Fault Masking in Tri-redundant Systems

Mohamed G. Gouda[1], Jorge A. Cobb[2], and Chin-Tser Huang[3]

[1] Department of Computer Sciences
The University of Texas at Austin
gouda@cs.utexas.edu
[2] Department of Computer Science
The University of Texas at Dallas
cobb@utdallas.edu
[3] Department of Computer Science and Engineering
University of South Carolina at Columbia
huangct@cse.sc.edu

Abstract. A tri-redundant version of a system S is a system T that is specified from S as follows. First, system T has the same number of processes and the same topology as system S. Second, each variable x in a process in system S is replaced by three variables x, x', and x'' in the corresponding process in system T. Third, the actions in each process in system S are modified before they are added to the corresponding process in system T and some new actions are added to the corresponding process in system T. In this paper, we show that a tri-redundant version T of a system S has interesting stabilization and fault-masking properties. In particular, we show that if S is stabilizing, then T is also stabilizing. We also show that if T ever reaches stabilization, and then a "visible fault" occurs, then the effect of the fault is masked and the reached stabilization of T remains in effect.

1 Introduction

A system S is called P-stabilizing, where P is a boolean expression over the variables in S, iff the following two conditions hold. First, any computation of S, that starts at a state where P is false, reaches a state where P is true. Second, the execution of any action in system S that starts at a state where P is true, ends at a state where P is true. See for example [4,5,9].

The fact that a system S is P-stabilizing indicates that S is fault-tolerant to some degree. In particular, if a fault ever causes system S to reach a state where P is false, further executions of the actions in S causes S to return to a state where P is true. Moreover, once S reaches a state where P is true, P continues to be true at each subsequent state of S.

There are (at least) two research directions that can be followed in order to enhance the relationship between stabilization and fault-tolerance. The first research direction is called fault-containment and it has been explored in [7,8,10]. The second research direction is called fault-masking and it is the subject of the current paper. We compare these two research directions next.

A.K. Datta and M. Gradinariu (Eds.): SSS 2006, LNCS 4280, pp. 304–313, 2006.

Let S be a P-stabilizing system, and let F be a class of faults each of which can change the value of some variable in S. Assume that each fault f in F is assigned a "severity measure" $m(f)$. System S is called F-containing iff for each fault f in F, any computation of S, that starts at a state s_f, where s_f can be reached by applying fault f to a state where P is true, reaches a state where P is true after at most $O(m(f))$ transitions from the starting state s_f. In other words, F-containment ensures that the time that system S needs to recover from a fault f in F is proportional to some measure of the severity of fault f.

Let S be a P-stabilizing system, and let F be a class of faults each of which can change the value of some variable(s) in S. System S is called F-masking iff for each fault f in F, and for each variable x whose value is changed by fault f, any computation of S, that starts at a state s_f, where s_f can be reached by applying fault f to a state where P is true, has an execution of some action ac that restores the value of variable x to its value before f is applied, and moreover any action execution, that preceeds the execution of ac in the computation, neither reads nor writes variable x. In other words, F-masking ensures that the application of any fault f in F has a limited effect on the action execution in system S.

In this paper, we describe a transformation that can transform any stabilizing system S to a "tri-redundant" version T such that T is both stabilizing and F-masking, where F is a rich class of faults called visible faults.

The concept of fault masking presented in this paper has somewhat similar objectives, if not the same technical details, as two earlier concepts: superstabilization and snap stabilization. A superstabilizing system [6] is a stabilizing system that dampens the effects of its own "topology changes" when they occur. This is accomplished by ensuring that the system satisfies a specified safety predicate from the instant when the topology of the system changes, causing the system to lose its stabilization, until the instant when the stabilization of the system is restored. A snap stabilizing system [3] is a stabilizing system that is guaranteed to always behave according to its specification regardless of how the state of the system is changed due to fault occurrence. Clearly, snap stabilization is a lofty goal. Unfortunately, many systems cannot be made snap stabilizing.

2 Stabilizing Systems

The *topology* of a system is a connected undirected graph, where each node represents one process in the system, and each edge between two nodes p and q indicates that processes p and q are neighbors in the system, and so each of the two processes can read the variables of the other process, as discussed below.

Each *process* in a system is specified by a finite set of variables and a finite set of actions. The values of each variable are taken from some bounded domain of values. Each action of a process p is of the form

$$\langle \mathsf{guard} \rangle \rightarrow \langle \mathsf{assignment} \rangle$$

where $\langle \mathsf{guard} \rangle$ is a boolean expression over the variables of process p and the variables of all neighboring processes of p, and $\langle \mathsf{assignment} \rangle$ is a sequence of assignment statements, each of which is of the form

$$x := \mathrm{E}(y, \ldots)$$

where x is a variable in process p, E is an expression of the same type as variable x, and y is a variable either in process p or in any neighboring process of p.

A *state* of a system S is specified by one value for each variable, taken from the domain of values of that variable, in each process in S.

A *transition* of a system S is a triple of the form

$$(s, ac, s')$$

where s and s' are two states of system S and ac is an action in some process in S such that the following two conditions hold.

i. *Enablement:* The guard of action ac is true at state s.
ii. *Execution:* Executing the assignment of action ac, when system S is in state s, yields system S in state s'.

A *computation* of a system S is a sequence of the form

$$(s_0, ac_0, s_1), (s_1, ac_1, s_2), \ldots$$

where each element $(s_i, ac_i, s_{(i+1)})$ is a transition of S such that the following two conditions hold.

i. *Maximality:* Either the sequence is infinite or it is finite and its last element $(s_{(z-1)}, ac_{(z-1)}, s_z)$ is such that the guard of every action in system S is false at state s_z.
ii. *Fairness:* If the sequence has an element $(s_i, ac_i, s_{(i+1)})$ and the guard of some action ac is true at state $s_{(i+1)}$, then the sequence has a later element $(s_k, ac_k, s_{(k+1)})$ where ac is ac_k or the guard of ac is false at state $s_{(k+1)}$.

A *predicate* P of a system S is a boolean expression over the variables in all processes in system S.

A predicate P of a system S is said to be *closed* in S iff for every transition (s, ac, s') of system S, if predicate P is true at state s, then P is true at state s'.

A system S is called *P-stabilizing* iff predicate P satisfies the following two conditions [1].

i. *Closure:* Predicate P is closed in system S.
ii. *Convergence:* Predicate P is true at a state in every computation of system S.

3 Systems with Tri-redundancy

In the previous section, we discussed how to specify a system S. Next, we describe how to specify a tri-redundant version T of any system S. The tri- redundant version T is specified from S as follows.

i. *Topology:* System S has the same number of processes and the same topology as system T. Thus, there is a natural one-to-one correspondence between the processes in S and those in T. For convenience, each process p in S has the same name as that of the corresponding process p in T.
ii. *Variables:* For each variable x in a process p in system S, there are three corresponding variables x, x', and x'' in the corresponding process p in system T. Each of the variables x, x', and x'' in system T is of the same type and has the same domain of values as variable x in system S. We refer to x in T as the original copy of variable x in S, and refer to x' and x'' in T as the shadow copies of x in S.
iii. *Actions:* For each action of the form $\langle \mathsf{guard} \rangle \rightarrow \langle \mathsf{assignment} \rangle$ in a process p in system S, there is a corresponding action of the form $\langle \mathsf{guard}' \rangle \rightarrow \langle \mathsf{assignment}' \rangle$ in the corresponding process p in system T such that the following three conditions hold.
   (a) First, each occurrence of a variable x in $\langle \mathsf{guard} \rangle$ is replaced by an occurrence of the original copy of x, also called x, in $\langle \mathsf{guard}' \rangle$.
   (b) Second, for each variable x that occurs in $\langle \mathsf{guard} \rangle$ or in $\langle \mathsf{assignment} \rangle$, add a conjunct of the form $(x = x' \wedge x' = x'')$ to $\langle \mathsf{guard}' \rangle$.
   (c) Third, each statement of the form $x := \mathrm{E}(y, \ldots)$ in $\langle \mathsf{assignment} \rangle$ is replaced by a statement of the form $(x, x', x'') := \mathrm{E}(y, \ldots)$ in $\langle \mathsf{assignment}' \rangle$. The latter statement computes the value of expression E and then assigns the computed value to each of the three copies x, x', and x'' in T.
iv. *Additional Actions:* For each original copy x in a process p in system T, add an action of the following form to process p in T

$$x \neq x' \vee x' \neq x'' \;\rightarrow\; (x, x', x'') := \mathrm{MJR}(x, x', x'')$$

where $\mathrm{MJR}(x, x', x'')$ is the bit-wise majority function applied to the three variables x, x', and x''. This function is defined in some detail next.

Recall that each variable in a system has a bounded domain of values and that the three copies x, x', and x'' have the same (bounded) domain $D(x)$ of values. Thus, every value of each of the three copies x, x', and x'' can be represented by the same number, say r, of bits. The function $\mathrm{MJR}(x, x', x'')$ computes a value in the same domain $D(x)$ of values, and so each value of $\mathrm{MJR}(x, x', x'')$ can be represented by r bits.

The bits of $\mathrm{MJR}(x, x', x'')$ can be computed from the bits of x, x', and x'' as follows. For every i in the range $0\,..\,(k-1)$, the i-th bit of $\mathrm{MJR}(x, x', x'')$ is computed as the majority of three bits: the i-th bit of x, the i-th bit of x', and the i-th bit of x''.

4 Stabilization Theorem

In this section, we show that if a system S is stabilizing, then any tri-redundant version T of S is also stabilizing.

Theorem 1. (Stabilization of Tri-Redudant Systems).
Let S be a P-stabilizing system, and T be a tri-redundant version of S. System T is Q-stabilizing, where Q is the predicate

$$P' \wedge (\textit{for every original copy of } x \textit{ in } T, x = x' \wedge x' = x'')$$

and predicate P is syntactically identical to predicate P'. (Note that P is a predicate of system S and P' is a predicate of system T. Thus, each occurrence of x in P refers to a variable x in system S, and each occurrence of x in P' refers to the original copy of x in system T.)

Proof. The proof is divided into two parts. In the first part, we show that predicate Q is closed in system T, and in the second part, we show that Q is true at a state in every computation of system T.

First Part: Let (t, ac', t') be a transition of system T and assume that predicate Q is true at state t, we need to show that Q is true at state t'.

Because Q is true at t, we conclude that the predicate (for every original copy of x in T, $x = x' \wedge x' = x''$) is true at t. Thus, the guard $(x \neq x' \vee x' \neq x'')$ of each additional action in system T is false at t, and so ac' in the transition (t, ac', t') is not an additional action in system T. Rather, ac' is an action in system T that corresponds to an action ac in system S. The two actions ac and ac' are of the form

$$ac : \langle \mathsf{guard} \rangle \rightarrow \langle \mathsf{assignment} \rangle$$
$$ac' : \langle \mathsf{guard}' \rangle \rightarrow \langle \mathsf{assignment}' \rangle$$

where $\langle \mathsf{guard}' \rangle$ is the predicate $\langle \mathsf{guard} \rangle \wedge$ (for every variable x that occurs in ac, $x = x' \wedge x' = x''$), also, $\langle \mathsf{assignment} \rangle$ and $\langle \mathsf{assignment}' \rangle$ are identical except that each statement $x := \mathrm{E}(y, \ldots)$ in $\langle \mathsf{assignment} \rangle$ is replaced by the statement $(x, x', x'') := \mathrm{E}(y, \ldots,)$ in $\langle \mathsf{assignment}' \rangle$.

Let s and s' be the two states of system S that correspond to states t and t', respectively, of system T. It follows that the triple (s, ac, s') is a transition of system S. Moreover, because predicate Q is true at state t, we conclude that P is true at state s.

From the fact that system S is P-stabilizing (and so P is closed in system S), and the fact that triple (s, ac, s') is a transition of system S, and the fact that P is true at state s, it follows that P is true at state s'. Thus, both P' and Q are true at state s'.

Second Part: Let the sequence (t_0, ac_0, t_1), (t_1, ac_1, t_2), ... be a computation of system T. We need to show that predicate Q is true at some state in this computation.

Let x be an original copy in system T where the predicate $(x \neq x' \vee x' \neq x'')$ is true at the initial state t_0 of this computation. Then the guard of the additional action $x \neq x' \vee x' \neq x'' \rightarrow (x, x', x'') := \mathrm{MJR}(x, x', x'')$ in T is true at t_0. From the fairness condition of the computation, it follows that the predicate $(x = x' \wedge x' = x'')$ is true at a later state t_j in the computation. Moreover, because

each action in system T either keeps the values of x, x', and x'' unchanged, or assigns each of them the same new value, the predicate $(x = x' \wedge x' = x'')$ remains true at each of the states that occur after t_j in the computation.

From the above discussion, the computation $(t_0, ac_0, t_1), (t_1, ac_1, t_2), \ldots$ has a suffix $(t_k, ac_k, t_{(k+1)}), (t_{(k+1)}, ac_{(k+1)}, t_{(k+2)}), \ldots$ where the predicate (for each original copy x in T, $x = x' \wedge x' = x''$) is true at each state t_k, $t_{(k+1)}$, $\ldots$ in this suffix. Along this suffix, the execution of system T mirrors that of system S. Because system S is P-stabilizing, predicate P' is true at some state t_z in this suffix. Therefore, predicate Q is true at the same state t_z in the computation. □

5 Fault Masking Theorem

Let S be a P-stabilizing system and T be a tri-redundant version of S. From the stabilization theorem of tri-redundant systems (in the previous section), T is Q-stabilizing where Q is the predicate ($P' \wedge$ (for each original copy x in T, $x = x' \wedge x' = x''$)). In this section, we argue that if T is at a legitimate state, one where Q is true, and then some fault, from a rich class of faults called visible faults, occurs, then the effects of the fault are masked and the system quickly returns to a legitimate state, one where Q is true. We start by defining visible faults.

A fault f is visible iff it changes the values of some variables in system T such that the following two conditions hold:

i. *Legitimacy:* Immediately before f occurs, system T is at a legitimate state where predicate Q is true. It follows that for every original copy x in T, $xa = xa' \wedge xa' = xa''$, where (xa, xa', xa'') is the value of (x, x', x'') immediately before f occurs.

ii. *Transparency:* For every original copy x in T,

$$\text{MJR}(xa, xa', xa'') = \text{MJR}(xb, xb', xb''),$$

where (xa, xa', xa'') is the value of (x, x', x'') immediately before f occurs and (xb, xb', xb'') is the value of (x, x', x'') immediately after f occurs.

Assume that a visible fault f occurs in system T, and also assume that f changes the value of some (x, x', x'') in T from (xa, xa', xa'') to (xb, xb', xb''). From the legitimacy condition of f, $xa = xa' \wedge xa' = xa''$. Thus, from the transparency condition of f and from the fact that f has changed the value of (x, x', x''), $xb \neq xb' \vee xb' \neq xb''$.

Let t be the state of system T immediately after f occurs. Then, the predicate $(x \neq x' \vee x' \neq x'')$ is true at state t. System T has two types of actions where the triple (x, x', x'') occurs: actions ac_0, ac_1, $\ldots$ that correspond to some actions, where x occurs, in system S and the added action ac:

$$ac \; : \; (x \neq x' \vee x' \neq x'') \; \rightarrow \; (x, x', x'') := \text{MJR}(x, x', x'')$$

The guard of each action ac_i in T has a conjunct $(x = x' \wedge x' = x'')$ and so none of these actions can be executed until after action ac is executed. From the transparency condition of f, executing action ac changes back the value of (x, x', x'') from (xb, xb', xb'') to (xa, xa', xa''). Thus, the effect of fault f on the triple, and ultimately on system T, is masked. This argument proves the following theorem.

Theorem 2. (Fault-Masking of Tri-Redundant Systems).
Let S be a P-stabilizing system and T be a tri-redundant version of S. System T is F-masking, where F is the class of visible faults.

6 A Tri-redundant Spanning Tree

As an example, consider a system S that consists of n processes $p[i : 0 \,..\, n-1]$. The processes in S maintain an outgoing spanning tree whose root is process $p[0]$. Each process $p[i]$ has a variable $ds[i]$ to store the smallest number of hops needed to go from $p[0]$ to $p[i]$. Also each process $p[i]$, other than process $p[0]$ has a variable $pr[i]$ to store index g of the parent $p[g]$ of $p[i]$. The processes in S can be specified as follows.

process $p[0]$

var $ds[0] : 0 \,..\, n$

begin
true $\rightarrow$ $ds[0] := 0$
end

process $p[i : 1 \,..\, n-1]$

var $ds[i] : 0 \,..\, n$
$pr[i]$: index of parent of $p[i]$ in spanning tree

par g : index of an arbitrary neighbor of $p[i]$

begin
$ds[i] \neq \min(n, ds[pr[i]] + 1) \rightarrow$
$ds[i] := \min(n, ds[pr[i]] + 1)$

$\square$ $ds[i] > ds[g] + 1 \rightarrow$
$ds[i] := ds[g] + 1;$
$pr[i] := g$
end

This system has been shown to be stabilizing [2]. Unfortunately the system is not F-masking for any reasonable class F of faults. Consider for example a fault that changes the value of $ds[0]$ in process $p[0]$ from 0 to 1. The first action in any neighboring process $p[g]$ can be executed and read the faulty value of $ds[0]$ before the correct value of $ds[0]$ is restored (by the action of process $p[0]$).

To achieve F-masking, for class F of visible faults, system S needs to be transformed to a tri-redundant version T. The processes in system T are specified as follows.

process $p[0]$

var $ds[0], ds'[0], ds''[0] \ : 0\,..\,n$

begin

$$(ds[0] = ds'[0] \ \wedge \ ds'[0] = ds''[0]) \ \rightarrow$$
$$(ds[0], ds'[0], ds''[0]) := 0$$

$\square$

$$(ds[0] \neq ds'[0] \ \vee \ ds'[0] \neq ds''[0]) \ \rightarrow$$
$$(ds[0], ds'[0], ds''[0]) := \mathrm{MJR}(ds[0], ds'[0], ds''[0])$$

end

process $p[i : 1\,..\,n-1]$

var $ds[i], ds'[i], ds''[i]$: $0\,..\,n$
$pr[i], pr'[i], pr''[i]$: index of parent of $p[i]$ in spanning tree

par g : index of an arbitrary neighbor of $p[i]$

begin

$$ds[i] \neq \min(n, ds[pr[i]] + 1) \ \wedge$$
$$(ds[i] = ds'[i] \ \wedge \ ds'[i] = ds''[i]) \ \wedge$$
$$(pr[i] = pr'[i] \ \wedge \ pr'[i] = pr''[i]) \ \wedge$$
$$(ds[pr[i]] = ds'[pr[i]] \ \wedge \ ds'[pr[i]] = ds''[pr[i]])$$
$$\rightarrow$$
$$(ds[i], ds'[i], ds''[i]) := \min(n, ds[pr[i]] + 1)$$

$\square$

$$ds[i] > ds[g] + 1 \wedge$$
$$(ds[i] = ds'[i] \ \wedge \ ds'[i] = ds''[i]) \ \wedge$$
$$(pr[i] = pr'[i] \ \wedge \ pr'[i] = pr''[i]) \ \wedge$$
$$(ds[g] = ds'[g] \ \wedge \ ds'[g] = ds''[g])$$
$$\rightarrow$$

$$(ds[i], ds'[i], ds''[i]) := ds[g] + 1;$$
$$(pr[i], pr'[i], pr''[i]) := g$$

$$\square \quad (ds[i] \neq ds'[i] \vee ds'[i] \neq ds''[i])$$
$$\rightarrow$$
$$(ds[i], ds'[i], ds''[i]) := \text{MJR}(ds[i], ds'[i], ds''[i])$$

$$\square \quad (pr[i] \neq pr'[i] \vee pr'[i] \neq pr''[i])$$
$$\rightarrow$$
$$(pr[i], pr'[i], pr''[i]) := \text{MJR}(pr[i], pr'[i], pr''[i])$$

end

7 Concluding Remarks

In this paper, we described a transformation to transform any system S to a tri-redundant version T. We showed that if S is stabilizing then T is both stabilizing and F-masking for the class F of visible faults.

In our presentation, we assumed that system S is stabilizing under the assumption that the actions of S are executed one at a time. Nevertheless, the presentation can be extended in straightforward manner to the case where system S is stabilizing under the assumption that any subset of actions (at most one action from each process) in S are executed at a time. In this case, system T is stabilizing and F-masking under the same assumption that any subset of actions (at most one action from each process) in T are executed at a time.

In the above presentation, we assumed that the redundant version of any system S has "three" copies (x, x', x'') of every variable x in S. However, the only magic that is associated with this number "three" is that it is odd, and so when any fault occurs in the redundant system, the MJR function can always return a meaningful value. Therefore, the above presentation can be generalized in a straightforward manner such that the redundant version of a system has $(2 \cdot r + 1)$ copies of every variable in that system, where r is a positive integer.

In [11], Huang and Gouda have shown how to utilize two ideas, namely state checksums and tri-redundancy, to design a stabilizing token system that masks visible faults. Surprisingly, the theory of fault masking presented in the current paper is based solely on the idea of tri-redundancy. The question, of how to enrich this theory by injecting the idea of state checksums into it, seems interesting and enticing, but so far remains open.

Acknowledgment

The work of M. G. Gouda is supported in part by the National Science Foundation under Grant No. 0520250. The work of J. A. Cobb is supported in part by

a UTD Project Emmitt startup grant. The work of C. T. Huang is supported in part by by the AFRL/DARPA under grant No. FA8750-04-2-0260. The authors would like to thank Professor Eunjin (EJ) Jung, at the University of Iowa, for her comments on an earlier version of this paper.

References

1. A. Arora and M. G. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19:1015–1027, November 1993.
2. N.-S. Chen, H.-P. Yu, and S.-T. Huang. A self-stabilizing algorithm for constructing spanning trees. *Inf. Process. Lett.*, 39(3):147–151, 1991.
3. A. Cournier, A. K. Datta, F. Petit, and V. Villain. Enabling Snap Stabilization. *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS-03)*, 2003.
4. E. W. Dijkstra. Self-stabilization in spite of distributed control. *ACM Communications*, 17:643–644, 1974.
5. S. Dolev. *Self-Stabilization*. MIT Press, 2000.
6. S. Dolev and T. Herman. Superstabilizing Protocols for Dynamic Distributed Systems. *Chicago Journal of Theoretical Computer Science*, Vol. 1997, Article 4, 1997.
7. S. Ghosh, A. Gupta, T. Herman, and S. Pemmaraju. Fault-containing self-stabilizing algorithms. In *Proceedings of 15th Annual ACM Symposium on Principles of Distributed Computing (PODC '96)*, pages 45–54, 1996.
8. S. Ghosh, A. Gupta, and S. Pemmaraju. A fault-containing self-stabilizing algorithm for spanning trees. *Journal of Computing Information*, 2:322–338, 1996.
9. T. Herman. A comprehensive bibliography on self-stabilization. *Chicago Journal of Theoretical Computer Science*, 1996.
10. T. Herman and S. Pemmaraju. Error-detecting codes and fault-containing self-stabilization. *Information Processing Letters*, 73:41–46, 2000.
11. C. T. Huang and M. G. Gouda. State Checksum and Its Role in System Stabilization. *Proceedings of the 4th International Workshop on Assurance in Distributed Systems and Networks (ADSN 2005)*, 2005.