process has a token, the value of

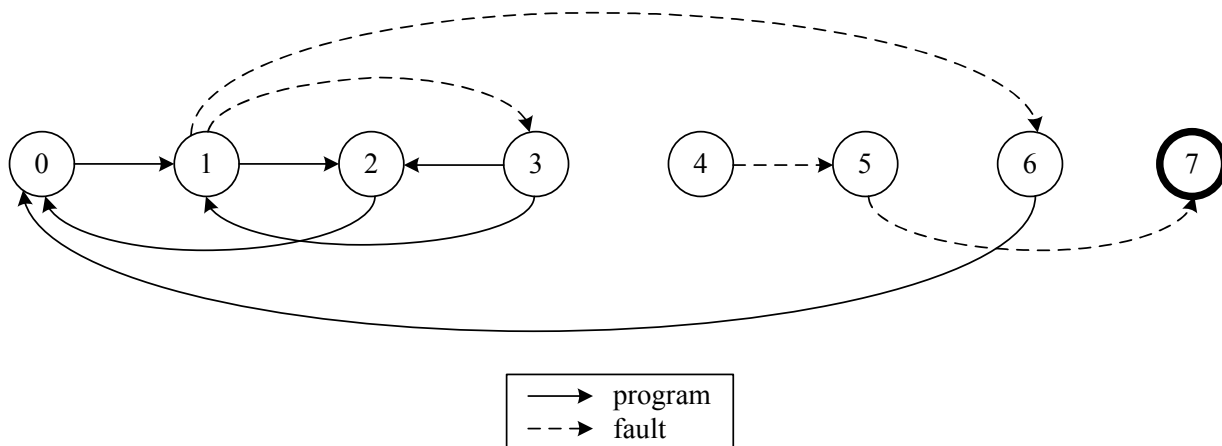


Figure 5: Fault-tolerant Never7 state-transition graph.

variable x of all processes $0..j$ is identical and the x value of all processes $(j+1)..N$ is identical. Letting X denote the string of binary values x_0, x_1, \dots, x_N , we have that X satisfies the regular expression $(0^l . 1^{(N+1-l)} \cup 1^l . 0^{(N+1-l)})$, which denotes a sequence of length $N+1$ consisting of zeros followed by ones or ones followed by zeros.

In token ring processes whose state is uncorrupted are not allowed to copy the value of a corrupted process. Note that in token ring, since this constraint has to be met only by execution of program actions (not fault actions), we specify it under the prohibited section in SYCRAFT.

Figure 6 shows how we model the token ring problem in SYCRAFT. Specifically, we define two

```

program BinaryTokenRing;
const N = 3;
var int x.0..N: domain 0..2;
{-----}
process p
  (x.0 == 0) & (x.N == 0)  -->  x.0 := 1;
  []
  (x.0 == 1) & (x.N == 1)  -->  x.0 := 0;

  read: x.0, x.N;
  write: x.0;
endprocess
{-----}
process q: 1..N
  x.q != x.(q - 1)          -->  x.q := x.(q - 1);

  prohibited (x.0 == 2) & (x.N == x.(0)');

  read: x.q, x.(q - 1);
  write: x.q;
endprocess
{-----}
fault TokenCorruption
  exists i, j in 0..N : (i != j) :: ((x.i != 2) & (x.j != 2))
  -->  (x.0 := 2) [] (x.1 := 2) [] (x.2 := 2) [] (x.3 := 2);
endfault
{-----}
invariant
  ((x.0 == 1) & (x.1 == 0) & (x.2 == 0) & (x.3 == 0)) |
  ((x.0 == 1) & (x.1 == 1) & (x.2 == 0) & (x.3 == 0)) |
  ((x.0 == 1) & (x.1 == 1) & (x.2 == 1) & (x.3 == 0)) |
  ((x.0 == 1) & (x.1 == 1) & (x.2 == 1) & (x.3 == 1)) |
  ((x.0 == 0) & (x.1 == 0) & (x.2 == 0) & (x.3 == 0)) |
  ((x.0 == 0) & (x.1 == 0) & (x.2 == 0) & (x.3 == 1)) |
  ((x.0 == 0) & (x.1 == 0) & (x.2 == 1) & (x.3 == 1)) |
  ((x.0 == 0) & (x.1 == 1) & (x.2 == 1) & (x.3 == 1)) ;
{-----}
prohibited
  exists i in 0..N :: ((x.i != 2) & (x.(i)' == 2));

```

Figure 6: Token ring program as input to SYCRAFT.

processes, p and q . Process p models process 0 in the ring and process q which ranges over $1..N$ models the rest of processes in the ring. Currently, SYCRAFT's grammar does not feature regular expressions. Thus, we have to model them explicitly in SYCRAFT.

An alert reader observes that the above program works correctly in the absence of faults. However, in the presence of faults, it deadlocks due to existence of several tokens or no tokens in the ring. The output of SYCRAFT after adding fault-tolerance to BinaryTokenRing is as follows (cf. Figure 7). Intuitively, a q -process in the synthesized program is allowed to copy the value of its predecessor, if this value is not corrupted. Notice that recovery action process p in the synthesized program stipulate recovery mechanism that starts from outside program invariant and reaches the invariant in one step.

For process $q = 1$

 UNCHANGED ACTIONS:

REVISED ACTIONS:

- 1- $(x0 == 1) \ \& \ (x1 == 2) \ \rightarrow \ (x1 := 1)$
 - 2- $(x0 == 0) \ \& \ (x1 == 2) \ \rightarrow \ (x1 := 0)$
 - 3- $(x0 == 1) \ \& \ (x1 == 0) \ \rightarrow \ (x1 := 1)$
 - 4- $(x0 == 0) \ \& \ (x1 == 1) \ \rightarrow \ (x1 := 0)$
-

NEW RECOVERY ACTIONS:

For process p :

 UNCHANGED ACTIONS:

- 1- $((x0 == 0) \ \& \ (x3 == 0)) \ \rightarrow \ (x0 := 1)$
 - 2- $((x0 == 1) \ \& \ (x3 == 1)) \ \rightarrow \ (x0 := 0)$
-

REVISED ACTIONS:

NEW RECOVERY ACTIONS:

- 1- $(x0 == 2) \ \& \ (x3 == 0) \ \rightarrow \ (x0 := 1)$
 - 2- $(x0 == 2) \ \& \ (x3 == 1) \ \rightarrow \ (x0 := 0)$
-

Figure 7: SYCRAFT output: fault-tolerant token ring.

7 Example 3: Byzantine Agreement

In Byzantine agreement [5], the program consists of a *general* g and three (or more) *non-general* processes: 0, 1, and 2 Figure 8. 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, where process j cannot read $b.k$ for $k \neq j$.

The program works as follows. Each non-general process copies the decision value from the general (Line 12) and then finalizes that value (Line 14). A fault action may turn a process to a Byzantine process, if no other process is Byzantine (Line 16). Faults also change the decision (i.e., variables d and f) of a Byzantine process arbitrarily and non-deterministically (Line 18). As mentioned earlier, 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 24-28). 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 21). 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 12-23).

The following states define the invariant predicate: (1) at most one process may be Byzantine (Line 31), (2) the d value of a non-Byzantine non-general process is either 2 or equal to dg (Line 32), and (3) a non-Byzantine undecided process cannot finalize its decision (Line 33), or, if the general is Byzantine and other processes are non-Byzantine their decisions must be identical and not equal to 2 (Line 35).

The safety specification of Byzantine agreement requires agreement and validity. *Agreement* requires that the final decision of two non-Byzantine processes cannot be different (Lines 37-38). 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 39). Notice that in the presence of a Byzantine process, the program may violate the safety specification.

Similar to other examples, the Byzantine agreement program works fine in the absence of faults. However, it may violate its safety specification (agreement or validity) or deadlock in the presence of faults. Figure 9 shows how SYCRAFT adds fault-tolerance to Byzantine agreement. As can be seen the first action is unchanged, i.e., a non-general process can copy the general's decision under no constraints. However, the second action of the input program is revised. Intuitively, the revised action stipulates that a non-general process can finalize its decision only if it agrees with a majority of non-general processes on its decision. Finally, safe recovery actions of a non-general process of the tolerant program resolves deadlock states by reading the decision of other non-general processes in order to adjust its own decision (recovery actions 6 and 7). The process has also the option of finalizing its decision while recovering (recovery actions 8 and 9).

```

1) program Byzantine_Agreement;
2) const
3)   int N = 2;
4)   int bot = 2;
5) var
6)   boolean bg;
7)   boolean b.0..N;
8)   boolean f.0..N;
9)   int dg: domain 0..1;
10)  int d.0..N: domain 0..2;
    {-----}
11) process j: 0..N
12)   (d.j == bot) & !f.j & !b.j  -->  d.j := dg;
13)   []
14)   (d.j != bot) & !f.j & !b.j  -->  f.j := true;

15)   fault Byzantine_NonGeneral
16)     !bg & (forall p in 0..N:: (!b.p)) -->  b.j := true;
17)     []
18)     b.j -->  (d.j := 1) [] (d.j := 0) [] (f.j := false) [] (f.j := true);
19)   endfault

20)   prohibited
21)     !b.j & !b.(j)' & f.j & ((d.j != d.(j)') | !f.(j)');

22)   read: d.0..N, dg, f.j, b.j;
23)   write: d.j, f.j;
24) endprocess
    {-----}
25) fault Byzantine_General
26)   !bg & (forall p in 0..N:: (!b.p)) -->  bg := true;
27)   []
28)   bg -->  (dg := 1) [] (dg := 0);
29) endfault
    {-----}
30) invariant
31)   (!bg & (forall p, q in 0..N:(p != q):: (!b.p | !b.q)) &
32)   (forall r in 0..N:: (!b.r => ((d.r == bot) | (d.r == dg)))) &
33)   (forall s in 0..N:: ((!b.s & f.s) => (d.s != bot))))
34)   |
35)   (bg & (forall t in 0..N:: (!b.t)) & (forall a,b in 0..N:: ((d.a == d.b) & (d.a != bot))));
    {-----}
36) specification:
37)   (exists p, q in 0..N: (p != q) :: (!b.(p)' & !b.(q)' & (d.(p)' != bot) &
38)   (d.(q)' != bot) & (d.(p)' != d.(q)') & f.(p)' & f.(q)') |
39)   (exists r in 0..N:: (!bg' & !b.(r)' & f.(r)' & (d.(r)' != bot) & (d.(r)' != dg')));

```

Figure 8: The Byzantine agreement problem as input to SYCRAFT.

References

- [1] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

```

-----
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)
-----

```

Figure 9: SYCRAFT output: fault-tolerant Byzantine agreement.

- [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] 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.
- [5] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.