FCPre: Extending the Arora-Kulkarni Method of Automatic Addition of Fault-Tolerance

Bastian Braun
University of Hamburg
braun@informatik.uni-hamburg.de

Abstract

*Synthesizing fault-tolerant systems from fault-*in*tolerant systems simplifies design of fault-tolerance. Arora and Kulkarni developed a method and a tool to synthesize fault-tolerance under the assumption that specifications are not history-dependent (fusion-closed). Later, Gärtner and Jhumka removed this assumption by presenting a modular extension of the Arora-Kulkarni method. This paper presents an implementation of the Gärtner-Jhumka method which is evaluated on several examples. As additional safety net, we have added automatic verification of the results using the model checker Spin. In the context of this work, a fault in the Gärtner-Jhumka method has been found. Though this fault is rare and does not cause incorrect results, there might be no result at all.*

1. Introduction

The design of fault-tolerant systems usually follows three steps: In the first step, designers fix the *fault-assumption*, i.e., the type and the amount of faults that the system should tolerate. In the second step, mechanisms are developed that detect and handle the faults in the system once they occur. Finally, in the third step, the mechanisms together with the integration into the functional system have to be validated to check that under all circumstances covered by the fault-assumption the system satisfies its specification.

It is well-known, that designing fault-tolerant systems is not a trivial task. All three steps can potentially lead to incorrect and fault-*in*tolerant systems. For example, the choice of the fault-assumption in practice is more an art than a craft: On the one hand, underestimating the criticality of possible faults as well as their likelihood may lead to unexpected system failures. On the other hand, overestimating the faults wastes resources and makes the fault-tolerance mechanisms more complicated (and error prone) than necessary.

One way to improve this situation is to use automated methods to construct fault-tolerant systems. Roughly speaking, the designer starts with a program that has been designed without fault-tolerance in mind, specifies the fault-assumption and presses a button: The outcome is a "fault-tolerant version" of the original program that now transparently tolerates the assumed faults. This is the idea which also underlies generic fault-tolerance approaches like Triple Modular Redundancy (TMR) or State Machine Replication (SMR) [11, 13].

In 2000, Arora and Kulkarni [10] published a method to automatically add fault-tolerance to arbitrary programs, thereby *synthesizing* fault-tolerant programs. It is based on a general theory of fault-tolerance [1] and can be regarded as a generalization of many practical techniques like error-correcting codes, TMR or SMR. The Arora-Kulkarni method (henceforth abbreviated with AK) has also been implemented in a tool called *FTSyn* (Fault-Tolerance Synthesis) [7]. While AK is generic for programs, it is restrictive with respect to other input: It assumes so-called *fusion closed* specifications.

Intuitively, fusion-closure of a specification means that at every time within the execution of a program it is possible to decide whether the specification holds or is violated only by the information contained in the current state of the system. In the case of a non-fusion closed specification, history information must be kept as additional state in order to decide whether the specification is satisfied or not. While being simple to add, history information causes an exponential growth of the state space if it is added in a generic way, e.g., by using history variables.

In this direction, Gärtner and Jhumka [9] proposed a method to extend AK to also handle non-fusion closed specifications. Intuitively, the Gärtner-Jhumka method (henceforth abbreviated by GJ) can be regarded as a fine-grained method to add history information which does not in general lead to state-space explosion. Moreover, GJ was designed as a modular extension of AK, meaning that the output of GJ is suitable for further processing using AK. However, GJ has not been implemented yet.

Contributions The contributions of this paper are threefold: Firstly, we present an implementation of GJ called *FCPre* (Fusion-Closure Preprocessor). FCPre works as a modular preprocessing extension of FTSyn, the implementation of AK. So for the synthesis of programs with non-fusion closed specifications, both tools are needed.

Secondly, we show how it is possible to increase the confidence in the output of fault-tolerance synthesis by adding automatic verification in the back end. More specifically, we have used the model checker Spin [14] to validate the fault-tolerance properties of the synthesized programs. We describe the lessons learnt from this experiment.

Thirdly, and as a result of the previous point, we report on a slight error in the proof of GJ. We have identified a case where the method does not work. While the fault does not invalidate the entire method, we evaluate its impact and describe cases in which the application of GJ may lead to incorrect results.

The source code of the implementation together with the documentation can be downloaded [2] and used to validate the results and to perform further experiments.

Paper Outline We present the system model and the background of AK and GJ in Section 2. Section 3 then describes the implementation of GJ together with the integration of Spin. The final Section 4 gives an evaluation of the implementation and describes the fault found in GJ.

2. Formal Background

In this section we present the formal background of fault-tolerance. We first introduce the system model, then define the problem of automatically adding fault-tolerance and finally describe the two algorithms to synthesize fault-tolerance. All definitions are adapted from the original work on AK [10] and GJ [9].

2.1. System Model

We model programs as finite state automata. A *state space* is an unstructured finite nonempty set C of states. We denote states by uncapitalized letters, e.g., $a, b, c, \ldots$. A *state transition* t *over* C is a pair (r, s) of states from C. A *program* (or *system*) Σ consists of a triple (C, I, T) where $I \subseteq C$ denotes the set of initial states and T the set of transitions.

A *trace over* C is a nonempty sequence $\sigma = s_1, s_2, s_3, \ldots$ of states over C. Traces can be finite or infinite. The *concatenation* of a finite trace α and a trace β is denoted by $\alpha \cdot \beta$. A *transition* t *occurs in* σ if there exists an i such that $(s_i, s_{i+1}) = t$.

A *property over* C is defined as a set of traces over C. A trace σ *satisfies* a property P iff $\sigma \in P$. If σ does not satisfy P we say that σ *violates* P.

There are two kinds of properties: *safety* and *liveness* properties. Informally, liveness properties are properties that are violated in infinite time and fulfilled in finite (but arbitrary) time. A liveness property says that something good will eventually happen. For example, "Eventually, the system will output a value." is a liveness property. Since we do not consider liveness properties in this work, we omit a formal definition of liveness.

By contrast, safety properties are violated in finite time and fulfilled in infinite time. A safety property denotes that something bad will not happen. Formally, a safety property is defined as follows. A *safety property* S *over* C is a property over C for which the following holds: For each trace σ which violates S there exists a prefix α of σ such that for all traces β, $\alpha \cdot \beta$ violates S. One example of a safety property is mutual exclusion ("No two processes access their critical section at the same time.").

The set of initial states I and the set of transitions T together describe a safety property S, where S contains all traces starting in a state in I and using only transitions from T. S is denoted by *safety-prop*(Σ), or (by abuse of notation) simply Σ. A *safety specification* is a safety property. In this work, we only consider safety specifications. A program Σ satisfies a specification $SPEC$ iff $\Sigma \subseteq SPEC$.

Let C be a state set, $s \in C$, X be a specification over C, α, γ be finite state sequences, and β, δ be state sequences over C. Specification X is *fusion closed* if the following holds: If $\alpha \cdot s \cdot \beta$ and $\gamma \cdot s \cdot \delta$ are in X then $\alpha \cdot s \cdot \delta$ and $\gamma \cdot s \cdot \beta$ are also in X.

A *fault model* F maps a program $\Sigma = (C, I, T)$ to a program $F(\Sigma) = (F(C), F(I), F(T))$ such that $F(C) = C$, $F(I) = I$, and $F(T) \supset T$. For a given fault model F and a specification $SPEC$, we say that a *program* Σ *is* F*-intolerant with respect to* $SPEC$ if Σ satisfies $SPEC$ but $F(\Sigma)$ violates $SPEC$. It is sufficient to restrict faults to the addition of transitions because additional transitions are the only way to compromise a safety specification. The loss of transitions would accordingly be the way to compromise a liveness specification. We exclude faults that modify initial states, i.e. those faults that occur before system starts.

Given two programs $\Sigma_1 = (C_1, I_1, T_1)$ and $\Sigma_2 = (C_2, I_2, T_2)$, a *state projection function* $\pi : C_2 \to C_1$ tells which states of Σ_2 are equivalent with respect to Σ_1. It is applicable to traces in the following way: for a trace $s_1, s_2, s_3, \ldots$ over C_2 holds that $\pi(s_1, s_2, \ldots) = \pi(s_1), \pi(s_2), \ldots$.

Let $\Sigma_1 = (C_1, I_1, T_1)$ and $\Sigma_2 = (C_2, I_2, T_2)$ be two programs. Program Σ_2 *extends* program Σ_1 using state projection function π iff $C_2 \supseteq C_1$, π is a total mapping from C_2 to C_1 (for simplicity we assume that for any $s \in C_1$

holds that $\pi(s) = s$), and $\pi(\Sigma_2) = \Sigma_1$.

Intuitively, a *fault-tolerant version* of a fault-intolerant program Σ_1 is a program Σ_2 which has the same behavior as Σ_1 if no faults occur, but additionally satisfies the given specification in the presence of faults. Formally, a program Σ_2 is the *F-tolerant version* of program Σ_1 for *SPEC* using state projection π iff Σ_1 is F-intolerant with respect to $SPEC$, Σ_2 extends Σ_1 using π, and $F(\Sigma_2)$ satisfies *SPEC*.

The basic task we would like to solve is to construct a fault-tolerant version for a given program and safety specification.

Definition 1 (fail-safe transformation problem) *Given a program Σ which is F-intolerant with respect to a safety specification SPEC. The* fail-safe transformation problem *consists of finding a fault-tolerant version of Σ.*

Fault-tolerance mechanisms can be affected by faults in the same way as the original program. So new faults can be created by F after installing fault-tolerance procedures. Assuming however that after every step of fault-tolerance synthesis new faults are added, then—in general—it would be impossible to build a fault-tolerant version of the program since the procedure of adding fault-tolerance mechanisms could go on endlessly. If the fault model guarantees that at some point in future, no new faults are generated, it is called *finite fault model*. We assume a finite fault model in the context of this work.

New faults may be created by F in the F-tolerant version concerning newly created states. However, the "old" faults must still be possible even in states that are equivalent under state projection π. This is captured in the concept of extension monotonicity.

Formally, a fault model F must be *extension monotonic*, i.e., for any two programs $\Sigma_1 = (C_1, I_1, T_1)$ and $\Sigma_2 = (C_2, I_2, T_2)$ such that Σ_2 extends Σ_1 using π holds:

$$F(T_1) \setminus T_1 \subseteq F(T_2) \setminus T_2$$

2.2. The Arora-Kulkarni method (AK) and FTSyn

AK solves the fail-safe transformation problem for fusion closed specifications. The basic idea of AK is the creation of redundant states, i.e., making program states that were reachable in the fault-intolerant program unreachable in the fault-tolerant version. They utilize that fusion closed safety specifications can be equivalently expressed as a set of so called "bad" transitions. Bad transitions are those which always cause a safety specification violation by their execution. Remember that fusion-closure means that the achievement or violation of the specification can be decided on every state without any further information. So, there needs to exist at least one transition $t = (a, b)$ with the following property: until the program is in state a, the specification holds. As soon as the transition is used and state b is reached, the specification is violated. For example, assume the specification "never c". It is quite easy to see that every transition leading to c must be a bad transition.

Assume $t = (a, b)$ is a bad transition, so its occurrence has to be prevented. Since the specification is fusion closed, the execution of t will cause any trace in which t occurs to violate the specification. So, t has to be made unreachable. To guarantee that, one has to distinguish between the following cases:

- If t is a reachable program transition, specification violations may occur without the incidence of a fault. This scenario is excluded through our assumptions.
- If t is a non-reachable program transition, it can be simply removed.
- If t is a fault transition, it cannot be just removed. In this case, the starting state a of t has to be made unreachable. But this is only possible if there is a non-reachable program transition on every path from a starting state to a that can be simply removed. Otherwise, there is no fault-tolerant version.

This line of reasoning does not work if we assume a non-fusion closed specification. It is easy to imagine that if the violation of the specification depends on the past (see above), a transition is not necessarily good or bad but it may be both.

Figure 1. Example of AK. The specification is "never d".

As an example, consider the program depicted in Fig. 1. Dashed edges indicate fault transitions, solid edges indicate program transitions. The challenge is to make state d unreachable. First, one has to check if the safety specification can be violated in the absence of faults. Since this is not the case here, we can continue. The transition leading to state d is (c, d). It is a program transition and it is not reachable from the initial state in the absence of faults. So, it can be eliminated because the removal does not change the behavior of the system in the fault free scenario.

The example in Fig. 1 helps to show one example that cannot be synthesized. Imagine there would be a program transition (b, c). In this case, state d would be reachable only by program transitions, so that no transition could be removed without changing the program behavior in the absence of faults. Such a removal would destroy the equivalence of the fault-intolerant and the fault-tolerant version.

FTSyn [7] was implemented by Ebnenasir as part of a PhD thesis [6] advised by Kulkarni, one of the authors of AK. The source code of FTSyn is available for download upon request (a version is also included in the distribution of FCPre [2]). FTSyn was written in the Java programming language.

2.3. The Gärtner-Jhumka method (GJ)

We now give an overview of the work of Gärtner and Jhumka [9] and contrast it with AK. In contrast to AK, GJ deals with specifications which are not necessarily fusion closed. GJ pays particular attention at those states where a specification violation cannot be definitely decided. These states are called *bad fusion points*. Informally, a bad fusion point has at least two different "pasts" (i.e., execution histories), one that leads to a specification violation, and another that will meet the specification. An example is given in Fig. 2. Observe that state c is a bad fusion point. By making the transition from c to d, an execution history of a, c will lead to a specification violation, while an execution history of a, b, c satisfies the specification. So, it is not possible to decide whether (c, d) is a bad transition or not without any information about the past. This effectively results from not assuming fusion-closure of specifications.

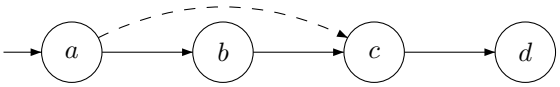

Figure 2. Example of GJ. The specification is "d implies previously b".

The aim of GJ is to remove bad fusion points. Bad fusion points are duplicated to separate the two cases. All such duplicate states map to the same state using a state projection function. In our example, state c is split up to remove the bad fusion point. This results in the program given in Fig. 3. The new state c' allows us to distinguish between the "good" and the "bad" transition. The transition (c, d) now has to violate the specification while (c', d) definitely meets the specification. Neither c nor c' are a bad fusion point after duplication.

After eliminating all bad fusion points of the original program Σ, GJ outputs an extension Σ' of Σ. Gärtner and Jhumka [9] show that applying a solution to the fail-safe transformation problem to Σ' results in a fault-tolerant version of Σ. The specification used for this step is simply the fusion-closure of the original (non-fusion closed) specification. In our example, the resulting program in Fig. 3 can be given to AK (implemented by FTSyn) for the synthesis. FTSyn finally eliminates the bad transition (c, d).

So overall, AK solves the fail-safe transformation problem for fusion closed specifications while GJ (in conjunction with AK) solves the problem for general safety specifications.

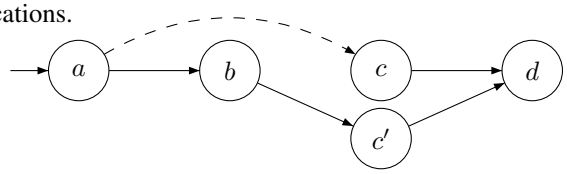

Figure 3. Next step of GJ. The specification is "d implies previously b".

3. Implementation of GJ

We now describe *FCPre*, the implementation of GJ. FCPre is a front-end to FTSyn, the implementation of AK. We describe FCPre in three steps: We first present an overview of the system design in Section 3.1. We then give an insight into the difficulties of implementing GJ in Section 3.2. Finally, Section 3.3 describes how we used the model checker Spin as a back-end to validate the synthesis process.

3.1. System design

As we want to synthesize programs, a program is expected as input. The algorithms however work on graphs. So, we use a FTSyn method to convert a given program in guarded commands [5] into a directed graph. This graph together with the specification is afterwards passed to FCPre.

The specification has to be given in an extra file. There, it is encoded as a boolean formula with comparison operators and logical operators. One of the latter is the *previously* operator. It returns true, if the following expression has been true somewhere in the past. A previously condition is in general bound to another criterion. In most cases, the incidence of an event implies the previous occurrence of another. For example, the reaching of a program state or a group of states that share a common criterion implies that previously another program state or some state of a group has been visited. This way, for example, it is possible to assert that program states are visited in a correct order. The formula is parsed and transformed into a specification tree. By the use of the specification tree, bad fusion points are eliminated and the fusion-closure of the specification is computed.

After the application of GJ, the output is again translated from a directed graph to guarded commands, i.e. a new program is generated. This program serves as input for FTSyn. The output of FTSyn is also a program. It is finally transformed to a valid input for the model checker Spin. The output of Spin denotes if the safety specification can be violated in the synthesized program.

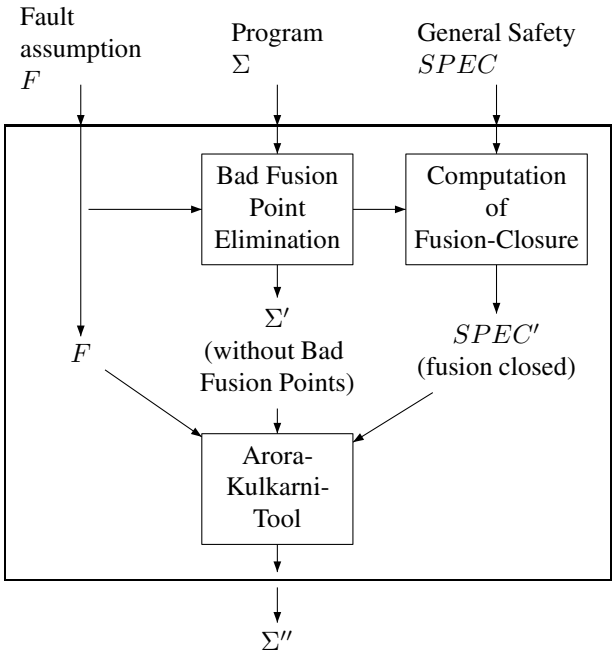

Figure 4. Classification of FCPre and FTSyn in the context of the general fail-safe transformation problem. FCPre computes the fusion-closure of the specification and eliminates the bad fusion points. FTSyn is represented by the Arora-Kulkarni-Tool and solves the fusion closed fail-safe transformation problem.

There are about six programs interacting in the fault-tolerance synthesis process including the model checking part. These programs are managed by a shell script. There are programs for the synthesis itself as well as for format casting in-between. The compilation, the execution, the removal, and the moving of files are done by this script.

3.2. Implemented algorithms

In this section, we give an overview of the details of the implementation of GJ. Data structures and algorithms are presented.

The most critical aspect of GJ is the elimination of bad fusion points. First, however, bad fusion points have to be identified.

Definition 2 (fusion, fusion point, bad fusion point) *Let s be a state and $\alpha = \alpha_{pre} \cdot s \cdot \alpha_{post}$ and $\beta = \beta_{pre} \cdot s \cdot \beta_{post}$ be two traces in which s occurs. Then we define*

$$\mathit{fusion}(\alpha, s, \beta) = \alpha_{pre} \cdot s \cdot \beta_{post}$$

If $\mathit{fusion}(\alpha, s, \beta) \neq \alpha$ and $\mathit{fusion}(\alpha, s, \beta) \neq \beta$ we call s a fusion point *of α and β.*

Let $SPEC$ be a specification, Σ be a system satisfying $SPEC$, s be a state of Σ, and F a fault model such that $F(\Sigma)$ violates $SPEC$. State s is a bad fusion point *of Σ for $SPEC$ in the presence of F iff there exist traces $\alpha, \beta \in SPEC$ such that s is a fusion point of α and β, $\mathit{fusion}(\alpha, s, \beta) \in F(\Sigma)$, and $\mathit{fusion}(\alpha, s, \beta) \notin SPEC$.*

The task is the discovery of a state s and two paths α and β that match the conditions in the definition. Finding a bad fusion point can be hard because testing the criteria, i.e. the existence of such paths, for each state is complex. So, we start with the identification of possible bad fusion points. First, all states that accomplish two necessary conditions are marked as "bad fusion point candidates". First, there need to be at least two incoming transitions, one that belongs to α and the other for β. The second condition says that there needs to be at least one fault transition in the past of one incoming path. So, we know that one of the paths should lead to a specification violation.

Without loss of generality, we assume that β is good and α is the bad trace, i.e. the one that contains the fault transition. So, we first try to find a path β from an initial state to an end state, i.e. a state that has no successor. This path has to satisfy the specification and it must contain s. We take the bad fusion point candidates as s, one after the other. For all initial states, we start the search algorithm.

```
find_traces(DAG F(Σ), bad fusion point
   candidate s){
for all c ∈ I(F(Σ)) do
   find_β(F(Σ), c, <c>, SPEC, s);
end for}
```

The following procedure looks for matching traces with $\beta \in SPEC$ and $s \in \beta$. We assume the program graph to be directed and acyclic, because a deterministic program with a cyclic graph might run infinitely long. If cyclic graphs are taken into account, our algorithm has to be slightly modified for termination. First, all paths through $F(\Sigma)$ are recorded. If a path l matches the criteria, it is taken as β and the algorithm that finds α is started.

```
find_β(DAG F(Σ), actual state c,
   β candidate l, specification SPEC,
   bad fusion point candidate s){
if (c has no successor) then
   /* l is β candidate */
   if (l ∈ SPEC) and (s ∈ l) then
      find_α(F(Σ), s, <s>, SPEC, l);
   end if
else /* c has successor */
   for all successors t of c do
      find_β(F(Σ), t, l · <t>, SPEC, s);
      break if a bad fusion point
       has been found;
   end for
end if}
```

We found a specification compliant path that includes a bad fusion point candidate s. The part after s is taken for the fusion. It is the first part that is still needed. It suffices to find a path from an initial state to s that satisfies the specification. According to the definition, the fusion of both traces has to violate the specification. The following algorithm is meant to find the trace α.

```
find_α(DAG F(Σ), actual state c,
   α candidate l, specification SPEC,
   trace β){
if (c has no predecessor) then
   /* l is α candidate */
   if (l ∈ SPEC) then
      if (fusion(l, s, β) ∉ SPEC) then
         /* α is found: l */
         /* s is bad fusion point */
         eliminate bad fusion point s
          with respect to α and β;
      end if
   end if
else /* c has predecessors */
   for all predecessors t of c do
      find_α(F(Σ), t, <t> · l, SPEC, β);
      break if a bad fusion point
       has been found;
   end for
end if}
```

The algorithm works in the following way. It starts at the bad fusion point candidate s and considers all paths from an initial state to s by going backwards from s. Probably, the continuation of α to an end state would cause a specification violation. But this is not important here. It is just interesting that α does not violate $SPEC$. If the algorithm succeeds to find α and β that match, the bad fusion point candidate is a real bad fusion point.

In our example given in Fig. 2, state c is marked as a bad fusion point candidate since it has two incoming transitions from a and b, and the fault transition (a, c) in the past. The good path β is a, b, c, d, α is a, c, and the bad fusion point is c. Note that α has to fulfill the specification but leads to a specification violation on continuation.

After explaining the method how a bad fusion point is identified, the next thing is the presentation of bad fusion point removal. The idea is the separation of both traces α and β. Therefore, a new state s' is generated as an exact copy of the bad fusion point s. The transitions that belong to β are then diverted to s'. Additionally, the outgoing transition of s that belongs to β is copied with initial state s'. Let us assume that the transition (s, t) is part of β. Then, after the removal, (s', t) is introduced in the system. Note, that the transition (s, t) is not immediately removed. However, this transition is now part of the bad trace and probably will be removed in the next step by FTSyn. Regarding the faults, it is important to be able to distinguish the "clone" states s and s'.

It is worth mentioning that the state projection function is implemented indirectly. The most important aspect of the state projection function is to ensure that duplicated states are relevant for the compliance with the specification, i.e. if a duplicated state is visited, the effect should be the same as if the original state is visited. For example, assume the specification "e implies previously b". The state projection function guarantees that if e is visited after b' but b is not visited at all, the specification is satisfied. If e' is visited, b or an equivalent state has to be visited before. This job is done by construction. The specification relies on program variables. The duplicated state has the same values as its original. So, all specification clauses that concern a state s, concern its duplication s' the same way. For example, assume a state s with the unique state number 3. The program variable x has the value 4. If s is duplicated, a new state s' is generated. It gets a unique state number, e.g. 13. The program variable x has the value 4 as in s. So, s and s' differ in at least one variable value (the state number), but by the specification, they are treated the same. The specification can only refer to x because the state number variable is introduced after the definition of the specification.

In the example displayed in Fig. 2, the bad fusion point c has to be removed. We generate the new state c' as a copy of c. The transition (b, c) is part of β and thus diverted to c', i.e. (b, c) is replaced by (b, c'). The outgoing transition (c, d) is just copied instead of diverted to c' because we do not want to change the behavior of the program yet (see Fig. 3).

Now, the computation of bad transitions is explained. Recall that after the removal of all bad fusion points, there are just good or bad transitions. The fusion closed specification can be expressed as a set of bad transitions. The algorithm starts at an initial state in the extended state space and traverses all transitions. If one transition violates the non-fusion closed specification, it is added to the set of bad transitions and the consecutive path is not considered. The set of bad transitions is passed as safety specification to FTSyn.

3.3. Interfacing with Spin

This section deals with interfacing fault-tolerance synthesis programs, i.e. FCPre and FTSyn, with the model checker Spin [14].

Brief Overview of Spin Spin (Simple Promela INterpreter) is used to verify the final results of FTSyn after each synthesis. Therefore, FTSyn's output in guarded commands has to be transformed to PROMELA (PROcess MEta LAnguage) [12], the input language of Spin. Spin has the ad-

vantage of portability across a large number of platforms, namely Unix, Linux, cygwin, Plan9, Inferno, Solaris, Mac, and Windows. Together with the presumption that FTSyn and FCPre are written in the Java programming language that also provides a certain portability with its virtual machine, the portability of Spin may become a big advantage. Good documentation of Spin is available [14], and the flexibility of runtime options is another strong point in favor of choosing Spin.

In the context of Spin, we make use of so-called *never claims*. Spin expects two input files. The first file contains the system itself, i.e. variable declarations, the process declarations including the fault process and the initialization. It has the file type *.prom* and is written in PROMELA. The second file ending with *.ltl* however consists of the specification coded as a never claim, i.e. a formula that states what should never happen in the execution of this program. The never claim is automatically generated by Spin given a LTL (Linear Time Logic) formula.

Using Spin to Validate Output As we had no influence on neither the output format of FTSyn nor the input format of Spin, one has to convert one into the other though it might be hard to maintain semantics. Not all needed information is encoded in the FTSyn output file, e.g. the fault transitions and the specification do not appear. This is the reason why additional files have to be generated by FCPre for Spin. These files contain detailed information about states besides fault transitions and the specification.

4. Evaluation

In this section, we present some results from using the implementation of GJ. First, we discuss one working example. Then, we present a case where GJ fails.

4.1. Illustrated example

We want to illustrate the functioning of the complete fault-tolerance synthesis by presenting one example produced by our tool. The example is one of those presented by Gärtner and Jhumka [8] in an extended technical report of their conference paper [9].

The example is not fully synthesized by Gärtner and Jhumka [8]. Here we finish the synthesis of this example. It is given in Fig. 5. There are two bad fusion points, namely c and d. They are removed by FCPre, c first and d afterwards, before both violating transitions (c, d) and (d, e), each violating one part of the specification, are deleted by FTSyn. As a proof of correctness, Spin verified the fault tolerance property.

We conducted some more experiments [3]. The synthesis tool works properly on these examples. Spin verified

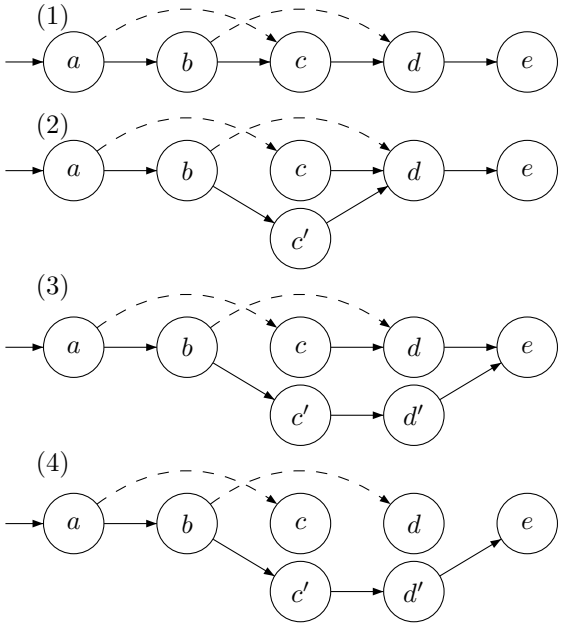

Figure 5. Example of a complete fault-tolerance synthesis. The specification is "(d implies previously b) and (e implies previously c)".

all results successfully. The algorithms of FCPre are not optimized for runtime. There may be faster algorithms for these purposes. The algorithms that are used are developed for stability and trace-ability. On a moderately powerful PC (900 MHz, 256 MB RAM) it took less than half a minute to fully synthesize each of the examples. Most of the time is needed to compile the source files using `javac`.

4.2. The Error in GJ

In this section, we describe the error that we found in the correctness proof of GJ. There is a case where GJ cannot continue synthesizing the program. We must say that this is not a fatal error. It is rare and the occurrence is evident so that the intermediate result does not appear to be the final result.

The error in the proof is located in Lemma 3, step 1.2 [8, p. 20]. In the context of the proof, it is abstracted from the kind of incoming and outgoing transitions. It is just assumed that every transition can be diverted. However, we assumed that fault transitions cannot be diverted. So, there is the implicit assumption that the adjacent transitions are program transitions. It would be preferable to make a case distinction at this point. So, the method itself might not be faulty but potentially incomplete. If one introduces an instruction which is executed in the case of a fault transition, the method would work in all cases.

We want to demonstrate the fault using an example.

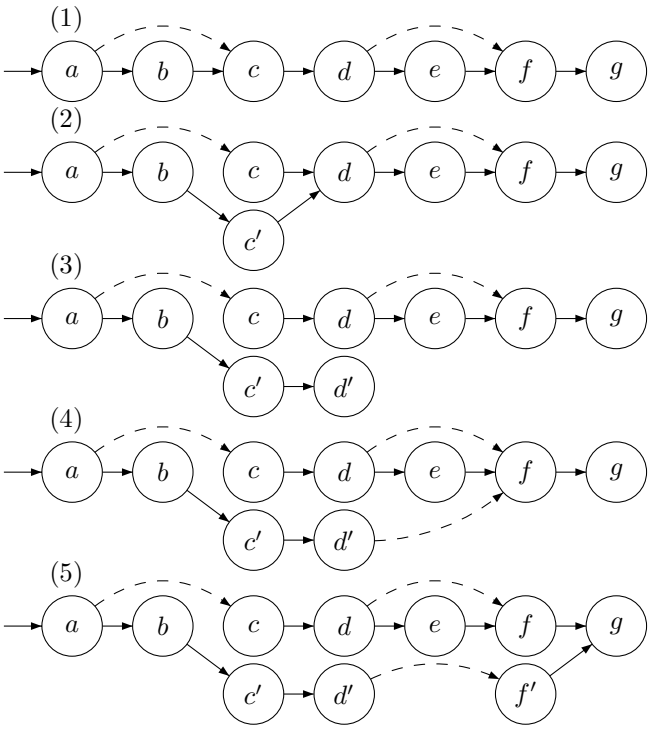

Figure 6. The graphical illustration of the fault in GJ correctness lemma. The specification is "g implies previously (b or e)".

Let us look at Fig. 6. The first bad fusion point c is duplicated. Afterwards, state d is a bad fusion point candidate. We look for applicable α and β and find $\alpha = a, c, d$ and $\beta = a, b, c', d, f, g$. The next step is the duplication of state d. The transition $(c', d) \in \beta$ is diverted to (c', d'). Finally, one would have to add the transition (d', f). This, however, is a fault transition that cannot be newly generated.

Let us assume for a moment that adding new fault transitions would be allowed but not the redirection or deletion. In this case, we would add a new fault transition (d', f) (cf. Fig. 6(4)). The next bad fusion point would be f, $\alpha = a, c, d, f$, and $\beta = a, b, c', d', f, g$. After the duplication of f, the fault transition (d', f) would have to be diverted to (d', f'). This cannot work.

The correctness proof of GJ is correct as long as the transition that belongs to the good path β and starts at the bad fusion point is not a fault transition.

Acknowledgment

The author wishes to thank Felix C. Freiling for helpful conversations and comments on an earlier version of this work. Arshad Jhumka provided hints regarding the discovered error in the proof of GJ. The author also would like to thank Ali Ebnenasir for useful advice in the early development phase of FCPre.

References

[1] A. Arora and S. S. Kulkarni. Component based design of multitolerant systems. *IEEE Transactions on Software Engineering*, 24(1):63–78, Jan. 1998.

[2] B. Braun. FCPre Project Software. Internet: `https://pi1.informatik.uni-mannheim.de:8443/pub/research/theses/FCPre-1.0-includesFTSyn.zip`, Aug. 2006. Includes the source code of FTSyn.

[3] B. Braun. Implementing automatic addition and verification of fault tolerance. Diploma thesis, RWTH Aachen University, Department of Computer Science, 2006.

[4] B. Braun. FCPre: Extending the Arora-Kulkarni Method of Automatic Addition of Fault-Tolerance. Technical Report 275-06, University of Hamburg, Department of Computer Science, Hamburg, Germany, Dec. 2006.

[5] E. W. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, Aug. 1975.

[6] A. Ebnenasir. Automatic synthesis of fault tolerance. Phd thesis, Michigan State University, Department of Computer Science, 2005.

[7] A. Ebnenasir. Ftsyn project. Internet: `http://www.cs.mtu.edu/~aebnenas/research/tools/ftsyn.htm`, Nov. 2006.

[8] F. C. Gärtner and A. Jhumka. Automating the addition of fail-safe fault-tolerance: Beyond fusion-closed specifications. Technical Report IC/2003/23, Swiss Federal Institute of Technology (EPFL), School of Computer and Communication Sciences, Lausanne, Switzerland, Apr. 2003.

[9] F. C. Gärtner and A. Jhumka. Automating the addition of fail-safe fault-tolerance: Beyond fusion-closed specifications. In *Proceedings of Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*, Grenoble, France, Sept. 2004.

[10] S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. In M. Joseph, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems, 6th International Symposium Proceedings*, number 1926 in LNCS, pages 82–93, Pune, India, Sept. 2000. Springer Verlag.

[11] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[12] Promela language reference. Internet: `http://spinroot.com/spin/Man/promela.html`, July 2006.

[13] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[14] Spin software tool. Internet: `http://spinroot.com/spin/whatispin.html`, July 2006.