Fault Masking of Stabilizing Systems

MOHAMED G. GOUDA
Department of Computer Sciences
The University of Texas at Austin
and
JORGE A. COBB
Department of Computer Science
The University of Texas at Dallas
and
CHIN-TSER HUANG
Department of Computer Science and Engineering
University of South Carolina at Columbia
and
SRIKANTH SASTRY and SCOTT M. PIKE
Department of Computer Science
Texas A&M University

A redundant version of a system S is a system R that is specified from S as follows. First, system R 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 an error correction code encoded variable X in the corresponding process in system R. Third, the actions in each process in system S are modified before they are added to the corresponding process in system R and some new actions are added to the corresponding process in system R. In this paper, we show that a redundant version R of a system S has interesting stabilization and fault-masking properties. In particular, we show that if S is stabilizing, then R is also stabilizing. We also show that if R ever reaches stabilization, and then a "visible fault" occurs, then the effect of the fault is masked and the reached stabilization of R remains in effect.

Categories and Subject Descriptors: C.2.4 [**Distributed Systems**]: Distributed applications, Distributed databases, Network operating systems—*Stabilizing systems, shared memory, fault tolerance*
General Terms: Masking fault-tolerance, fault masking, shared memory, distributed computing, distributed systems, component based design, correctors, detectors, distributed systems, error correcting codes, stabilization, visible faults

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 [Dijkstra 1974; Dolev. 2000; Herman 1996].

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 [Ghosh et al. 1996; Ghosh et al. 1996; Herman and Pemmaraju 2000]. The second research direction is called fault-masking and it is the subject of the current paper. We compare these two research directions next.

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 $\mathrm{O}(m(f))$ 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 precedes 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 "redundant" version R such that R 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 [Dolev and Herman 1997] 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 [Cournier et al. 2003] 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 [Arora and Gouda 1993].

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 OF TRI-REDUNDANT SYSTEMS

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

THEOREM 4.1. (Stabilization of Tri-Redundant 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 IN TRI-REDUNDANT SYSTEMS

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 $(P' \wedge (\text{for each original copy } x \text{ in } T,\ x = x' \wedge x' = x''))$. 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 5.1. (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
$ds[i] \neq \min(n, ds[pr[i]] + 1) \rightarrow$
$ds[i] := \min(n, ds[pr[i]] + 1)$

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

This system has been shown to be stabilizing [Chen et al. 1991]. 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

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

$$
\begin{aligned}
&ds[i] > ds[g] + 1 \wedge \\
&(ds[i] = ds'[i] \wedge ds'[i] = ds''[i]) \wedge \\
&(pr[i] = pr'[i] \wedge pr'[i] = pr''[i]) \wedge \\
&(ds[g] = ds'[g] \wedge ds'[g] = ds''[g]) \\
&\qquad\qquad \rightarrow \\
&(ds[i], ds'[i], ds''[i]) := ds[g] + 1; \\
&(pr[i], pr'[i], pr''[i]) := g
\end{aligned}
$$

$$
\begin{aligned}
&(ds[i] \neq ds'[i] \vee ds'[i] \neq ds''[i]) \\
&\qquad\qquad \rightarrow \\
&(ds[i], ds'[i], ds''[i]) := \mathrm{MJR}(ds[i], ds'[i], ds''[i])
\end{aligned}
$$

$$
\begin{aligned}
&(pr[i] \neq pr'[i] \vee pr'[i] \neq pr''[i]) \\
&\qquad\qquad \rightarrow \\
&(pr[i], pr'[i], pr''[i]) := \mathrm{MJR}(pr[i], pr'[i], pr''[i])
\end{aligned}
$$

end

7. ERROR CORRECTING CODES

The previous sections illustrated a case study of masking visible faults in the context of tri-redundant systems using MJR correctors to restore the system to a safe state. We now extend the technique to give a more general characterization of masking visible faults, and of visible faults themselves.

Tri-redundant systems achieve fault detection and correction through data redundancy. Each bit $x[i]$ in the original system is replaced by three bits $x[i]$, $x'[i]$, and $x''[i]$ in the tri-redundant version. When a visible fault affects the system, the value of one of the bits among the three is changed. The non-agreement of values among the bits enables error detection. The MJR function compares the values among the three copies of the data to correct this fault. Data redundancy is the key to such error detection and correction.

There are many techniques for data redundancy that achieve varying degrees of error detection and correction. One such family of techniques is Error Correcting Codes [Hamming 1950], on which there has been considerable research ([Pless and Huffman 1998; Lin and Jr. 2004]). Some examples include Hamming Codes [Hamming 1950], Reed-Muller codes [Muller 1954], Golay Code [Golay 1949], SEC-DED codes [Hsiao 1970], BCH codes [Hocquenghem 1959; Bose and Ray-Chaudhuri 1960], and Reed-Solomon codes [Reed and Solomon 1960] [1].

Error correcting codes are schemes to encode data using redundant bits such that departure from this encoding (due to a fault) can be detected and corrected. Error correcting codes take a data item of length k bits, and create a larger data item

[1]This list is neither complete, nor representative of the error correcting codes described in current literature.

of length n bits. The redundancy resulting from the extra $(n - k)$ bits is used to recreate the original data in the event that some bits become corrupted.

Error correcting codes have a limit on the number of bit errors that they can correct. Let this bound be c. Thus, the effectiveness of an error correcting scheme is measured by the triple (n, k, c) [2]. For example, tri-redundant systems use a $(3, 1, 1)$ error correcting code. They use 3 bits (in the tri-redundant construction) to encode each bit (in the original system). The error correcting code, however, cannot a correct two-bit corruption if the two bits corrupted are from the same triple $(x[i], x'[i], x''[i])$. In the worst case, the error correcting scheme used in tri-redundant systems can correct no more than single-bit errors, hence the specification $(3, 1, 1)$.

Different degrees of error correcting capability, and space complexity can be achieved by varying the three parameters n, k, and c. Increasing n increases the redundancy in an error correcting code. This increases the space complexity because more bits are used to encode the k bits of original data. However, the increased redundancy allows for better error correction capability; therefore, with a sufficiently large n, the value of c increases as well.

Any error correcting code is operationally defined by four functions: *encode*, *decode*, *detect*, and *correct*.

i. *encode* takes a data item of length k and returns a data item of length n encoded with redundancy information. Function *encode* is one-to-one, *i.e.*, $encode(d) = encode(d')$ iff $d = d'$.

ii. *decode* takes a data item of length n and returns the original data item of length k. The function ensures that $decode(encode(d)) = d$. Furthermore, if the input to *decode* is not a possible output of *encode*, then *decode* returns an arbitrary value.

iii. *detect* analyzes an n-bit data item for bit-errors. It returns true iff its input is not a possible output of the function *encode*.

iv. *correct* takes any data item of length n and returns a data item of length n, under the following restriction. Let d and d' be data items of length n, d be a possible output of *encode*, and d' differ from d by at most c bits. Then, $correct(d') = d$. In effect, *correct* corrects up through c-bit errors in an n-bit data item.

8. REDUNDANT SYSTEMS

The four functions *encode*, *decode*, *detect*, and *correct*, described in the previous section, operate on data items of size k, or n bits (as applicable). However, to construct a redundant system R from a given system S, we need functions that can operate on variables of different sizes. We construct such functions as follows.

[2] Typically, error correcting codes are specified using the notation (n, k, d) where d is the minimum Hamming distance. The Hamming distance between two bit vectors is the minimum number of bits that need to be changed for one of vectors to be converted to the other. The error correcting capability of an (n, k, d) error correcting code is given by $c = \lfloor \frac{d-1}{2} \rfloor$. In order to simplify our presentation, we specify the error correcting codes with the parameters (n, k, c) instead of (n, k, d).

Let x be viewed as an array $x[0], x[1], ...$ where each $x[i]$ is a block of k bits. If the last block contains fewer than k bits, then it is padded appropriately to obtain a k-bit block. Let X be a variable viewed as an array $X[0], X[1], ...$ where each $X[i]$ is a block of n bits. Consider the following functions:

i. $X := Encode(x)$ assigns $encode(x[i])$ to the i^{th} element $X[i]$ of X.
The function $Encode$ encodes variable x to X

ii. $x := Decode(X)$ assigns $decode(X[i])$ to the i^{th} element $x[i]$ of x.
The function $Decode$ decodes variable X to x.

iii. $Correct(X)$ assigns $correct(X[i])$ to the i^{th} element $X[i]$ of X.
The function $Correct$ corrects any c or fewer bit errors in X. In fact, $Correct$ can correct larger bit errors in X as long as each block $X[i]$ has fewer than c-bit errors. In the worse case, the error correction capability is c bit errors or fewer.

iv. $Detect(X) = false$ iff $detect(X[i]) = false$ for all $X[i]$.
The function $Detect$ returns $true$ if there is at least one block $X[i]$ with c bit errors or fewer ($c \neq 0$).

We now use the above four functions to specify a redundant version R of any system S as follows.

i. *Topology:* System S has the same number of processes and the same topology as system R. Thus, there is a natural one-to-one correspondence between the processes in S and those in R. For convenience, each process p in S has the same name as that of the corresponding process p in R.

ii. *Variables:* For each variable x in a process p in system S, there is a variable X in the corresponding process p in system R such that $X := Encode(x)$.

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 R such that the following three conditions hold.
   (a) First, each occurrence of a variable x in $\langle \mathsf{guard} \rangle$ is replaced by $Decode(X)$, 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 $\neg Detect(X)$ to $\langle \mathsf{guard}' \rangle$.
   (c) Third, each statement of the form $x := \mathrm{E}(y, \dots)$ in $\langle \mathsf{assignment} \rangle$ is replaced by a statement of the form $(X := Encode(\mathrm{E}(Decode(Y), \dots))$ in $\langle \mathsf{assignment}' \rangle$. The latter statement computes the value of expression E, encodes it, and assigns the encoded value to X.

iv. *Additional Actions:* For each variable X in a process p in system R, add an action of the following form to process p in R

$$ac \ : \ (Detect(X)) \ \rightarrow \ X := Correct(X)$$

9. STABILIZATION OF REDUNDANT SYSTEMS

In Section 4, we showed the stabilization preserving property of the transformation of a system S to its tri-redundant variant T. Theorem 4.1 can be applied to redundant version R of S as follows.

THEOREM 9.1. (Stabilization of redundant systems)
Let S be a P-stabilizing system, and let R be a redundant version of S. R is Q-stabilizing, where Q is the predicate

$$P' \wedge (\textit{for every variable } X \textit{ in } R, Detect(X) = false)$$

and predicate P' can be obtained from P by syntactically replacing each variable x in P with $Decode(X)$ (where $X = Encode(x)$) in P'.

PROOF. The proof is similar to the proof of Theorem 4.1. The proof can be derived from the proof for Theorem 4.1 by replacing $(x \neq x' \vee x' \neq x'')$ with $(Detect(X) = true)$, replacing $(x = x' \wedge x' = x'')$ with $(Detect(X) = false)$, replacing $(x, x', x'') := E(y, ...)$ with $X := Encode(E(Decode(Y), ...))$, and replacing $(x, x', x'') := \text{MJR}(x, x', x'')$ with $X := Correct(X)$. □

10. FAULT-MASKING OF REDUNDANT SYSTEMS

Let S be a P-stabilizing system and let R be a redundant version of S (constructed through the transformation described in Section 8). From the stabilization theorem of redundant systems (Section 9), R is Q-stabilizing where Q is the predicate ($P' \wedge$ (for every variable X in R, $Detect(X) = false$)). In this section, we argue that if R is in a legitimate state, where Q is true, and then some fault, from a class of faults called (n, k, c)-visible faults, occurs, then the effects of the fault are masked and the system quickly returns to a legitimate state, where Q is true. We start by defining (n, k, c)-visible faults.

A fault f is (n, k, c)-visible iff it changes the values of some variables in system R such that the following two conditions hold:

i. *Legitimacy:* Immediately before f occurs, system R is at a legitimate state where predicate Q is true. It follows that for every variable X in R, $Detect(X) = false$.

ii. *Transparency:* For every variable X in R,

$$Correct(X_b) = Correct(X_f),$$

where X_b is the value of X immediately before fault f occurs and X_f is the value of X immediately after f occurs.

Assume that an (n, k, c)-visible fault f occurs in system R, and also assume that f changes the value of some variable X in R from X_b to X_f. From the legitimacy condition of f, $Detect(X_b) = false$. Thus, from the transparency condition of f and from the fact that f has changed the value of X, $Detect(X_f)$ must be *true*.

Let t be the state of system R immediately after f occurs. Then, the predicate $Detect(X)$ is true at state t. System R has two types of actions where the variable X occurs: actions ac_0, ac_1, ... that correspond to the actions in system S where x

occurs, and the added action ac:

$$ac \ : \ (Detect(X)) \ \rightarrow \ X := Correct(X)$$

The guard of each action ac_i in R has a conjunct $(\neg Detect(X))$ and so none of these actions can be executed until after action ac is executed. From the transparency condition of f, executing action ac corrects the value of X from X_f to X_b. Thus, the effect of fault f on X, and ultimately on system R, is masked. This argument proves the following theorem.

THEOREM 10.1. (Fault-Masking of Redundant Systems)
Let S be a P-stabilizing system and R be a redundant version of S. System R is F-masking, where F is the class of (n, k, c)-visible faults.

□

In other words, system R can mask transient faults that corrupt no more than c bits of any variable X in R. Recall that each variable X in R consists of blocks of length n bits each. Let there be B such blocks in system R. Provided each (n, k, c)-visible fault affects a distinct n-bit block each time, system R can mask up to B (n, k, c)-visible faults occurring before R can execute any program action. This is however, the best case scenario for fault masking; in the worst case, system R can mask no more c corrupted bits in the system.

11. EXAMPLE: USING HAMMING CODES

As an example, we now outline the construction of a redundant system R from a system S using a specific error correcting code. For the purpose of illustration, we consider the simplest non-trivial error correcting code *viz.* (7,4) Hamming Code.

The (7,4) Hamming code encodes a block of 4-bit word to a block of 7-bit code, and can detect and correct up to a single bit error, but can only detect 2 bit errors. Therefore, the parameters of the error correcting code is given by $n = 7$, $k = 4$, and $c = 1$. The encoding, decoding, detection, and correction operations are performed as follows.

For all Hamming codes, encoding is performed using a *generator matrix* (G), decoding performed using a *receiver matrix* (R), and detection and correction is performed using a *parity matrix* (H). For (7,4) Hamming code, the generator, receiver and parity matrices are:

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad R = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \quad H = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{pmatrix}$$

The *encode* function is defined as follows.

$$encode(x) = G \times x$$

Note that x is the 4-bit word treated as a column vector. The result is a row vector that is treated as the encoded 7-bit word.

The *decode* function is defined as follows.

$$decode(X) = R \times X$$

Note that X is 7-bit code treated as a row vector, and the result is a column vector that is treated as the decoded 4-bit word.

The *detect* function is defined as follows.

$$detect(X) = H \times X$$

The function $detect(X)$ returns a zero matrix if there is no error in X, else it returns a non-null column matrix.

The *correct* function is defined as follows.

$$correct(X) = X + detect(X)$$

The addition in the above expression is modulo-2 addition. Note that if X is error free, then $detect(X)$ returns a zero matrix, thus $correct(X) = X$ if X is error-free.

The above functions can then be used to construct a redundant system R from any system S as described in Section 8.

12. DISCUSSION

In general terms, the technique employed to transform a system S to a redundant system R that achieves fault masking with respect to visible faults can be described as follows.

i. Replace every variable x in system S with a corresponding variable X in system R using a transformation that embeds data redundancy into the variable. The tri-redundant transformation accomplishes this through triplication, while the redundant system transformation accomplishes this through the *Encode* function described in Section 8.

ii. Strengthen the guard in every action of system R using a set of predicates sufficient to detect a visible fault. In a tri-redundant system R, the predicate is of the form $(x = x' \wedge x' = x'')$ for each triple (x, x', x'') in R. In redundant system R, the predicate is of the form $Detector(X)$ (described in Section 8) for each variable X in R.

iii. When the detector predicates detect a visible fault in some variable X, disable all the program actions that read or write the variable X.

iv. Invoke a set of corrective actions to repair the visible fault. In tri-redundant system, this is accomplished through the actions of the form $(ac : x \neq x' \vee x' \neq x'' \rightarrow (x, x', x'') := \text{MJR}(x, x', x''))$, and in redundant systems, through actions of the form $(ac : (Detect(X)) \rightarrow X := Correct(X))$.

v. After the visible fault has been repaired, enable the program actions to resume normal execution.

It is shown in [Arora and Kulkarni 1998b] that any fault tolerant system can be viewed as a composition of a fault intolerant system and a set of *detectors* and *correctors*. Detectors are predicates that evaluate to true when a transient fault occurs in the system. Correctors are actions that repair these transient faults.

Detectors and correctors achieve fault tolerance as follows: Upon detecting a transient fault that puts the system in an unsafe state, the detectors disable the program actions, and enable the correctors. The corrector actions repair the fault and drive the system to a safe state. Upon reaching a safe state, the ordinary program actions resume execution. The above mechanism is used in [Arora and Kulkarni 1998a] and [Kulkarni and Ebnenasir 2003] to transform fault intolerant systems to masking fault tolerant systems. The detectors and correctors described in our presentation are used to preserve stabilization while adding fault masking from visible faults.

Previous work on fault containment have used error detecting codes [Herman and Pemmaraju 2000]. The approach used in these methods is to disable the program actions when an error is detected by the error detecting codes, and overwrite the corrupted value in the variable with some legitimate value. Such overwriting may not result in a safe state. The system then relies on the self-stabilizing property of its program actions to restore the system to a safe state.

Our approach on the other hand, does not depend on the stabilizing properties of the system it transforms; it merely preserves these properties. This suggests that the fault masking techniques described in this paper can be applied to non-stabilizing systems as well, to achieve fault masking with respect to visible faults.

13. CONCLUDING REMARKS

In this paper, we described a transformation to transform any system S to a redundant version R. We showed that if S is stabilizing then R is both stabilizing and F-masking for the class F of visible faults. Any non-visible faults that affect the system may be assumed to be repairable by the stabilizing properties of the system.

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 R is stabilizing and F-masking under the same assumption that any subset of actions (at most one action from each process) in R are executed at a time.

In the above presentation, we started with a system S, applied a tri-redundant transformation to get system T. System T used three times the number of bits that S does, and masks a single bit transient fault in the worst case. We then extended this transformation using error correcting codes to yield a family of transformations based on the choice of the error correcting scheme, the amount of redundancy in the data, and the worst case severity of maskable transient faults.

The transformation of fault masking presented in the current paper is based on error correcting codes. There is a vast body of research in the areas of information theory and cryptography that focus on data integrity, correctness, and repairability. Results from these areas could be explored to enrich the techniques and frameworks for fault masking in distributed systems.

In [Huang and Gouda 2005], 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 work of Srikanth Sastry and Scott M. Pike is supported by the Advanced Research Program of the Texas Higher Education Coordinating Board under Project Number 000512-0007-2006. 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

Arora, A. and Gouda, M. 1993. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering 19,* 11, 1015–1027.

Arora, A. and Kulkarni, S. S. 1998a. Designing masking fault-tolerance via nonmasking fault-tolerance. *IEEE Transactions on Software Engineering 24,* 6, 435–450.

Arora, A. and Kulkarni, S. S. 1998b. Detectors and correctors: A theory of fault-tolerance components. In *Proceedings of the The 18th International Conference on Distributed Computing Systems*. IEEE Computer Society, Washington, DC, USA, 436–443.

Bose, R. C. and Ray-Chaudhuri, D. K. 1960. On a class of error correcting binary group codes. *Information and Control 3*, 68–79.

Chen, N.-S., Yu, H.-P., and Huang, S.-T. 1991. A self-stabilizing algorithm for constructing spanning trees. *Information Processing Letters 39,* 3, 147–151.

Cournier, A., Datta, A. K., Petit, F., and Villain, V. 2003. Enabling snap-stabilization. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems*. IEEE Computer Society, Washington, DC, USA, 12–19.

Dijkstra, E. W. 1974. Self-stabilizing systems in spite of distributed control. *Commun. ACM 17,* 11, 643–644.

Dolev., S. 2000. *Self-Stabilization*. MIT Press, Cambridge, MA.

Dolev, S. and Herman, T. 1997. Superstabilizing protocols for dynamic distributed systems. *Chicago Journal of Theoretical Computer Science 1997,* 4.

Ghosh, S., Gupta, A., Herman, T., and Pemmaraju, S. 1996. Fault-containing self-stabilizing algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*. ACM Press, New York, NY, USA, 45–54.

Ghosh, S., Gupta, A., and Pemmaraju, S. 1996. A fault-containing self-stabilizing algorithm for spanning trees. *Journal of Computing Information 2,* 1, 322–338.

Golay, M. J. E. 1949. Notes on digital coding. *Proceedings of IRE 37*, 657.

Hamming, R. W. 1950. Error detecting and error correcting codes. *Bell Systems Technical Journal 29*, 147–160.

Herman, T. 1996. A comprehensive bibliography on self-stabilization. `http://www.cs.uiowa.edu/ftp/selfstab/bibliography`.

Herman, T. and Pemmaraju, S. 2000. Error-detecting codes and fault-containing self-stabilization. *Inf. Process. Lett. 73,* 1-2, 41–46.

Hocquenghem, A. 1959. Codes corecteurs dérreurs. *Chiffres 2*, 147–156.

Hsiao, M. Y. 1970. A class of optimal minimum odd-weight-column sec-ded codes. *IBM Journal of Research and Development 14*, 395.

Huang, C.-T. and Gouda, M. G. 2005. State checksum and its role in system stabilization. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Workshops*. Fourth International Workshop on Assurance in Disributed Systems and Networks (ADSN). IEEE Computer Society, Washington, DC, USA, 29–34.

Kulkarni, S. S. and Ebnenasir, A. 2003. Enhancing the fault-tolerance of nonmasking programs. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems*. IEEE Computer Society, Washington, DC, USA, 441–449.

Lin, S. and Jr., D. J. C. 2004. *Error control coding*, 2nd ed. Pearson Prentice Hall, Upper Saddle River, NJ, USA.

Muller, D. E. 1954. Applications of boolean algebra to switching circuits design and to error detection. *IRE Transactions EC-3*, 38–49.

Pless, V. S. and Huffman, W. C., Eds. 1998. *Handbook of coding theory: Volume I and II*. Elsevier Science B. V., Amsterdam, The Netherlands.

Reed, I. S. and Solomon, G. 1960. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics 8*, 300–304.