User Manual
A Framework for The Synthesis of Fault-Tolerant Programs

By

ALI EBNENASIR

Michigan State University

Computer Science and Engineering Department

2002

ABSTRACT

A Framework for The Synthesis of Fault-Tolerant Programs

By

ALI EBNENASIR

We present the design and the implementation of a framework for adding fault-tolerance to existing fault-intolerant distributed programs. The input to our framework is an abstract structure of the fault-intolerant program, its specification, and a class of faults that perturbs the program. The output of our framework is the abstract structure of the fault-tolerant program. Our framework also enables one to add new heuristics for adding fault-tolerance. Further, it is possible to change the internal representation of different entities involved in synthesis while reusing the rest of the framework.

We have used this framework for automated synthesis of several fault-tolerant programs including token ring, Byzantine agreement, and agreement in the presence of Byzantine and failstop faults. These examples illustrate that the framework can be used for synthesizing programs that tolerate different types of faults (process restarts, Byzantine and failstop) and programs that are subject to multiple faults (Byzantine and failstop) simultaneously. We also note that our framework has been used for pedagogical purposes.

© Copyright 2002 by Ali Ebnenasir
All Rights Reserved

Acknowledgements

Thanks to Arun Chippada for coding of part of the implementation as his Master's project.

Table of Contents

LIST OF TABLES **viii**

LIST OF FIGURES **ix**

1 Introduction **1**

2 Overview **3**

3 Specifying The Fault-Intolerant Programs **7**

3.1 Syntax and Semantics 7

3.2 The Specification of Agreement Program 12

3.2.1 The Description of The Input File 18

4 Input Translator **22**

4.1 Generating The Program State 23

4.2 Generating The Fault-Intolerant Program 25

4.3 Generating The Faults 28

4.4 Generating The Invariant 29

4.5 Generating The Safety Specification 31

4.6 Generating The Initial States 34

4.7 Generating The Output Generator 35

5 Framework Instantiation **39**

6 Supervised Synthesis **41**

6.1 Applying Heuristics 43

APPENDIX **45**

LIST OF REFERENCES **55**

List of Tables

List of Figures

2.1 The framework architecture. 4

Chapter 1

Introduction

In this manual, we present a framework where the developers of fault-tolerance can synthesize fault-tolerant programs from their fault-intolerant versions. Since the complexity of synthesis is exponential, developers need to apply heuristics during the synthesis to reduce the complexity. Moreover, developers of fault-tolerance may need to use heuristics in different order for different problems.

Our goal is to introduce our framework by describing its input, its output, its design, and a succinct description of its implementation. Towards this end, we first present an overview of our framework so that the reader can realize the architecture of the framework. Second, we describe the way that developers of fault-tolerance can communicate with our framework. To achieve this goal, we explain the syntax and the semantics of the interaction language. Third, we present the object-oriented design of our framework where we present the classes and design patterns that are involved in the organization of the framework. Fourth, we briefly describe the implementation of our framework. Finally, we present some examples that illustrate the use of our framework.

Remark. We assume that the reader is familiar with the theoretical background of our research work based on which we have developed this framework. Otherwise, we

refer the interested reader to [1–4].

Chapter 2

Overview

In this chapter, we present a quick overview of our framework. We also present the architecture of the framework where we show different components and their relationships.

We first introduce the architecture of our framework (cf. Figure 2.1) that shows how it enables the addition of fault-tolerance to an existing fault-intolerant program. Then, we describe the input and the output of our framework. Further, we illustrate how the users can interact with the framework in order to *semi-automatically* synthesize a fault-tolerant program from its fault-intolerant version.

The framework consists of the components that represent the program being synthesized, faults, program invariant, program specification, and the synthesis algorithm. Given a fault-intolerant program and a particular class of faults, the framework generates the reachability graph of the program. Then, the synthesis algorithm iteratively manipulates the initial reachability graph in order to generate the reachability graph of the fault-tolerant program. At each step of the synthesis, the reachability graph represents an intermediate program that is being synthesized.

The synthesis algorithm by itself consists of the following steps: identify ms, identify mt, remove mt, mark invariant, and *ResolveDeadlock*. The set of states ms

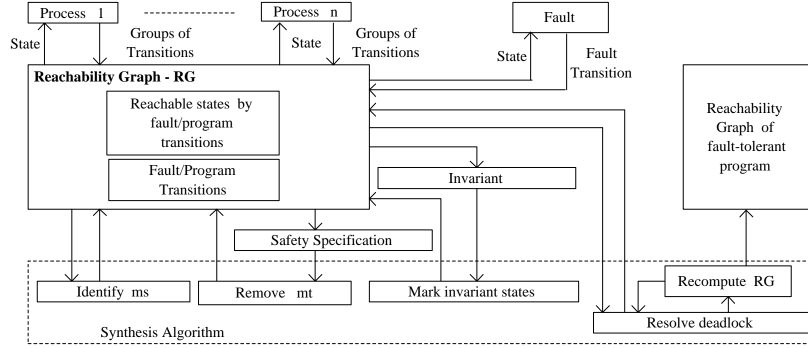

Figure 2.1: The framework architecture.

identifies states from where safety of specification will be violated by one or more fault transitions. The set of transitions mt represents the transitions that reach ms or directly violate safety. After removing mt transitions from the reachability graph, the synthesis algorithm resolves deadlock states (if any). We refer the reader to [4] for further information.

The input/output of the framework. We use Dijkstra's guarded commands [5] as the interaction language between the user and the synthesis framework. A guarded command (action) is of the form $g \rightarrow st$, where g is a state predicate and st is a statement that updates the program variables. The guarded command $g \rightarrow st$ can be executed in a state where g is true; to execute this command, st is executed atomically. In other words, a guarded command includes all program transitions $\{(s_0, s_1) : g$ holds at s_0 and the execution of st at s_0 takes the program to state $s_1\}$.

The faults are also modeled as a set of guarded commands that update program variables. The invariant of the fault-intolerant program is represented as a state predicate. Since we internally identify the safety specification by a *set of transitions* that the fault-tolerant program should not execute (i.e., the set of safety-violating transitions), the safety specification is specified as a set of transitions as well. We divide the specification of safety-violating transitions into three parts: *destination*, and *relation*. In the *destination* part, the users can specify the safety-violating transitions that are identified only by their destination states. In the *relation* part, the users specify the safety-violating transitions that are identified by both their source and

destination states. Finally, the output is also in the guarded command language. (As an example, we present a masking fault-tolerant agreement program in the Appendix that tolerates both failstop and Byzantine faults.)

User interactions. Although the framework can automatically synthesize a fault-tolerant program without user intervention, there are some situations where (i) user intervention can help to speed up the synthesis of fault-tolerant programs, or (ii) a fully automatic approach fails. Hence, our framework permits developers to semi-automatically supervise the synthesis procedure. In such *supervised synthesis*, the fault-tolerance developers interact with the framework and apply their insights during the synthesis. In order to achieve this goal, we have devised some *interaction points* where the developers can stop the synthesis algorithm and query it.

At each interaction point, the users can make the following kinds of *queries*: (i) apply a specific heuristic for a particular task that the framework should try to satisfy and the strategies that it should use to satisfy that predicate); (ii) apply some heuristics in a particular order; (iii) view the incoming program (respectively, fault) transitions to a particular state; (iv) view the outgoing program (respectively, fault) transitions from a particular state; (v) check the membership of a particular state (respectively, transition) to a specific set of states (respectively, transition); e.g., check the membership of a given state s in the set of states ms, and finally (vi) view the intermediate representation of the program that is being synthesized.

The developers of fault-tolerance can use the queries to obtain the current version of the program and choose the additional steps (e.g., resolving deadlock states, resolving non-progress cycles, etc.) that need to be taken for adding fault-tolerance. While we expect that the queries included in this version will be sufficient for a large class of programs, we also provide an alternative for the case where these queries are insufficient. Specifically, in this case, the users of our framework can obtain the corresponding intermediate program in Promela modeling language [6]; this program

can then be checked by SPIN to determine the exact scenario where the intermediate version does not provide the required fault-tolerance. We note that while the code that interprets the counterexamples given by SPIN is not currently implemented, it will be available in the next version of the framework.

Chapter 3

Specifying The Fault-Intolerant Programs

In this chapter, we describe the syntax and the semantics of the input language by which developers of fault-tolerance specify the fault-intolerant program, its invariant, its specification, and the faults. The framework generates the output in the same language.

3.1 Syntax and Semantics

In order to provide the required input for the framework, we have to create a text file with the following structure.

```
1  program programName
2
3   var
4  bool var1;
5  int var2=0,    domain   0 .. 1;
6
7
8
```

```
 9 // The structure of process pName
10    process  pName
11    begin
12   guard1 ->  statement1;
13     |
14   guard2 ->  statement2;
15     |
16     .
17     .
18     .
19     |
20   guardk ->  statementk;
21
22   read  list of variables that pName can read;
23   write list of variables that pName can write;
24   end
25
26 // Faults are represented as a process.
27
28   fault  faultName
29    begin
30   guard1 ->  statement1,
31     |
32     .
33     .
34     .
35     |
36   guardM ->  statementM,
37   end
38
39 // The invariant of the program.
40   invariant
41   A state predicate S in terms of program variables.
```

```
42
43 // The specification of the program is specified in three parts starting
44 // with  specification keyword.
45
46    specification
47
48 // The  destination part identifies a set of states that every
49 // transition that reaches it violates safety.
50
51    destination
52
53    A state predicate that identifies a set of states which the program
54    should not reach.
55
56 // The  relation part identifies a set of transitions that violate safety.
57
58    relation
59
60    A state predicate that identifies a set of transitions that
61    the program is not allowed to execute.
62
63 // The  init section is used for specifying the initial states.
64
65    init
66 // Each initial state is specified using the  state keyword.
67
68    state
69
70    state
```

In the rest of this section, we describe each part of the input file, respectively (cf. Appendix for the grammar of the input language). In the structure of the input file, we have shown all the keywords in *italics*. Also, in this section, we represent

keywords in *italics* form. The input text file consists of the following sections: program *Name*, *variable declaration, process specification, fault specification, the invariant, the specification*, and *the initial states.* We show the structure of the program as the following BNF statement.

(ProgramDeclarator())

(VariableDeclarator())

(ProcessDeclarator())+

(FaultDeclarator())

(Invariant())

(Specification())

(Initialization())

The syntax of program the ProgramDeclarator is as follows.

ProgramDeclarator() : "program" Name()

After the *program* keyword, we write a user-defined name. In the *variable declaration* section, we have to specify all program variables. Each variable should be declared as follows.

varType varName [= initValue], *domain* lb .. ub;

In this version, we only provide Boolean (denoted *bool*) and integer (denoted *int*) types for variables. Also, we have the option to initialize the variables in the declaration. For integer variables, we need to identify the domain of each variable. Since we assume that the domain of each variable is finite, we specify the domain of each integer variable as a *closed* interval from a lower bound to an upper bound (denoted *lowerbound* .. *upperbound*).

To specify a process, we use the following syntax.

process processName *begin* processBody *read* readRestriction;

write writeRestrictions; *end*

The processName is an arbitrary identifier that becomes the name of the process. The syntax of the processBody consists of a set of actions. Each action consists of a guard and a statement (cf. the Appendix for the detail grammar). After the specification of program actions, we have to identify the read/write restrictions imposed on the process. The *read* keyword should follow by a list of variables that the process is allowed to read, and the *write* keyword should follow by a list of variables that the process is allowed to write. In this list of variables, each variable is separated from other variables by a comma. The program may contain more than one process. Thus, we use the above syntax to specify all the program processes.

We specify faults similar to *process specification*. The only difference is that we specify no read (respectively, write) restrictions for faults. Thus, we again use guarded commands to specify faults in the body of a process structure.

We represent the *invariant* of a program as a set of states. Thus, in the input file, we have to write a Boolean condition in the *invariant* section that identifies the set of states that belong to the invariant. The syntax of this section is as follows.

invariant BooleanExpression()

In order to represent the safety specification of the fault-intolerant program, we use the following structure.

specification (*destination* BooleanExpression() — *noDestination*)
(*relation* BooleanExpression() — *noRelation*)

The keyword *specification* shows the start of the specification section. The *destination* section is followed by a Boolean expression that represents a set of states where the program should not reach. The *relation* keyword is followed by a Boolean condition that represents a set of transitions that the program should not execute. If, for a particular problem, the specification does not need a *destination* (respectively, *relation*) section then we just write the keyword *noDestination* (respectively, *noRelation*).

Note that the Boolean expressions that we specify in the *specification* section are in terms of the modified names of the program variables. We attach the letter d to the name of the variables that are involved in the Boolean expression that we write in the *destination* section. Also, since the *relation* section identifies a set of safety-violating transitions in the state space of the program, we need to identify the value of different variables at the source and the destination of the safety-violating transitions. Thus, in the Boolean expression that follows the keyword *relation*, we attach the letter s (respectively, the letter d) to variables where they represent a value at the source (respectively, destination) state. In order to illustrate this issue, we present a concrete example in the next section.

3.2 The Specification of Agreement Program

In this section, we present an example of specifying a fault-intolerant program. Specifically, we show how the developers of fault-tolerance can specify an agreement program that consists of a general process and four non-general processes that are perturbed by both Byzantine and failstop faults. The structure of the input text file is as follows:

```
1  program Byzantine-Failstop
2   var
3  bool bi;
4  bool bj;
5  bool bk;
6  bool bl;
7  bool bg;
8
9  int dg=0,    domain   0 .. 1;
10 int  di,    domain   -1 .. 1;  // (di == -1) means process i has not yet decided.
11 int  dj,    domain   -1 .. 1;
```

```
12   int  dk,    domain   -1 .. 1;
13   int  dl,    domain   -1 .. 1;
14
15   bool  fi;
16   bool  fj;
17   bool  fk;
18   bool  fl;
19
20   bool  upi;
21   bool  upj;
22   bool  upk;
23   bool  upl;
24
25 // The structure of process i.
26    process   i
27    begin
28  ((di == -1) && (fi == 0) && (upi == 0)) ->  di = dg ;
29 |
30 ((di != -1) && (fi == 0) && (upi == 0)) ->  fi = 1 ;
31
32   read di, dj, dk, dl, dg, fi, upi, bi;
33   write di, fi;
34   end
35
36 // The structure of process j.
37    process   j
38    begin
39 ((dj == -1) && (fj == 0) && (upj == 0)) -> dj = dg;
40 |
41 ((dj != -1) && (fj == 0) && (upj == 0)) -> fj = 1;
42
43   read di, dj, dk, dl, dg, fj, upj, bj;
44   write dj, fj;
```

```
45   end
46
47 // The structure of process k.
48    process  k
49    begin
50 ((dk == -1) && (fk == 0) && (upk == 0)) -> dk = dg;
51 |
52 ((dk != -1) && (fk == 0) && (upk == 0)) -> fk = 1;
53
54   read di, dj, dk, dl, dg, fk, upk, bk;
55   write dk, fk;
56   end
57
58 // The structure of process l.
59    process  l
60    begin
61 ((dl == -1) && (fl == 0) && (upl == 0)) -> dl = dg;
62 |
63 ((dl != -1) && (fl == 0) && (upl == 0)) -> fl = 1;
64
65   read di, dj, dk, dl, dg, fl, upl, bl;
66   write dl, fl;
67   end
68
69 // Faults are represented as a process.
70
71   fault  FailstopAndByzantine
72    begin
73 ((upi == 1)&&(upj == 1)&&(upk == 1)&&(upl == 1)) -> upi = 0, upj = 0,
74                                                     upk = 0, upl = 0,
75 |
76 ((bi == 0)&&(bj == 0)&&(bk == 0)&&(bl == 0)&&(bg == 0)) -> bi = 1, bj = 1,
77                                                   bk = 1, bl = 1, bg = 1,
```

```
78 |
79 ((bi == 1)) -> di = 1 , di =0 ,
80 |
81 ((bj == 1)) -> dj = 1 , dj =0 ,
82 |
83 ((bk == 1)) -> dk = 1 , dk =0 ,
84 |
85 ((bl == 1)) -> dl = 1 , dl =0 ,
86 |
87 ((bg == 1)) -> dg = 1 , dg =0 ,
88   end
89
90 // The invariant of the program.
91   invariant
92 ( (
93   ((bg==0) &&
94        (((bi == 1) && (bj == 0)&& (bk == 0)&& (bl == 0)) ||
95         ((bj == 1) && (bi == 0)&& (bk == 0)&& (bl == 0)) ||
96         ((bk == 1) && (bj == 0)&& (bi == 0)&& (bl == 0)) ||
97         ((bl == 1) && (bj == 0)&& (bk == 0)&& (bi == 0)) ||
98         ((bi == 0) && (bj == 0)&& (bk == 0)&& (bl == 0)) ) &&
99         ((bi==1)||(di==-1)||(di==dg))&&
100        ((bj==1)||(dj==-1)||(dj==dg))&&
101        ((bk==1)||(dk==-1)||(dk==dg))&&
102        ((bl==1)||(dl==-1)||(dl==dg))&&
103        ((bi==1)||(fi==0)||(di!=-1) )&&
104        ((bj==1)||(fj==0)||(dj!=-1) )&&
105        ((bk==1)||(fk==0)||(dk!=-1) )&&
106        ((bl==1)||(fl==0)||(dl!=-1) ) ) ||
107
108  ((bg==1)&& (bi==0)&&(bj==0)&&(bk==0)&&(bl==0)&& (
109        ( (((upi == 1) && (upj == 1)&& (upk == 1)&& (upl == 1)))  &&
110                                 ((di==dj)&&(dj==dk)&&(dk==dl)&&(di!=-1)) ) ||
```

```
111            ( (((upi == 1) && (upj == 1)&& (upk == 1)&& (upl == 0)))  &&
112                                                    ((di==dj)&&(dj==dk)&&(di!=-1)) ) ||
113            ( (((upi == 1) && (upj == 1)&& (upk == 0)&& (upl == 1)))  &&
114                                                    ((di==dj)&&(dj==dl)&&(di!=-1)) ) ||
115            ( (((upi == 1) && (upj == 0)&& (upk == 1)&& (upl == 1)))  &&
116                                                    ((di==dk)&&(dk==dl)&&(di!=-1)) ) ||
117            ( (((upi == 0) && (upj == 1)&& (upk == 1)&& (upl == 1)))  &&
118                                                    ((dj==dk)&&(dk==dl)&&(dj!=-1)) )
119            ))
120             )
121            &&
122            (
123            ((upi == 0) && (upj == 1) && (upk ==1) && (upl == 1)) ||
124            ((upi == 1) && (upj == 0) && (upk ==1) && (upl == 1)) ||
125            ((upi == 1) && (upj == 1) && (upk ==0) && (upl == 1)) ||
126            ((upi == 1) && (upj == 1) && (upk ==1) && (upl == 0)) ||
127            ((upi == 1) && (upj == 1) && (upk ==1) && (upl == 1))   ))
128
129 // The specification of the program is specified in three parts starting with
130 //  specification keyword.
131
132   specification
133
134 // The  source part identifies a set of states that every transition
135 // originating at them violates safety.
136
137    noSource
138
139 // The  destination part identifies a set of states that every
140 // transition reaching them violates safety.
141
142   destination
143 (
```

```
144 ( (bid == 0) && (bjd == 0) && (upid == 1) && (upjd == 1) && (did != -1) &&
145                     (djd != -1) && (did != djd) && (fid == 1)  && (fjd == 1)) ||
146 ( (bid == 0) && (bkd == 0) && (upid == 1) && (upkd == 1) && (did != -1) &&
147                     (dkd != -1) && (did != dkd) && (fid == 1) && (fkd == 1)) ||
148 ( (bid == 0) && (bld == 0) && (upid == 1) && (upld == 1) && (did != -1) &&
149                      (dld != -1) && (did != dld) && (fid == 1) && (fld == 1)) ||
150 ( (bjd == 0) && (bkd == 0) && (upkd == 1) && (upjd == 1) && (djd != -1) &&
151                      (dkd != -1) && (djd != dkd) && (fjd == 1) && (fkd == 1)) ||
152 ( (bjd == 0) && (bld == 0) && (upld == 1) && (upjd == 1) && (djd != -1) &&
153                      (dld != -1) && (djd != dld) && (fjd == 1) && (fld == 1)) ||
154 ( (bkd == 0) && (bld == 0) && (upkd == 1) && (upld == 1) && (dkd != -1) &&
155                       (dld != -1) && (dkd != dld) && (fkd == 1) && (fld == 1)) ||
156
157 ((bgd == 0) && (bid == 0)  && (did != -1) && (did != dgd) && (fid == 1)) ||
158 ((bgd == 0) && (bjd == 0)  && (djd != -1) && (djd != dgd) && (fjd == 1)) ||
159 ((bgd == 0) && (bkd == 0)  && (dkd != -1) && (dkd != dgd) && (fkd == 1)) ||
160 ((bgd == 0) && (bld == 0)  && (dld != -1) && (dld != dgd) && (fld == 1))
161 )
162
163 // The  relation part identifies a set of transitions that violate safety.
164
165   relation
166 ((((bis == 0)&& (bid == 0) &&  (fis == 1) && (dis != did))) ||
167             (((bjs == 0) && (bjd == 0) && (fjs == 1) && (djs != djd)))||
168             (((bks == 0) && (bkd == 0) && (fks == 1) && (dks != dkd)))||
169             (((bls == 0) && (bld == 0) && (fls == 1) && (dls != dld)))||
170             (((bis == 0) && (bid == 0) && (fis == 1) && (fid == 0)))||
171             (((bjs == 0) && (bjd == 0) && (fjs == 1) && (fjd == 0)))||
172             (((bks == 0) && (bkd == 0) && (fks == 1) && (fkd == 0)))||
173             (((bls == 0) && (bld == 0) && (fls == 1) && (fld == 0))))
174
175 // The  init section is used for specifying the initial states.
176
```

```
177    init
178
179 // Each initial state is specified using the  state keyword.
180
181    state
182 bi = 0; bj = 0; bk = 0; bl = 0; bg = 0; dg = 0; di = -1; dj = -1; dk = -1; bl = -1;
183 fi = 0; fj = 0; fk = 0; fl = 0; upi = 1; upj = 1; upk = 1; upl = 1;
184
185
186    state
187 bi = 0; bj = 0; bk = 0; bl = 0; bg = 0; dg = 1; di = -1; dj = -1; dk = -1; bl = -1;
188 fi = 0; fj = 0; fk = 0; fl = 0; upi = 1; upj = 1; upk = 1; upl = 1;
189
```

3.2.1 The Description of The Input File

The fault-intolerant agreement program consists of four non-general processes i, j, k, l and a general g. Each non-general process has four variables d, f, b, and up. Variable di represents the decision of a non-general process i, fi denotes whether i has finalized its decision, bi denotes whether i is Byzantine or not, and upi states whether i has failed or not. Process g also has variables dg and bg. We assume that the process g never fails. Thus, the variables of the agreement program are as shown in the *var* section (cf. lines 2-23).

Transitions of the fault-intolerant program. If process i has not copied a value from the general and i has not failed (i.e., upi $= 1$) then i copies the decision of the general (first action in the body of *process* i (cf. line 28)). If i has copied a decision and as a result di is different from -1 then i can finalize its decision if it has not failed (second action in the body of *process* i (cf. line 30)). Other non-general processes (j, k, and l) have a similar structure as shown in the input file (cf. lines 37-67).

Read/Write restrictions. Each non-general process i is allowed to read the following set of variables: $\{di, dj, dk, dl, dg, fi, upi, bi\}$. Thus, i can read the d values of other processes and all its variables. The set of variables that i can write is $\{di, fi\}$. Read/write restrictions of each process are specified in its body after the program actions (using *read* and *write* keywords (e.g., lines 32-33)).

Faults. A Byzantine fault transition can cause a process to become Byzantine if no process is initially Byzantine. A Byzantine process can arbitrarily change its decision (i.e., the value of d). Moreover, the program is subject to failstop faults such that at most one of the non-general processes can be failed, and as a result, it will stop executing any action. The developers of fault-tolerance should specify the faults similar to an independent process that can perturb program variables (cf. lines 71-86).

Invariant. The users of our framework should represent the invariant of the program as a state predicate. In particular, the invariant is a Boolean function (over program variables) that takes a state s and identifies whether s is an invariant state or not.

In the agreement program, the bg variable partitions the invariant into two parts: the set of states s_1 where g is non-Byzantine (cf. line 91), and the set of states s_2 where g is Byzantine (cf. line 106). When g is non-Byzantine, at most one of the non-generals could be Byzantine (cf. lines 92-96). Also, for every non-general process i that is non-Byzantine (i) i has not yet decided or it has copied the value of dg (cf. lines 97-100), and (ii) i has not yet finalized or i has decided (cf. lines 101-104). When g becomes Byzantine, all the non-general processes are non-Byzantine and all the processes that have not failed agree on the same decision (cf. lines 106-116). The invariant of the agreement program stipulates the above conditions on the states in which at most one non-general process has failed (cf. lines 121-125).

Safety specification. The safety specification requires that if g is Byzantine, all

the non-general non-Byzantine processes that have not failed should finalize with the same decision (*agreement*). If g is not Byzantine, then the decision of every finalized non-general non-Byzantine process should be the same as dg (*validity*). Thus, safety is violated if the program executes a transition that satisfies at least one of the conditions specified in the *specification* section of the input file (cf. lines 129-168).

The *specification* section is divided into three parts: *source*, *destination*, and *relation* parts. Intuitively, in the *source* part (cf. line 133), we specify a condition that identifies a set of states s_{source}, where if a transition t originates at s_{source} then t violates the safety of the specification. In the *destination* part (cf. lines 137-156), we write a state predicate that identifies a set of states $s_{destination}$, where if a transition t reaches $s_{destination}$ then t violates safety. In the *relation* part (cf. lines 160-168), we specify a condition that identifies a set of transitions that should not be executed by the program. Note, that we have added a suffix "s" (respectively, suffix "d") to the variable names in the specification section that stands for *source* (respectively, *destination*). Since the relation condition specifies a set of transitions t_{spec} using their source and destination states, we need to distinguish between the value of a specific variable x in the source state of t_{spec} (i.e., xs means the value of x in the source state of t_{spec}) and in the destination state of t_{spec} (i.e., xd means the value of x in the destination state of t_{spec}).

In the case that the program specification does not stipulate any source condition on safety-violating transitions, we leave the source section empty with the keyword *noSource* (cf. line 133). (We use similar keywords *noDestination* and *noRelation* for the cases where we do not have destination or relation conditions in the specification, respectively.)

Initial states. The keyword *init* (cf. line 172) identifies the section of the input file where the user has to specify some initial states. These initial states should belong to the invariant. For each initial state, the user should use the reserved word *state* (cf.

line 176). In the *state* section (cf. lines 176-178 and 181-183), the user should assign some values to the program variables that belong to their corresponding domain.

Chapter 4

Input Translator

In this chapter, we illustrate how we automatically translate the input text file to a set of Java files that are integrated into the framework. Towards this end, we describe the relation between each section of the input file (described in Chapter 3) and the corresponding data structure that we generate.

Depending on the problem at hand, developers of fault-tolerance should instantiate an instance of our framework that is integrated with a set of problem-dependent Java files. In other words, to synthesis a fault-tolerant program from its fault-intolerant version, we need to give the fault-intolerant program to our framework so that we can use the core of the framework for the synthesis of the fault-tolerant program. Towards this end, we translate the abstract structure of the fault-intolerant program (specified in guarded command) to a set of Java classes that we can integrate into our framework and generate an instance of the framework that enables us to synthesis the corresponding fault-tolerant program.

In order to translate the input text file to the corresponding Java files, developers of fault-tolerance should use our parser on the command line as follows.

java MyParser *inputFilename*

The execution of the above command generates the required java files namely,

State.java, ProgramImplementation.java, Fault.java, Invariant.java, SafetySpecification.java, InitialStates.java, Tool.java, and a set of Java files corresponding to each process with the same name given to the process in the input file.

In Section 4.1, we illustrate how we generate a Java file that represents each state of the program. Then, in Section 4.2 we introduce the structure of the program and its processes. We present the data structure for the internal representation of faults in Section 4.3. In Sections 4.4 and 4.5, we describe the generation of the Java files that correspond to the invariant and the safety specification. Finally, we generate a Java file that is responsible for the generation of the output fault-tolerant program.

4.1 Generating The Program State

In principle, we identify each program state by a specific valuation to the program variables. In practice, we generate a Java file that specifies a class State. The State class includes all the program variables as its state variables. Moreover, the State class has extra state variables to store more information about a state. For example, we have a Boolean variable invariant in the State class that shows if an instance of the State class is an invariant state. We present a partial structure of the State class as follows.

```
1 public class State {
2
3  int stateno;
4          int vars[];
5          int no_vars;
6          State next;
7          LinkedList out_ptransitions;
8          LinkedList in_ptransitions;
9          LinkedList out_ftransitions;
10         LinkedList in_ftransitions;
```

```
11          boolean invariant;
12          boolean ms;
13
14 public int getValue (int i)  {
15          if( (i < no_vars) && (i >= 0) )  {
16                   return vars[i];   }
17          else  {
18                   System.out.println("Erroneous variable no" );  }
19          return Parameters.INVALID_VALUE;
20    }
21
22
23 public void setValue (int i, int v) {
24          if( (i < no_vars) && (i >= 0) )  {
25                   vars[i] = v;    }
26          else   {
27                   System.out.println("Erroneous variable no" ); }
28  }
29
30 public void markInvariant() {
31          invariant = true;
32   }
33
34 public boolean isInvariant() {
35          return invariant;
36    }
37
38 public boolean is_ms() {
39          return ms;
40    }
41
42 public void set_ms()  {
43          ms = true;
```

```
44    }
45
46 public LinkedList getInFaultTransitions() {
47            return in_ftransitions;
48    }
49
50 public LinkedList getOutFaultTransitions() {
51            return out_ftransitions;
52  }
53
54 public LinkedList getInProgramTransitions() {
55            return in_ptransitions;
56    }
57
58 public LinkedList getOutProgramTransitions() {
59            return out_ptransitions;
60    }
```

As we can observe in the State class, we have a set of methods for manipulating the state variables of the State class. Also, we have a set of methods for extracting information about an instance of the State class (e.g., isInvariant() method). Each instance of the State class has four sets of transitions: input program transitions, input fault transitions, output program transitions, and output fault transitions. Observe that there exist some methods for the manipulation of these transitions.

4.2 Generating The Fault-Intolerant Program

We describe the automatic generation of the ProgramImplementation class. This class models the abstract structure of the fault-intolerant program. Accompanied with the generation of this class, we generate Java classes corresponding to each process of the fault-intolerant program. In this subsection, we first describe the structure of

the ProgramImplementation class, and then we describe the structure of the Java files that are created for each process.

```
1
2 public class ProgramImplementation implements Program_Implementor {
3
4          Component cmpts[];
5          int no_components;
6          public ProgramImplementation(){
7          no_components = ProblemSpecific.NO_COMPONENTS;
8          cmpts = new Component[no_components];
9
10
11        // Create the internal representation of the first process
12         try {
13
14                    }
15         catch(Exception e)    {
16 System.out.println(" Exception in constructing Process i " + e);       }
17
18
19
20      // Create the internal representation of the second process
21         try {
22
23                    }
24         catch(Exception e)    {
25 System.out.println(" Exception in constructing Process i " + e);  }
26
27
28
29      . . .
30      . . .
```

```
31
32
33 }
34
35
36 public Stack exploreImp( State s, Hashtable states)   {
37         Stack ns = new Stack();
38         for(int i = 0; i < no_components; i++)    {
39                  cmpts[i].explore(ns,s,states);   }
40                  return ns;
41              }
42
43
44 public boolean solveDeadlockImp( State s, Invariant inv, Hashtable states,
45 SafetySpecification spec, Program fitp )    {
46         for(int i = 0; i < no_components; i++)    {
47                  if( cmpts[i].solveDeadlock( s, inv, states, spec, fitp ) )
48  return true;  }
49         return false;     }
50
51
52 public boolean solveDeadlockMoreImp( State s, Invariant inv, Hashtable states,
53 SafetySpecification spec, Program fitp, Hashtable recStates ) {
54         for(int i = 0; i < no_components; i++)  {
55         if( cmpts[i].solveDeadlockMore( s, inv, states, spec, fitp, recStates ) )
56   return true;   }
57         return false;    }
58
59 public boolean producesTransitionImp(Transition t)   {
60         for(int i = 0; i < no_components; i++)   {
61                  if( cmpts[i].producesTransition(t) )    return true;    }
62         return false;     }
63
```

```
64 public boolean isDeadlockedImp(State s)   {
65         for(int i = 0; i < no_components; i++)    {
66                 if( !cmpts[i].isDeadlocked(s) )    return false;   }
67         return true;    }
68
69
70 public void printImp()   {
71         System.out.println("Printing Program.....");
72         System.out.println();
73         for(int i = 0; i < no_components; i++)  {
74                 System.out.println("Printing component" + i);
75                 cmpts[i].print();   }
76           }
77 }
```

The ProgramImplementation class implements the interface defined by the Bridge design pattern applied on the abstract class Program. This way, the implementation of the program structure remains independent of the abstract design of the framework. The ProgramImplementation class has three sections. In this first section, we declare the state variables of the class. In the second section, we create a try block corresponding to each process of the fault-intolerant program (cf. the documentation of the code for the details of this section.). In the third section, we have a set of methods that we use for the generation of the reachability graph and resolution of the deadlock states.

4.3 Generating The Faults

The structure of the Fault class is very simple. The constructor of the Fault class creates a linked list structure of the fault actions. The input translator converts the fault actions represented as a process in the input text file to a lined list of actions

in the Fault class. Then, the synthesis framework uses these fault actions to generate the reachability graph.

```
1 public class Fault {
2
3 LinkedList actions;
4
5 public Fault()   {
6
7          actions = new LinkedList();
8
9         .  .  .
10
11                  }
12
13  Stack explore( State s, Hashtable states) {
14         Stack ns = new Stack();
15         ListIterator i = actions.listIterator(0);
16         while( i.hasNext() )  {
17                 .  .  .
18                 }
19                 return ns;
20  }
21
22  public void print()  {            }
23 }
```

4.4 Generating The Invariant

Using the *invariantCondition* specified in the input text file and program variables, the input translator creates the following Java class.

```
1 public class Invariant {
2
3 public Invariant()   {  }
4
5 public boolean satisfies( State s ) {
6
7          int bi;
8          int bj;
9          int bk;
10         int bg;
11         int dg;
12         int di;
13         int dj;
14         int dk;
15         int fi;
16         int fj;
17         int fk;
18
19
20         bi = s.getValue(0);
21         bj = s.getValue(1);
22         bk = s.getValue(2);
23         bg = s.getValue(3);
24         dg = s.getValue(4);
25         di = s.getValue(5);
26         dj = s.getValue(6);
27         dk = s.getValue(7);
28         fi = s.getValue(8);
29         fj = s.getValue(9);
30         fk = s.getValue(10);
31
32  if (  invariantCondition )
33                   return true;
```

```
34
35 return false;
36  }
37 }
```

The constructor of this Invariant class is empty. The input translator, declares all program variables in the satisfies method. Then, using the value of the variables in the parameter state of the satisfies method, the membership of the given state to the invariant is determined. As an example, we have shown in the above Java code the structure of the Invariant class that is generated for the Byzantine agreement program introduced in Section 3.2. The *invariantCondition* specified in the if statement is exactly the same condition specified in the input file in the *invariant* section.

4.5 Generating The Safety Specification

The structure of the SafetySpecification class is very similar to the structure of the Invariant class. The only difference is that in the SafetySpecification class, we need to check the validity of a given transition instead of checking the membership of a given state to the invariant.

```
1 public class SafetySpecification {
2
3 LinkedList predicate_list;
4
5 public SafetySpecification()   {  }
6
7 public boolean violatesSafety( Transition t ) {
8          State source = t.getSource();
9          State dest = t.getDestination();
10         int bis;
11         int bjs;
```

```
12          int bks;
13          int bgs;
14          int dgs;
15          int dis;
16          int djs;
17          int dks;
18          int fis;
19          int fjs;
20          int fks;
21
22
23          int bid;
24          int bjd;
25          int bkd;
26          int bgd;
27          int dgd;
28          int did;
29          int djd;
30          int dkd;
31          int fid;
32          int fjd;
33          int fkd;
34
35
36          bis = source.getValue(0);
37          bjs = source.getValue(1);
38          bks = source.getValue(2);
39          bgs = source.getValue(3);
40          dgs = source.getValue(4);
41          dis = source.getValue(5);
42          djs = source.getValue(6);
43          dks = source.getValue(7);
44          fis = source.getValue(8);
```

```
45          fjs = source.getValue(9);
46          fks = source.getValue(10);
47
48
49          bid = dest.getValue(0);
50          bjd = dest.getValue(1);
51          bkd = dest.getValue(2);
52          bgd = dest.getValue(3);
53          dgd = dest.getValue(4);
54          did = dest.getValue(5);
55          djd = dest.getValue(6);
56          dkd = dest.getValue(7);
57          fid = dest.getValue(8);
58          fjd = dest.getValue(9);
59          fkd = dest.getValue(10);
60
61
62  if (  destinationCondition )          return true;
63
64  if ( relationCondition )                   return true;
65
66  return false;
67   }
68
69 public void print()  {          }
70  }
```

As an example, we have shown in the above Java code the structure of the SafetySpecification class that is generated for the Byzantine agreement program introduced in Section 3.2. The *destinationCondition* and *relationCondition* specified in the if statements are exactly the same condition specified in the input file in the *specification* section.

4.6 Generating The Initial States

In this subsection, we describe the automatic generation of the InitialStates class from input file. In this class, the input translator generates Java code for the creation of a linked list of initial states. The input translator uses the values of the variables from the input file in order to generate code for the instantiation of initial states.

```
1 public class InitialStates {
2
3          public InitialStates() { }
4
5         static LinkedList getInputStates()  {
6                  State s0 = new State();
7                          s0.setValue(0,0);
8                          s0.setValue(1,0);
9                          s0.setValue(2,0);
10                         s0.setValue(3,0);
11                         s0.setValue(4,0);
12                         s0.setValue(5,-1);
13                         s0.setValue(6,-1);
14                         s0.setValue(7,-1);
15                         s0.setValue(8,0);
16                         s0.setValue(9,0);
17                         s0.setValue(10,0);
18
19                 State s1 = new State();
20                         s1.setValue(0,0);
21                         s1.setValue(1,0);
22                         s1.setValue(2,0);
23                         s1.setValue(3,0);
24                         s1.setValue(4,1);
25                         s1.setValue(5,-1);
26                         s1.setValue(6,-1);
```

```
27                          s1.setValue(7,-1);
28                          s1.setValue(8,0);
29                          s1.setValue(9,0);
30                          s1.setValue(10,0);
31
32 LinkedList inputstates = new LinkedList();
33                           inputstates.add(s1);
34                           inputstates.add(s0);
35                           return inputstates;
36      }
37 }
```

Observe that the getInputStates returns a linked list of states that are used for the expansion of the reachability graph.

4.7 Generating The Output Generator

After the synthesis framework synthesizes a fault-tolerant program, it has to transform the reachability graph of the fault-tolerant program to guarded commands that are understandable for the user. To achieve this goal, the input translator automatically generates the OutputGenerator class in Java code.

```
1 public class OutputGenerator {
2
3          public OutputGenerator() {     }
4
5 static void PrintProcess_i(AbstractReachabilityGraph g,
6                               String action, OutputFile outf){
7                                 . . .
8                 }
9 static void PrintProcess_j(AbstractReachabilityGraph g,
10                                         String action, OutputFile outf){
11                                . . .
```

```
12              }
13 static void PrintProcess_k(AbstractReachabilityGraph g,
14                                     String action, OutputFile outf){
15                              . . .
16              }
17
18 static public void printFIProgram(AbstractReachabilityGraph g,OutputFile of){
19          of.writeLine("No. of states: "+g.getNumStates());
20          of.writeLine(" ");
21          of.writeLine("*******  The fault-intolerant program  ********");
22          of.writeLine(" ");
23          of.writeLine("----------  The actions of Process i ----------");
24          of.writeLine(" ");
25          PrintProcess_i(g,"set_di_val0",of);
26          of.writeLine(" ");
27          PrintProcess_i(g,"set_di_val1",of);
28          of.writeLine(" ");
29          PrintProcess_i(g,"set_fi_val1",of);
30          of.writeLine(" ");
31          of.writeLine("----------  The actions of Process j ----------");
32          of.writeLine(" ");
33          PrintProcess_j(g,"set_dj_val0",of);
34          of.writeLine(" ");
35          PrintProcess_j(g,"set_dj_val1",of);
36          of.writeLine(" ");
37          PrintProcess_j(g,"set_fj_val1",of);
38          of.writeLine(" ");
39          of.writeLine("----------  The actions of Process k ----------");
40          of.writeLine(" ");
41          PrintProcess_k(g,"set_dk_val0",of);
42          of.writeLine(" ");
43          PrintProcess_k(g,"set_dk_val1",of);
44          of.writeLine(" ");
```

```
45          PrintProcess_k(g,"set_fk_val1",of);
46          of.writeLine(" ");
47          }
48
49 static public void printFTProgram(AbstractReachabilityGraph g,OutputFile of){
50          of.writeLine("No. of states: "+g.getNumStates());
51          of.writeLine(" ");
52          of.writeLine("*******  The fault-tolerant program  ********");
53          of.writeLine(" ");
54          of.writeLine("----------  The actions of Process i ----------");
55          of.writeLine(" ");
56          PrintProcess_i(g,"set_di_val0",of);
57          of.writeLine(" ");
58          PrintProcess_i(g,"set_di_val1",of);
59          of.writeLine(" ");
60          PrintProcess_i(g,"set_fi_val1",of);
61          of.writeLine(" ");
62          of.writeLine("----------  The actions of Process j ----------");
63          of.writeLine(" ");
64          PrintProcess_j(g,"set_dj_val0",of);
65          of.writeLine(" ");
66          PrintProcess_j(g,"set_dj_val1",of);
67          of.writeLine(" ");
68          PrintProcess_j(g,"set_fj_val1",of);
69          of.writeLine(" ");
70          of.writeLine("----------  The actions of Process k ----------");
71          of.writeLine(" ");
72          PrintProcess_k(g,"set_dk_val0",of);
73          of.writeLine(" ");
74          PrintProcess_k(g,"set_dk_val1",of);
75          of.writeLine(" ");
76          PrintProcess_k(g,"set_fk_val1",of);
77          of.writeLine(" ");
```

```
78              }
79
80  }
```

The structure of the OutputGenerator class consists of two parts. First, we have a section that includes a method corresponding to each process. For example, for the Byzantine agreement program, the input translator generates three methods corresponding to non-general processes i, j, and k (cf. Section 3.2).

In the second part of the code of the OutputGenerator class, the input translator generates two methods respectively for generating the fault-intolerant and the fault-tolerant programs. In each of these section, the methods specified in the first part are invoked to generate the guarded command of each process.

Chapter 5

Framework Instantiation

In this chapter, we show how developers can instantiate an instance of the framework for the synthesis of a specific program. Towards this end, we present the procedure by which the instantiation takes place.

After the translation of the input file to a set of Java files, we need to copy the generated Java files to the same folder where the core of the framework exists. At this step, first we have to compile the java files corresponding to each process. For example, in the case of Byzantine agreement we do the following.

$>$ javac i.java

$>$ javac j.java

$>$ javac k.java

Afterwards, we compile `Tool.java` file that is the main file for the instantiation of the framework.

$>$ javac Tool.java

The compilation of `Tool.java` file results in the compilation of the core of the framework. Afterwards, we are ready to instantiate the framework for the synthesis of the fault-tolerant version of the fault-intolerant program specified in the input text file. We run the following command.

```
> java Tool
```

After running the above command, developers will be prompted by the command line of the framework. In the next section, we describe the way that developers can supervise the synthesis of the fault-tolerant program.

Chapter 6

Supervised Synthesis

In this section, we describe the capabilities of our framework by which developers of fault-tolerance can direct the synthesis of fault-tolerant programs where automatic synthesis fails. Towards this end, we describe the user interface of the framework and the steps of the synthesis.

The main menu of the framework is as follows:

A: Automatic

S: Semi-automatic

T: Terminate

SYNFT:>

There exist three options namely *Automatic, Semi-automatic*, and *Terminate*. The first option means that the synthesis will be performed automatically and developers have no option for the intervention during the synthesis. The *Terminate* option terminates the execution of the framework. And, the SYNFT:¿ is the command line of the framework. The *Semi-automatic* option provides the means for developers to conduct the synthesis of the fault-tolerant program. When developers choose this option the following menu appears.

E: Expand reachability graph.

I: Identify states from where safety of specification is violated by fault transitions.

R: Remove safety violating transitions.

M: Mark the invariant states.

S: Solve deadlock states.

Q: Query.

The first option asks the framework to generate the internal representation of the fault-intolerant program as a directed graph; i.e., *reachability* graph. Notice that developers have to perform the first state before moving to the subsequent steps of the synthesis. After the generation of the reachability graph, it becomes possible to choose other option in the desired order. However, in the current version of the framework, the synthesis algorithm is a deterministic implementation of the non-deterministic algorithm presented in [1]. Thus, it is suggested to perform the rest of the options in order.

The option M results in the identification of the invariant states. The option I asks the synthesis algorithm to identify the set of states (i.e., denoted ms) from where safety of specification is violated directly by fault transitions. The identification of ms states results in the identification of a set of transitions (i.e., denoted mt) that reach a state in ms or directly violated safety. The option R asks the synthesis algorithm to remove mt transitions. The removal of mt transitions may create states that do not have any outgoing program transitions (i.e., deadlock states). The option S asks the framework to apply existing heuristics for resolving deadlock states. The complexity of synthesis mostly stems from deadlock resolution.

In every subsequent step after expanding the reachability graph developers have the option to explore the reachability graph in order to understand the cause of failures of the synthesis. By selecting the option Q, developers can query the framework for particular state or transition. In the case of making a query on the states of the reachability graph, the following menu appears.

1) View the incoming program transitions.

2) View the incoming fault transitions.

3) View the outgoing program transitions.

4) View the outgoing fault transitions.

5) Is this state an *ms* state?

6) Is this state an invariant state?

7) Return to the main menu.

Observe that the above menu allows developers to walk through the reachability graph in the cases where the state space is manageable. Otherwise, there exists an option to model the program being synthesized in Promela modeling language and take advantage of the SPIN model checker. Using model checkers allows developers of fault-tolerance to analyze the behaviors of the intermediate program when the synthesis of the fault-tolerant program fails. The counter examples generated by SPIN reflects the shortcomings of the existing heuristics in the synthesis of the fault-tolerant program.

6.1 Applying Heuristics

In this section, we show how developers of fault-tolerance can select suitable heuristics where they need to resolve deadlock states. The deadlock resolution step follows after the removal of safety-violating transitions.

In the main menu of the framework, there exists an option related to deadlock resolution (i.e., the option S). Upon selection of the option S, the following menu appears:

1: Heuristic1.

2: Heuristic2.

3: Return.

The above menu provides the choice of selecting one of the existing heuristics for deadlock resolution. Developers of fault-tolerance can select one of the existing heuristics during the synthesis. The number of existing heuristics depends on the heuristics that are currently integrated into the framework. We note that as developers design new heuristics, they can integrate them into the framework and add new options to the above menu in GraphImplementation2.java file. The integration of new heuristics is fairly simple. We refer the interested readers to [4] for the details of integrating new heuristics to the framework..

APPENDIX

```
1
2 /* Copyright (C) 2002 Ali Ebnenasir.
3  *
4  * Do not modify without amending copyright; distribute freely but retain
5  * copyright message.
6  *
7  * Java files generated by running JavaCC on this file (or modified versions
8  * of this file) may be used in exactly the same manner as this file.
9  *
10 * Author: Ali Ebnenasir
11 * Date: 9/29/02
12 *
13 * This file contains a grammar for specifying the input file of the SYNFT.
14 * In the input file, the user should specify a fault-intolerant program,
15 * its invariant, its specification, initial states, and faults.
16 *
17 */
18
19
20 TOKEN : /* RESERVED WORDS AND LITERALS */
21
22   < PROGRAM: "program" >
23 | < VAR: "var" >
24 | < DOMAIN: "domain" >
25 | < PROCESS: "process" >
26 | < BEGIN: "begin" >
27 | < END: "end" >
28 | < READ: "read" >
29 | < WRITE: "write" >
30 | < BOOLEAN: "boolean" >
31 | < INT: "int" >
32 | < FALSE: "false" >
33 | < TRUE: "true" >
```

```
34 | < FAULT: "fault" >
35 | < INVARIANT: "invariant" >
36 | < SPECIFICATION: "specification" >
37 | < SOURCE: "source" >
38 | < DESTINATION: "destination" >
39 | < RELATION: "relation" >
40 | < INIT: "init" >
41 | < STATE: "state" >
42 | < NOSOURCE: "noSource" >
43 | < NODESTINATION: "noDestination" >
44 | < NORELATION: "noRelation" >
45
46
47 TOKEN : /* LITERALS */
48
49   < INTEGER_LITERAL:
50          <DECIMAL_LITERAL>
51        | <HEX_LITERAL>
52   >
53 |
54   < #DECIMAL_LITERAL>
55 |
56   < #HEX_LITERAL>
57
58
59
60 TOKEN : /* SEPARATORS */
61
62   < LPAREN: "(" >
63 | < RPAREN: ")" >
64 | < SEMICOLON: ";" >
65 | < COMMA: "," >
66 | < DOT: "." >
```

```
67 | < DOUBLEDOT: ".." >
68 | < ARROW: "->" >
69 | < BAR: "|" >
70
71
72
73
74 TOKEN : /* OPERATORS */
75
76   < ASSIGN: "=" >
77 | < GT: ">" >
78 | < LT: "<" >
79 | < BANG: "!" >
80 | < EQ: "==" >
81 | < LE: "<=" >
82 | < GE: ">=" >
83 | < NE: "!=" >
84 | < AND: "&&" >
85 | < PLUS: "+" >
86 | < MINUS: "-" >
87 | < REM: "%" >
88 | < TILDA: "~" >
89 | < OR: "||" >
90
91
92
93 /**********************************************
94  *      THE LANGUAGE GRAMMAR STARTS HERE       *
95  **********************************************/
96
97 /*
98  * Program syntax follows.
99  */
```

```
100
101 InputProgram() :
102   ( ProgramDeclarator() )
103   ( VariableDeclarator() )
104   ( ProcessDeclarator() )+
105   (FaultDeclarator())
106   (Invariant())
107   (Specification())
108   (Initialization())
109
110
111 ProgramDeclarator() :  "program"  Name()
112
113
114
115 VariableDeclarator() : "var"  VarDeclaration()
116
117
118
119 VarDeclaration() :
120                 ( BooleanDeclaration() | IntegerDeclaration())+
121
122
123 BooleanDeclaration() :
124                  "boolean"   VariableId()      [ "="  Literal() ] ";"
125
126
127
128
129  IntegerDeclaration() :
130                 "int"    VariableId()  [ "="  Literal()] ","
131
132                      DomainDeclarator()  ";"
```

```
133
134
135
136
137 VariableId() :
138    <IDENTIFIER> ( "[" "]" )*
139
140
141
142 DomainDeclarator()  :
143   "domain"  ( Literal() ".."  Literal() )
144             ( "~" Literal() ".."  Literal() )*
145
146
147 Literal() :
148 <INTEGER_LITERAL>
149 |
150  BooleanLiteral()
151
152
153
154
155 BooleanLiteral() :
156   "true"   |    "false"
157
158
159
160
161 /*
162  * Process syntax follows.
163  */
164
165 ProcessDeclarator() :
```

```
166 "process"   Name()
167 "begin"   ProcessBody()         RWRestrictions()  "end"
168
169
170 ProcessBody() :
171   Action() ("|"  Action())*
172
173
174
175 /*
176  * Fault syntax follows.
177  */
178
179 FaultDeclarator() :
180 "fault"   Name()
181 "begin"   FaultBody()         "end"
182
183
184
185 FaultBody() :
186   Action() ("|"  Action())*
187
188
189 /*
190  * Invariant syntax should be in DNF form.
191  */
192
193 Invariant():
194 "invariant"  BooleanExpression()
195
196
197
198
```

```
199 /*
200  * Specification syntax consists of two parts: source states and destination states.
201  */
202
203 Specification():
204 "specification" ("destination"  BooleanExpression() | "noDestination" )
205 ("relation" BooleanExpression()
206   | "noRelation" )
207
208
209
210 /*
211  * Initialization syntax consists of variables value assignment.
212  */
213
214
215
216 Initialization():
217     "init" ("state" ( InState() ) )*
218
219
220 InState():
221 ( Statement()  ";" )+
222
223 Action Action() :
224  BooleanExpression()  "->"
225  ( Statement()  (t =";" | t=",") )+
226
227
228
229
230
231
```

```
232  RWRestrictions() :
233 "read"  NameList()  ";"  "write"  NameList()   ";"
234
235
236 NameList() :
237          Name()  ( ","  Name()  )*
238
239
240 Name() :
241    <IDENTIFIER>
242
243
244
245 Guard() :
246 ( Conjunction()) ("||"Conjunction())*
247
248
249
250
251 term():
252
253 LOOKAHEAD(2)   Comparison()
254 ( LOOKAHEAD(2) ("&&" | "||")
255 Comparison() )* |
256 "("   BooleanExpression(expr)  ")"
257
258
259
260 BooleanExpression() :
261 term() (LOOKAHEAD(2) ("&&" | "||") term() )*
262
263
264
```

```
265 Conjunction() :
266 "("     Comparison()  ("&&" comp = Comparison() )*
267 ")"
268
269
270
271 Disjunction() :
272 "("   Comparison()   ("||"  Comparison()  )* ")"
273
274
275 Comparison():
276 "(" (( Literal()  |  Name() )   ComparisonOperators()
277 ( Literal()  |  Name() )  )  ")"
278
279
280 ComparisonOperators() :
281 "==" |  "!=" |  "<"  |
282  ">" |  "<=" |  ">="
283
284
285 Statement() :
286 Name()   "="  (  Literal() |  Name() )
```

Bibliography

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

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

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

[4] Sandeep S. Kulkarni and Ali Ebnenasir. A framework for automatic synthesis of fault-tolerance. Technical Report MSU-CSE-03-16, Computer Science and Engineering, Michigan State University, East Lansing MI 48824, Michigan, July 2003.

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

[6] Spin language reference. `http://spinroot.com/spin/Man/promela.html`.