INFORMATION PROCESSING 68 - NORTH-HOLLAND PUBLISHING COMPANY - AMSTERDAM (1969)

DESIGN OF DYNAMICALLY CHECKED COMPUTERS

WILLIAM C. CARTER and PETER R. SCHNEIDER
IBM Watson Research Center,
Yorktown Heights, New York, USA

A model for discussing dynamically checked logic circuits is used to establish the characteristics required for combinational, sequential and checking circuits to be self-testing during normal operation. It is proved that a totally checked computer can be designed using synthesis techniques based on iteration between tentative function designs and checker designs. This iteration process is controlled by a probabilistic means for evaluating the effectiveness of the dynamic checking. Methods for system use of the checking circuit's outputs are discussed.

1. PROBLEM

The design of computers with near perfect dynamic failure* detection requires knowledge of how to:

1) encode logic signals and generate the corresponding functional transformations;
2) implement both combinational and sequential circuits which produce non-code outputs when they fail; and
3) design checking circuits which, during normal computer processing signal both the occurrences of errors* in the regular function circuits and failures in the checker itself.

Allied with these are the problems of developing design evaluation procedures and system operational procedures using the checkers to locate failures and initiate retry or configuration.

2. MODEL

The general circuit model will be as shown in fig. 1 with:

$$X \equiv \{x^k = (x_1^k, \ldots, x_n^k) \mid x_i^k \in \{0,1\}\}$$

$$S \equiv \{s^k = (s_1^k, \ldots, s_m^k) \mid s_i^k \in \{0,1\}\}$$

$$Y \equiv \{y^k = (y_1^k, \ldots, y_r^k) \mid y_i^k \in \{0,1\}\}$$

$$Z \equiv \{z^k = (z_1^k, \ldots, z_q^k) \mid z_i^k \in \{0,1\}\}$$

and

$$\tau : (X, S) \to Y \quad \sigma : (Y, S) \to S \quad \omega : (X, S) \to Z.$$

This division may be somewhat arbitrary but will allow certain results concerning testability to be proved. Any variations must then be judged in light of their impact on the overall testability. The model is assumed to be synchronous, i.e. the σ mapping occurs at precise well defined times.

Variables before encoding will be distinguished by placing a caret ($\hat{}$) above them. The one-to-one, in-to injections which perform the encoding are,

$$\mu_x : \hat{X} \to X, \; n \geq \hat{n} \qquad \mu_y : \hat{Y} \to Y, \; r \geq \hat{r},$$

$$\mu_s : \hat{S} \to S, \; m \geq \hat{m} \qquad \mu_z : \hat{Z} \to Z, \; q \geq \hat{q}.$$

For notational convenience it will be said that for all $\hat{x}^k \in \hat{X}$, $\mu_x(\hat{x}^k) \equiv x^k \in X$, i.e. the superscript identifiers will be preserved. The range of μ_x, $\mu_x(\hat{X}) \subset X$, will be the *code space* associated with X and the co-range of μ_x, $[X \cap \sim \mu_x(\hat{X})] \subset X$, will be the detectable *error space* for X. Similar statements will be applicable for S, Y and Z.

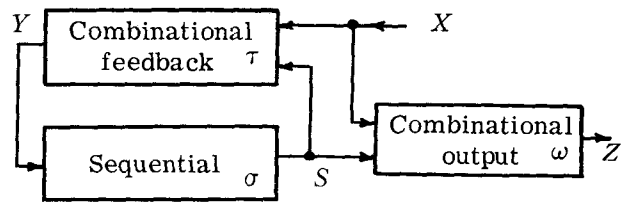

Fig. 1. Circuit model.

* *Failure* will refer to a mutation of the correct machine and *error* will mean an incorrect logic signal due to a failure.

3. MAPPINGS

Using these μ encodings, the mappings τ, σ and ω for the encoded variables are defined on the code spaces of S, X, Y and Z through a straightforward extension of the $\hat{\tau}$, $\hat{\sigma}$ and $\hat{\omega}$ mappings defined on $\hat{S}$, $\hat{X}$, $\hat{Y}$ and $\hat{Z}$. The encoded functional mappings are *not* defined for any situation where x^k or s^i or y^j are elements of the error space of X, S and Y respectively; these are considered to be classical don't care conditions.

Fig. 2 presents a pictorial description of what has happened in defining these new mappings for the particular function τ. Under error-free operation the mappings will always occur between elements in the unshaded, code space areas. The two basic Don't Care Assignment (DCA) policies are given below.

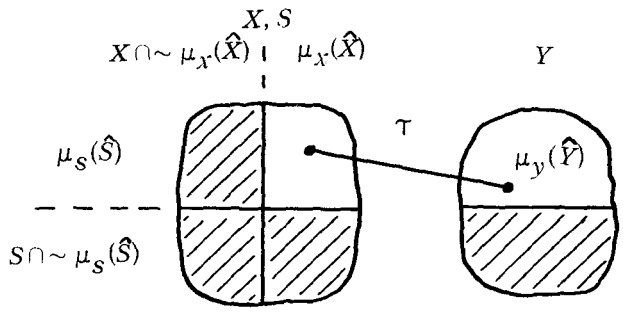

Fig. 2. Mapping after encoding.

DCA-1

Always have domain error space elements mapped to range error space elements. Then if an erroneous piece of data (an $x^k \notin \mu_X(\hat{X})$) is injected into the system, the mappings will be transferred into the error space and will stay there unless another failure occurs! This is a severe restriction on the implementation but permits checking to occur only at the Z output since errors persist and propagate.

DCA-2

Allow domain error space elements to be mapped to range code space elements. This provides greater freedom of implementation but requires checking at all interfaces if undetected errors are to be prevented.

Intermediate between two extremes are the situations where some of the mappings use DCA-1 and others DCA-2. Then, checking must always occur at the interface preceding a DCA-2 mapping if all errors are to be detected.

The previous remarks serve as a basis for proof of the following theorem.

Theorem 1. Necessary conditions for dynamically detecting all single failures which can produce errors in the circuit outputs are that all interface variables $\hat{X}$, $\hat{S}$, $\hat{Y}$ and $\hat{Z}$ must be encoded so that $n > \hat{n}$, $m > \hat{m}$, $r > \hat{r}$ and $q > \hat{q}$ and that checking must occur at all interfaces preceding a DCA-2 mapping.

Even when all computer mappings use DCA-1, it is still advisable to check at more interfaces than just the final outputs. Otherwise, the system might have to operate for a long time after the failure in order to propagate the failure to the final outputs. Also, the state of the machine may be so far removed from the correct state as to make an algorithmic failure analysis impossible.

4. CHECKING

Once the variables have been encoded by the mappings μ_X, μ_S, μ_Y and μ_Z it is possible to define corresponding checking functions α_X, α_S, α_Y and α_Z. For simplicity, only the α_X check on X will be defined in detail; the others follow directly.

$$\alpha_X : X \to C \equiv \{c^k = (c_1^k, \ldots, c_\nu^k) \mid c_i^k \in \{0, 1\}\}.$$

By making α_X a combinational logic function of the interface variables x_i the resulting dynamic check is restricted to the type which is performed immediately by sampling the existing interface data, i.e. makes no use of the history of previous performance.

The restrictions on α_X are:

1) the check on the code space and the error space must be distinguishable, i.e.
$$[\alpha_X(\mu_X(\hat{X}))] \cap [\alpha_X(X \cap \sim \mu_X(\hat{X}))] = \phi;$$
2) α_X must have some well defined and easily used property; and
3) an implementation of α_X must exist which is *self-tested using inputs from the code space* of X.

Theorem 2. Necessary conditions for a dynamic checker to meet the above restrictions are that $\nu \geq 2$ and each $c_i = 1, \ldots, \nu$ take on the values 0 and 1 for at least one x^k in the code space.

Proof. Restrictions 1 and 3 imply $\nu \geq 2$, otherwise the final circuit output is untested for failure to the one value it uses for all code space inputs. Similarly, if any c_i always

takes on one particular value, say 1, for all $x^k \in \mu(X)$ then the ith output line of the checker is not tested for being stuck at that value, say stuck-at-1, using code space elements. Thus each checker output must take on both 0 and 1 values for code space inputs. Q.E.D.

For simplicity, this paper will restrict its examples to $\nu = 2$. If $\{A\}$ and $\{B\}$ are subsets spanning $\{x_1, \ldots, x_n\}$, then checkers which meet the above criteria are of the form $c_1(A)$, $c_2(B)$ with c_1, c_2 having, for example, odd parity for code space inputs and even parity for error space inputs. Each circuit tree must be completely checked by the respective $\{A\}$ and $\{B\}$ portions of the code space inputs.

Assume that the X interface contains nine variables $x_1, \ldots, x_9$ and that code vectors have odd parity. Then fig. 3 is a dynamic checking circuit of the required type with $A = \{x_1x_3x_5x_7x_9\}$ and $B = \{x_2x_4x_6x_8\}$. It is clear that each circuit tree is exhaustively tested, i.e. sees all possible input combinations during the occurrences of normal code space elements at the interface.

Next, assume the X interface contains the five variables $x_1, \ldots, x_5$ in a 2-out-of-5 code. The left half of table 1 shows the code space portion for a particular α_X. If the error space conditions with less than 2 ones are mapped to 0, 0 and those with more than 2 ones are mapped to 1, 1, the circuit in fig. 4 can be shown to meet all the above requirements. The right portion of table 1 indicates this by denoting for each code space input (row) the circuit lines and failures tested by that input, e.g. 00011 tests a_1 and c_1 stuck-at-1 and b_1 and c_2 stuck-at-0. Each circuit line (column) is tested by at least one code input. All inputs are also tested.

The final example in fig. 5 is a checker which operates on 2 two-rail encoded line pairs, i.e. $x_1 = \bar{x}_2$ and $\bar{x}_3 = x_4$ in the code space. It generates an even parity output if either or both rails have identical values on their pair of lines or if the inputs are correct and a failure in the circuit exists. Such circuits can be interconnected in a tree to check an interface of arbitrary size.

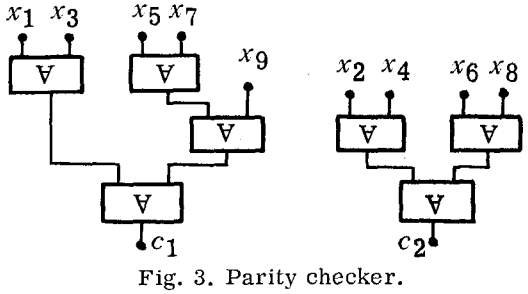

Fig. 3. Parity checker.

Table 1
Function and lines tested

Function							Lines tested									
x_1	x_2	x_3	x_4	x_5	c_1	c_2	a_1	a_2	a_3	a_4	b_1	b_2	b_3	b_4	c_1	c_2
0	0	1	0	1	1	0	0	0	0	0	1	1	1	1	0	1
0	1	0	0	1	1	0	0	0	0	0	1	1	1	1	0	1
1	0	0	0	1	1	0	0	0	0	0	1	1	1	1	0	1
0	0	1	1	0	1	0	0	0	0	0	1	1	1	1	0	1
0	1	0	1	0	1	0	0	0	0	0	1	1	1	1	0	1
1	0	0	1	0	1	0	0	0	0	0	1	1	1	1	0	1
0	0	0	1	1	0	1	1				0				1	0
0	1	1	0	0	0	1		1				0			1	0
1	0	1	0	0	0	1			1				0		1	0
1	1	0	0	0	0	1				1				0	1	0

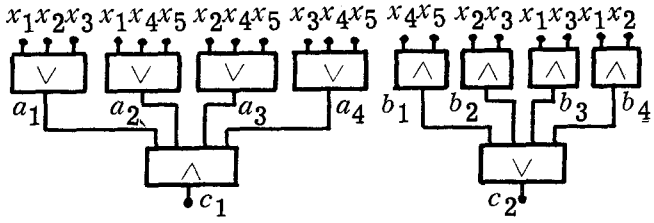

Fig. 4. A 2-out-of-5 checker.

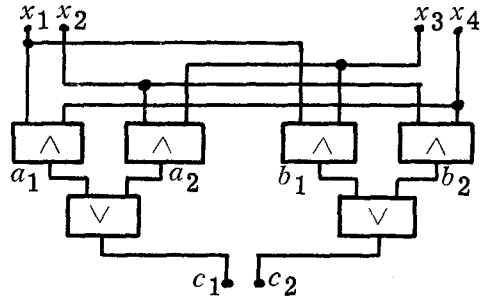

Fig. 5. Two-rail code checker.

After implementing several checkers of the above type it becomes clear that restrictions 1 and 2 are easy to satisfy; restriction 3 is not. Although design algorithms do not yet exist, many guidelines and rules do [1, 2].

5. COMBINATIONAL MAPPINGS

The requirements are:

1) to have every single circuit failure manifest itself as an interface error for at least one code input, and

2) never have single failures produce interface errors resulting in an erroneous code output.

Theorem 3. Given $\hat{\tau}$ and $\hat{\omega}$, there exist encodings and implementations of τ and ω which meet the above requirements.

Proof. Use a parity encoding and implement each output as a separate, independent logic tree. Alternately, use a classic two-rail implementation. Q.E.D.

When implemented using DCA-2 these examples require slightly less than duplication. They also require checking at the input interface, the one source of commonality for the logic trees or the two rails. Although more efficient encodings and mappings may exist, unfortunately, algorithms for obtaining them do not. A completely checked n input, 2^n output decoder, designed according to these procedures, appears in ref. [2].

The great similarity between the design of checkers and combinational circuits should be clear: checking functions are just special DCA-1 combinational mappings.

6. SEQUENTIAL MAPPINGS

The requirements are the same as those for combinational mappings. Liu [3] proved that it is always possible to augment a state table so that the augmented table can be realized as a set of h k-stage Non-linear Feedback Shift Registers (NFSR). Over the Galois field of characteristic 2 the column vector

$$\hat{S} = (\hat{s}_{11}, \ldots, \hat{s}_{1k}; \ldots; \hat{s}_{i1}, \ldots, \hat{s}_{ik}; \ldots; \hat{s}_{hi}, \ldots, \hat{s}_{hk}),$$

$$\hat{m} = h \times k,$$

specifies the present state of the NFSR. Let A be the binary matrix whose elements are 1 only for subscripts $(i, i+1)$, i not a multiple of k, and let $\hat{u}$ be the Boolean function vector with nonzero elements $\hat{y}_i = \hat{\tau}(\hat{X}, \hat{S})$ only at those positions where i is a multiple of k. The next state of an NFSR is then defined by the linear operator $\hat{\sigma}$ through the equation

$$\hat{\sigma}\hat{S} = A\hat{S} \vee \hat{u}.$$

Fig. 6 shows such an NFSR.

If T is a subset of the subscripts of the S vector, then the set of shift register elements $s[T]$ satisfy a parity encoding iff

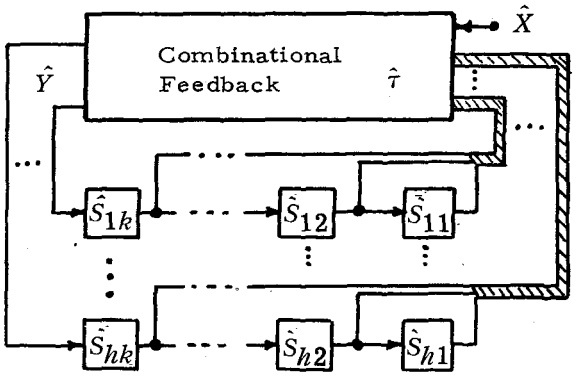

Fig. 6. NFSR.

$$\forall(\sigma s[T]) = \forall(s[T])$$

i.e., the parity of this set of shift register elements is unchanged during the transitions from state to state. Using this definition and the $\hat{\sigma}$ operator, various encodings μ_S may be devised. For example, let $T_i = \{i, 1; \ldots; i, k\}$ and define $\sigma s[T_i]$ directly from $\hat{\sigma}\hat{s}[T_i]$. Then define the parity encoding μ_S on each shift register by adding a new cell s_{ik+1} to each register with

$$\sigma s_{ik+1} = s_{ik+1} \ \forall(\forall\sigma s[T_i]) \ \forall(\forall s[T_i]).$$

Using the equation defining $\hat{\sigma}$ (also σ)

$$\sigma s_{ik+1} = s_{ik+1} \ \forall s_{i1} \ \forall \tau_i(X, S).$$

Fig. 7 shows a typical encoded register.

The previous construction indicates proof of the following:

Theorem 4. Given $\hat{\sigma}$, there exist encodings and implementations of σ which meet the above requirements.

A corollary to this theorem, based on a modified construction, is that as few as 1 extra cell need be added for encoding.

In these encodings, parity, either even or odd, is preserved. A single error will change the parity and cause error space conditions to persist

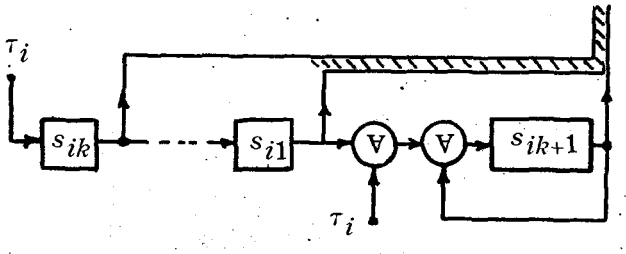

Fig. 7. Encoded register.

until another error occurs. Thus a solid failure may cause recurring errors which, in turn, cause the parity (error space conditions) to oscillate.

Massey and Liu [4] showed how to perform transformations between an NFSR with an external input only at one end and an NFSR with a variable number of inputs added (∀) between stages of the register. Although their work pertained to special NFSR's, it can be generalized to the set of NFSR's used here in order to obtain circuitry tradeoffs. In particular, several simple encodings exist which result in registers similar to that in fig. 7 except the function τ is modified and the exclusive-OR with input τ_i is not required.

7. EVALUATION

For ease of discussion consider the Z interface. A problem of major concern in evaluating any design is the determination of the probability that a given failure in either ω (denoted $F(\omega)$) or α_z (denoted $F(\alpha_z)$) will be detected. The following probabilities, conditional on the existence of the given F, define the pertinent information at some given time.

p_1 = Pr. [error in $\omega \mid F(\omega)$]

p_2 = Pr. [error at Z interface $\mid F(\omega)$]

p_3 = Pr. [Z in error space $\mid F(\omega)$]

p_4 = Pr. [error in $\alpha_z \mid F(\alpha_z)$]

p_5 = Pr. [error at checker output $\mid F(\alpha_z)$]

p_6 = Pr. [checker signals error $\mid F(\alpha_z)$]

p_7 = Pr. [checker signals error $\mid z$ in error]

Since it is necessary that an error exist somewhere in the circuit before it can reach an output and since an error must exist at the outputs before the output can enter the error space, the following ordering exists:

$$p_1 \geq p_2 \geq p_3, \qquad p_4 \geq p_5 \geq p_6.$$

To enable some simple computations, assume the computing process to be stationary and independent. Then for each p_i a $P_i(T)$, representing the occurrence of the event in question in time T given the failure F, is given as

$$P_i(T) = 1 - (1-p_i)^{T \times R} \qquad i = 1, \ldots, 6$$

where R is the cycle rate.

The ideal design criterion for the function circuits is to have $p_1 = p_2 = p_3$ with $P_3(T) \to 1$ for $T \ll$ MTBF. An acceptable alternative is $p_1 \geq p_2$ providing the other conditions hold. In the latter case undetected errors can exist in the circuitry but the failure will always be detected as soon as errors affect the interface data and it is still assured that the ultimate detection occurs before a second failure can develop. Any design for which $p_2 > p_3$ must be viewed very carefully since undetected interface errors could exist. Similar criteria can be specified for the checking circuits using p_4, p_5 and p_6. An ideal checking circuit design should also have $p_7 = 1$ although exceptions may be allowed. For example, if it is *impossible* for the Z interface to reach a given z^k in the single failure environment and $P_3(T) \to 1$ for $T \ll$ MTBF, then it might be permissible to design α_z so that $\alpha_z(z^k)$ is in the code space of the checker output, i.e. no error is signalled.

When the circuits are analyzed it may be that $P_i(T)$ does not approach 1 fast enough because certain key interface elements occur with a very low probability during normal operation. Then it will be necessary to revise the circuit designs or introduce periodic diagnosis or interface exercising to force $P_i(T) \to 1$.

As an example consider the 2-out-of-5 checker in fig. 4 which has $p_4 > p_5 = p_6$ and $p_7 = 1$. For the case of equally likely inputs it can be shown that $p_6 \geq \frac{1}{10}$. This means $P_6(T)$ approaches 1 rapidly for a T corresponding to about 100 machine cycles. Now assume that code input 00011 occurs with probability 10^{-12}. Since this input is a unique test for the failure a_1 stuck-at-1 (cf. table 1), it can be shown that $P_6(T) < 1$ for T corresponding to over 10^{12} machine cycles. Four inputs have this property of being unique tests for certain failures and if any one occurs with a low probability, trouble ensues. However, symmetry can be used to redesign the circuit so that a different four inputs (with higher probability of occurrence) have the property of being the unique tests.

8. SYNTHESIS

The synthesis approach used to date has been to examine many classes of encodings of the type described in section 2 and, for each, discover whether a dynamic checking circuit of the type discussed in section 4 exists. This builds a library of useable encodings and corresponding checking circuits. Then, when it comes to designing the combinational and sequential function circuits, the synthesis process starts with the libra-

ry of permissible encodings and, in an iterative process, attempts to design a testable circuit as outlined in sections 5 and 6. If it is not possible to generate an implementation, it becomes necessary to expand the library or restructure the system architecture.

9. SYSTEM OPERATION

Consider a self-repairing computer system where, for each functional unit, at least one identical, standby spare is provided [2]. Fig. 8 shows a typical system interface involving two Sender and two Receiver units, each designed according to procedures given in this paper, with the input section of Receiver 2 detailed for purposes of illustration. To keep the interface simple it is assumed that only one data line per Sender unit exists; extension to more lines is straightforward. The dual output checking circuits in each Sender drive a Status Register and Encoder that records which Sender has not failed and generates an error-correction encoded signal to specify this unit. The Decoder in each receiver accepts this encoded signal and generates a 0 if Sender 1 is to be used and a 1 if Sender 2 is to be used. The logic tree then selectively gates the information into the functional portion of the Receiver. Such an interface can be made highly failure-tolerant; failures in the Senders are removed by having the checkers change the "status"; failures in the Status Register and Encoder are masked through the redundant nature of the transformation; and failures in the Decoder/logic tree are indistinguishable from those in the actual function, hence are handled at the next interface.

In other designs, the dual outputs of the many checkers throughout the system could be reduced (using the circuit in fig. 5) to one pair which triggers an instruction retry or machine check.

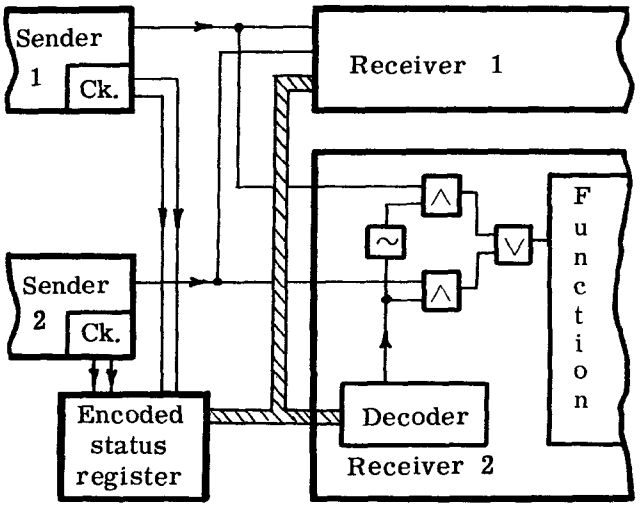

Fig. 8. System interface.

It might also be desirable to automatically log the state of all the checkers after a failure for subsequent use by the programmer or maintenance personnel. Once the concept of total self-testing is available, the variations for system use proliferate.

REFERENCES

[1] J. P. Roth, W. G. Bouricius and P. R. Schneider, Programmed Algorithms to Compute Tests to Detect and Distinguish Between Failures in Logic Circuits, IEEE TEC, Vol. 16 (1967) 567.

[2] J. P. Roth, W. G. Bouricius, W. C. Carter and P. R. Schneider, Phase II of an Architectural Study for a Self-Repairing Computer, SAMSO TR-67-106 (1967).

[3] C. L. Liu, Sequential-Machine Realization Using Feedback Shift Registers, Proc. 5th Annual Symp. on Switching Circuit Theory and Logical Design, October 1964, p. 209.

[4] J. L. Massey and R. W. Liu, Equivalence of Nonlinear Shift-Registers, IEEE Trans. IT, Vol. 10 (1964).