for $n \geq 64$ is that to reduce the area the two-rail checker C is implemented as a tree with an extra PLA level.

IV. Conclusion

We have presented a new design for TSC 1-out-of-n code checkers. It is shown that for many values of $n < 1025$ our design would have less hardware cost than that of an existing scheme [1]. In fact, for a few important cases (when $n \leq 32$ is a power of 2), our checker is also faster than the other design.

Since the initial conclusion of this work, two new designs for TSC 1-out-of-n code checkers have come to the author's attention. The first is a work by Kotočová [5]. Here a TSC design for some 1-out-of-n code checkers has been described. The checker has three gate delays and can be minimized according to the number of gates or the number of gate inputs. The n input lines are grouped according to certain rules and are fed into, say, an OR-AND-OR circuit. No estimate of hardware cost is provided. The second design, described in [9], also groups the n inputs, but now into two matrices. Then algebraic manipulations, such as shifts and scalar multiplication, are done on the columns of these matrices according to an algorithm. This is then translated into a circuit implementation which always has four gate delays, and is shown to be superior to Anderson's design in the total number of gate inputs that it requires.

Finally, none of the works cited so far, including this one, provides a TSC checker for the 1-out-of-3 code. David has designed such a checker using memory elements [3]. This design involves a translation from the 1-out-of-3 code into the 1-out-of-4 code, using two bits of memory. The resulting code can then be checked by a TSC checker for the 1-out-of-4 code. Recently, another solution to this problem has been suggested [4]. Here the 1-out-of-3 code is combined with any available m-out-of-n code to obtain a reduced $(m + 1)$-out-of-$(n + 3)$ code with p code words. Then the resulting code is translated into the 1-out-of-p code, for which a TSC checker can be designed. To date, however, no combinational TSC checker has been designed for the 1-out-of-3 input.

Acknowledgment

Many thanks to Prof. E. J. McCluskey for his guidance and support. Also, special thanks to Prof. J. Wakerly for his many invaluable comments and suggestions.

References

[1] D. A. Anderson and G. Metze, "Design of totally self-checking check circuits for m-out-of-n codes," *IEEE Trans. Comput.*, vol. C-22, pp. 263–269, Mar. 1973.

[2] W. C. Carter and P. R. Schneider, "Design of dynamically checked computers," in *Proc. 4th Congress IFIP*, vol. 2, Edinburgh, Scotland, Aug. 5–10, 1968, pp. 878–883.

[3] R. David, "Totally self-checking 1-out-of-3 checker," *IEEE Trans. Comput.*, vol. C-27, pp. 570–572, June 1978.

[4] P. Golan, "A new totally self-checking checker for 1-out-of-3 code," in *Proc. 4th Int. Conf. on Fault-Tolerant Syst. Diagnostics*, Brno, Czechoslovakia, Sept. 28–30, 1981, pp. 246–247.

[5] M. Kotočová, "Design of totally self-checking check circuits for some 1-out-of-n codes," in *Proc. 4th Int. Conf. on Fault-Tolerant Syst. Diagnostics*, Brno, Czechoslovakia, Sept. 28–30, 1981, pp. 241–245.

[6] M. A. Marouf and D. A. Friedman, "Efficient design of self-checking checkers for m-out-of-n codes," in *Proc. 7th Annu. Symp. on Fault-Tolerant Comput.*, Los Angeles, CA, June 28–30, 1977, pp. 143–149.

[7] V. I. Maznev, "Design of self-checking $1/p$-checkers," *Automat. Remote Contr.*, vol. 39, part 2, pp. 1380–1383, Sept. 1978; transl. from *Avtomat., Telemekh.*, pp. 142–145, Sept. 1978.

[8] C. Mead and L. Conway, *Introduction to VLSI Systems*. Reading, MA: Addison-Wesley, 1980.

[9] V. Rabara, "Design of self-checking checker for 1-out-of-n code ($n > 3$)," in *Proc. 4th Int. Conf. on Fault-Tolerant Syst. Diagnostics*, Brno, Czechoslovakia, Sept. 28–30, 1981, pp. 234–240.

[10] S. M. Reddy, "A note on self-checking checkers," *IEEE Trans. Comput.*, vol. C-23, pp. 1100–1102, Oct. 1974.

[11] J. F. Wakerly, *Error Detecting Codes, Self-Checking Circuits and Applications*. New York: Elsevier, 1978.

[12] S. L. Wang and A. Avizienis, "The design of totally self checking circuits using programmable logic arrays," in *Proc. 9th Annu. Symp. on Fault-Tolerant Comput.*, Madison, WI, June 20–22, 1979, pp. 173–180.

Watchdog Processors and Structural Integrity Checking

DAVID JUN LU

***Abstract*—The use of watchdog processors in the implementation of Structural Integrity Checking (SIC) is described. A model for ideal SIC is given in terms of formal languages and automata. Techniques for use in implementing SIC are presented. The modification of a Pascal compiler into an SIC Pascal preprocessor is summarized.**

***Index Terms*—Control flow, error detection, Pascal, structural integrity checking (SIC), structured programming, watchdog processor.**

I. Introduction

The demand for detection of errors in computers during the execution of computer programs has increased. It has become quite difficult to devise tests that effectively exercise VLSI computer systems because of the great complexity of the hardware and software. The increased complexity has introduced new varieties of errors and increased the number of possible errors. Meanwhile, the introduction of many new computer applications has increased dependence on computers.

Techniques using circuit redundancy and coding theory, such as parity checking of memory, can be used within VLSI chips to provide detection of low-level errors. The increased circuit density, however, reduces access to the internal states of each module. At the system level, there remains the problem of determining ways in which VLSI components can be employed to detect errors, especially errors in the overall behavior of a computer system.

The use of auxiliary processors or processes for error detection is an extension of the well known technique of watchdog timers [18], [16]. Major issues in the design of watchdog processors are the selection of operations to be checked for errors, the means of encoding and transmitting information from the main processor to the watchdog, and the programming of the watchdog. Structural integrity checking (SIC) consists of analyzing high-level control flow structures in computer programs, attaching labels to these structures for error detection, and checking the integrity of these structures at run time by using a watchdog processor or process. SIC uses syntax driven methods to encode program structures for error detection, and to generate the program for the watchdog.

SIC detects errors in the high-level behavior of a computer system

Manuscript received September 28, 1981; revised January 12, 1982. This work was supported in part by the National Science Foundation under Grant MCS-7904864, DARPA Contract MDA903-79-C-0680, and the U.S. Army Electronics Research and Development Command under Contract DAAK20-80-C-0266. Earlier versions of this work were presented at the National Electronics Conference, Chicago, IL, October 1980 and the IEEE Distributed Data Acquisition, Computing, and Control Symposium, Miami Beach, FL, December 1980. Earlier versions also were published in "Concurrent testing and checking in computer systems" (Ph.D. dissertation), Stanford University, June 1981; and Center for Reliable Computing, Tech. Rep. 81-5, Stanford University, July 1981.

The author is with the Center for Reliable Computing, Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, Stanford, CA 94305.

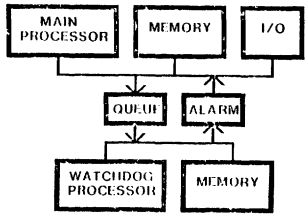

Fig. 1. Typical hardware configuration.

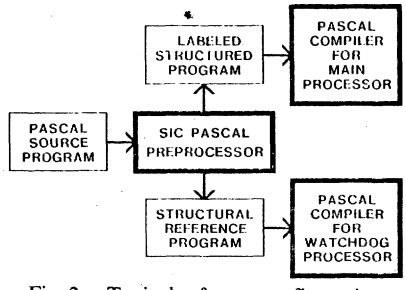

Fig. 2. Typical software configuration.

by checking the structure of control flow during program execution. Correct control flow is a fundamental part of correct execution of computer programs. Anything that prevents the proper updating of the program counter could result in control flow errors. Faults in hardware such as the program counter, address circuits, or microcode memory could result in control flow errors. Faults in software such as bugs in compilers, unintentional modification of program code, or operating system problems also could cause control flow errors. Many error detection techniques applied at lower levels of hardware and software make strong assumptions about the nature of faults. SIC provides additional error detection by trying to detect errors caused by faults, such as multiple faults in a circuit, that escape detection at lower levels.

There are many previous approaches to execution checking. A software structure called the recovery block [8] was defined to provide an orderly means for testing the results of a program module and using alternate software in case of errors. In other studies [10], [2] the sequence of execution is recorded and compared with sequences determined during programming or testing. One scheme [2] tags points in the sequence with distinct prime numbers and compresses them into a check symbol using arithmetic operations. Another study [15] emphasizes capability-based [6] architectural design for error detection.

SIC differs from previous forms of concurrent control flow checking [1], [19] in several ways. Instead of analyzing the flow of control by means of Petri nets or graph techniques, SIC makes use of the control structure information embedded in the syntax of languages suited for structured programming [4], [3], such as Pascal [9]. Structured programming is a discipline used in software design and coding. SIC takes advantage of the fact that control flow in a structured program is determined primarily by the four program constructs known as concatenation, selection, repetition, and abstraction. Examples of such constructs are BEGIN ··· END blocks (concatenation), IF ··· THEN ··· ELSE statements (selection), REPEAT ··· UNTIL statements (repetition), and procedure calls (abstraction). SIC does not require users to provide formal abstract descriptions of their computer programs. SIC programs for the main processor and the watchdog are derived directly from the original program for the main processor. SIC also differs in the use of short labels (e.g., 8 bits) to signal the steps in the sequence of execution.

A typical hardware configuration for SIC is shown in Fig. 1. This configuration is a typical link between two processors in a distributed computing system. A corresponding typical software configuration is shown in Fig. 2, using the language Pascal as an example. The SIC preprocessor is a program that reads in the Pascal source program and analyzes the control flow structure. The preprocessor emits two

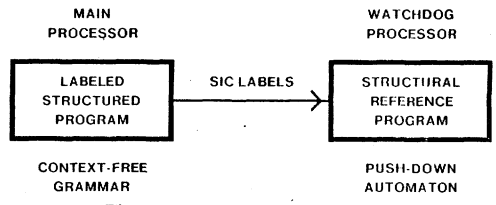

Fig. 3. Automata model of SIC.

Pascal programs. The labeled structured program, for the main processor, is the source program with new statements that transmit the label of a construct to the watchdog when the construct is executed by the main processor. The transmission of labels could be performed concurrently with computations by dedicated hardware in the main processor. The second output of the preprocessor is the structural reference program for the watchdog processor. The structural reference program contains statements to receive and check the labels from the main processor.

II. Formal Description

The operations for SIC can be modeled as the activities of two automata (Fig. 3). One automaton is the execution of the labeled structured program in the main processor of the system. The second automaton is the execution of SIC operations in the watchdog processor. In this model of ideal SIC, a unique label is assigned to each executable element of the structured program in the main processor. As the main processor executes the labeled structured program, it transmits each label it encounters as an SIC message. The watchdog receives these labels and compares them with labels generated within the watchdog by a structural reference program. If there is a mismatch, then the watchdog signals the detection of an error to the system component that manages error detection and recovery. Depending on the system, that component would be a main processor, a control processor, or an operator. Otherwise, if each label matches, then the watchdog signals completion of checking. SIC is defined for checking a single sequential process. Extensions to include context switching such as interrupts is a subject for future study.

The reference program is similar in structure to the program being checked. However, it does not contain code to execute the actions of the checked program. Hence, the computational requirements for the watchdog are less than those for the main processor. For example, a 16-bit processor with 64 Kbytes of memory could serve as a watchdog for a 32-bit main processor with 1 Mbyte of memory. Since the reference program does not execute the actions of the checked program, the watchdog processor must compare the alternative labels of conditional branches, such as those in selection constructs (IF or CASE statements) or those in repetition constructs (FOR, WHILE, or REPEAT statements), with the actual label from the main processor. These comparisons control the execution of the reference program. The comparison and other basic functions of the watchdog processor can be checked by techniques such as self-checking circuits, hardware redundancy, and watchdog timers.

Ideal SIC does not include execution semantics closely related to program control such as checking the branch taken in a selection or the number of cycles in a repetition. For example, in an IF ··· THEN ··· ELSE statement, SIC checks that the control flow follows either the THEN branch or the ELSE branch. SIC does not, however, check that the branch matches the IF condition because that depends on execution semantics. As another example, consider checking the number of cycles in a FOR statement, or an upper bound on the number of cycles in a WHILE statement. This checking is not part of ideal SIC although it might be added as an extension. On the other hand, another kind of counting is part of ideal SIC. For abstraction constructs (procedure or function calls), SIC checks that each call is matched by an exit. A count is implicitly maintained by the stack of the structural reference program. In general, SIC checks the structure of control flow, but does not check the details of control flow related to execution semantics such as the values of variables.

Ideal SIC can be defined rigorously in terms of formal languages and automata [7]. The strings of labels produced by the executions of a labeled structured program can be described by a context-free grammar (CFG) derived from the structure of the program. The watchdog processor accepts exactly those strings described by the CFG and rejects any strings corresponding to structural errors in the control flow, or any other strings, for that matter. For a given structural reference program, the watchdog recognizes the strings produced by a specific structured program, rather than just checking that the strings are produced by some structured program. In some cases, it would be possible to find an equivalent regular grammar (RG). Then it would be possible, in theory, to reduce the function of the watchdog to a finite automaton (FA). For example, if there are no recursive procedure or function calls in the structured program, then it is possible to describe the strings with a grammar that is not self-embedding. The structural reference program, however, would have more states, would bear less resemblance to the structured program, and would be more difficult to derive from the structured program.

The executions of a labeled structured program produce strings of labels described by a CFG $G = (N, T, P, S)$. N is the set of variables, which are nonterminals corresponding to the nonatomic constructs in the structured program. T is the set of terminals, which are uniquely assigned labels for each element of the structured program. An element of a structured program is defined as a construct that is atomic in the sense that it is one of the indivisible modules used to build the structured program. S, the start symbol, is a distinguished member of N that derives $L(G)$, the language of G. $L(G)$ is exactly the set of label strings that can be produced by executions of the corresponding structured program (note that $L(G)$ is not the computer language used to write the program). P is the set of productions, which are derived from the structured program.

Each construct in the structured program results in one or more members of P. If there is a concatenation construct A of constructs $B1, B2, \cdots, Bm$ in the program, then there is a production $A' \rightarrow B1'B2' \cdots Bm'$ in P, where Bi' is the label of Bi if Bi is an element, otherwise Bi' is a nonterminal if Bi is a nonatomic construct. If there is a selection construct A of constructs $B1, B2, \cdots, Bm$ in the program, then there is a set of productions $(A' \rightarrow B1', A' \rightarrow B2', \cdots, A' \rightarrow Bm')$ in P, where Bi' is the same as above. Similarly, if there is a REPEAT $\cdots$ UNTIL type of repetition construct A of a construct B, then there is a pair of productions $(A' \rightarrow B', A' \rightarrow B'A')$ in P. If there is a WHILE $\cdots$ DO or FOR $\cdots$ DO repetition construct A of a construct B, then there is a pair of productions $(A' \rightarrow \langle\text{empty}\rangle, A' \rightarrow B'A')$ where $\langle\text{empty}\rangle$ is the empty string. CFG's can be extended [7] to include productions of the form $A' \rightarrow \langle\text{empty}\rangle$.

If there is an abstraction construct (procedure or function call) A of a construct (procedure or function) B in the program, then there is a production $A' \rightarrow B'$ in G. Note that if construct B is redefined within construct C, then the calls to B within C are understood to be calls to a new construct D, which derives a string of terminals distinct from that of B. Since the left side of each production in the grammar G is a single variable, G is a CFG (extended with $\langle\text{empty}\rangle$ rules).

Many programming languages permit the use of the abstraction construct known as function call. Depending on the language, these function calls may be permitted to appear in various places not considered above. For example, function calls may appear in expressions for array subscripts, or in control expressions of selection or repetition constructs. While this makes the formal description more complicated, the structure of the description is similar to the model presented above. In particular, the left side of each production in the grammar G is still a single variable, so G is still a CFG.

The automaton necessary and sufficient for accepting the strings described by a CFG is a push-down automaton (PDA), a finite automaton with a push-down store (stack) accessible only through the top. The method for deriving the structural reference program so that the watchdog behaves as such a PDA is straightforward. If there is a concatenation construct A of constructs Bi in the structured program, then the reference program contains a concatenation of operations that check the terminal strings of each construct Bi. If there is a selection construct A of constructs Bi in the structured program, then the reference program contains a selection of operations that check the terminal strings of each construct Bi. The selection in the reference program is controlled by testing the first terminal for membership in the set of first terminals that each Bi' can derive. If there is a REPEAT $\cdots$ UNTIL repetition construct A of a construct B in the structured program, then the reference program contains a REPEAT $\cdots$ UNTIL repetition of operations that check the terminal strings of B. The UNTIL condition of the repetition becomes true when the next terminal does not belong to the set of first terminals that B' can derive. If there is a WHILE $\cdots$ DO or FOR $\cdots$ DO repetition construct A of a construct B in the structured program, then the reference program contains a WHILE $\cdots$ DO repetition of operations that check the terminal strings of B. The WHILE condition of the repetition becomes false when the next terminal does not belong to the set of first terminals that B' can derive. If there is an abstraction construct A of a construct B in the structured program, then the reference program contains an abstraction of (call to) a procedure to check the terminals of B. In the conditional branches of selections or repetitions, the sets of first terminals that the alternative branches Bi' can derive are mutually exclusive. The assignment of terminals in this manner assures unambiguous, and therefore deterministic, acceptance of strings that do not have structural errors.

The construct known as GOTO is discouraged in structured programming, but most programming languages still allow some forms of it. Since the structure of the watchdog program mimics the structure of the main program, SIC can be implemented for GOTO's by placing GOTO's and labels (distinct from SIC labels) in the watchdog program structure, in correspondence with GOTO's and labels in the structure of the main program. A model of ideal SIC including GOTO's can be created by introducing continuations, which correspond to "the remainder of the computation to be performed by the program" [5]. The grammar is no longer strictly context-free because its interpretation must be modified [14].

The model of ideal SIC precisely defines what is, and what is not, included in SIC. It provides a standard for complete SIC, against which implementations can be compared. This model provides a framework for using syntax driven methods to label the main program and to generate the watchdog program. This is an important advance over earlier forms of concurrent control flow checking because it eliminates the need for intermediate representations such as directed graphs or Petri nets, and because it is compatible with recursive procedure and function calls.

III. Implementation

The size of the structural reference program, and hence the size of the watchdog's memory, can be reduced by limiting the structural resolution of the label assignments. This can be done either globally, or locally by the programmer. For example, the labeling of the structured program could be limited to 3 levels from the top, so that any construct beyond level 3 would be treated as a structural element. On the other hand, special control comments could allow the programmer to explicitly force the treatment of a nonatomic construct as an element. This would permit the programmer to specify the amount of checking required for various sections of the program. Limiting the structural resolution limits the amount of structural detail, and thus reduces the size of the structural reference program.

Limiting the size of the reference program does not necessarily limit the rate at which SIC labels are generated by the main processor during execution. To reduce delays in error detection, the watchdog should process the SIC messages as fast as the main processor can generate them. If a queueing buffer is placed between the main processor and watchdog, then the peak rate requirements on the watchdog can be relaxed so that, for most cases, the watchdog does not cause the main processor to wait. Another timing problem may arise if the main processor can generate SIC messages faster than it

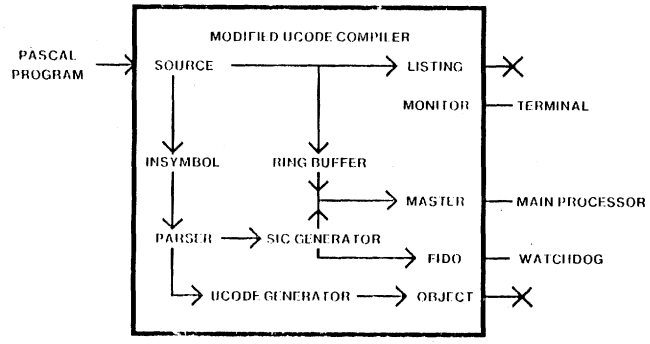

Fig. 4. Pascal preprocessor.

TABLE I
EXAMPLES OF SIC

SOURCE PROGRAM	MAIN PROCESSOR	WATCHDOG PROCESSOR

PROGRAM A;
<LABEL DECLARATIONS>;
	PROGRAM A(SICS);
	<LABEL DECLARATIONS>;
	VAR SICS:TEXT;
	PROCEDURE
	SEND(X:INTEGER);
	BEGIN
	WRITELN(SICS,X)
	
	END;
		PROGRAM A(SICS);
		<LABEL DECLARATIONS>;
		VAR SICS:TEXT;
		FUNCTION CPU:INTEGER;
		VAR X:INTEGER;
		BEGIN
		READLN(SICS,X);
		CPU:=X
		END;
		PROCEDURE ALARM;
		BEGIN
		WRITELN(TTY,
		'SIC ALARM!');
		HALT
		END;

<DECLARATIONS>; BEGIN B END.	<DECLARATIONS>; BEGIN REWRITE(SICS); B END.	<CHECK DECLARATIONS>; BEGIN RESET(SICS); <CHECK B> END.
BEGIN A; B END	BEGIN SEND(X); A; SEND(Y); B END	BEGIN IF CPU<>X THEN ALARM; <CHECK A>; IF CPU<>Y THEN ALARM; <CHECK B> END

IF A THEN
 	
B
ELSE
 	
C
	IF A THEN
	BEGIN
	SEND(-X);
	B
	END
	ELSE
	BEGIN
	SEND(-Y);
	C
	END
		
		IF CPU=-X THEN
		<CHECK B>
		ELSE
		
		<CHECK C>

CASE A OF B: C; D: E END	CASE A OF B: BEGIN SEND(-X); C END; D: BEGIN SEND(-Y); E END END	CASE CPU OF -X: <CHECK C>; -Y: <CHECK E> OTHERWISE ALARM END

WHILE A DO
 	
B
	BEGIN
	WHILE A DO
	BEGIN
	SEND(-X);
	B
	END;
	SEND(-Y)
	END
		
		WHILE CPU=-X DO
		<CHECK B>

REPEAT
 	
A;
 	
B
UNTIL C
	BEGIN
	REPEAT
	SEND(-X);
	A;
	SEND(Y);
	B
	UNTIL C;
	SEND(-Z)
	END
		BEGIN
		IF CPU<>-X THEN ALARM;
		REPEAT
		<CHECK A>;
		IF CPU<>Y THEN ALARM;
		<CHECK B>
		UNTIL CPU<>-X
		
		END

TABLE I (CONTINUED)

SOURCE PROGRAM	MAIN PROCESSOR	WATCHDOG PROCESSOR

FOR A:=B TO C DO
 	
D
	BEGIN
	FOR A:=B TO C DO
	BEGIN
	SEND(-X);
	D
	END;
	SEND(-Y)
	END
		
		WHILE CPU=-X DO
		<CHECK D>

PROCEDURE A;
<DECLARATIONS>;
BEGIN
B
END;
	PROCEDURE A;
	<DECLARATIONS>;
	BEGIN
	B
	END;
		PROCEDURE A;
		<CHECK DECLS>;
		<CHECK BEGIN
		B
		END>;

BEGIN A END	BEGIN A END	BEGIN A END

FUNCTION A:TYPE;
<DECLARATIONS>;
BEGIN
B
END;
	FUNCTION A:TYPE;
	<DECLARATIONS>;
	BEGIN
	B
	END;
		PROCEDURE A;
		<CHECK DECLS>;
		<CHECK BEGIN
		B
		END>;

BEGIN C:=A END	BEGIN C:=A END	BEGIN A END
LABEL 1; 1: A; GOTO 1	LABEL 1; 1: SEND(X); A; GOTO 1	LABEL 1; 1: IF CPU<>X THEN ALARM; <CHECK A>; GOTO 1

can transmit them (e.g., if the main processor chip includes a cache memory). This problem is alleviated by serially buffering the messages within the main processor chip and sending them in parallel format to the watchdog. The use of short labels, described below, makes this technique attractive. Both of these techniques would introduce delays in the detection of structural errors.

SIC labels can be implemented in many ways. In general, the labels are attached to the compiled code of the structured program either between the machine instruction words or directly to the instruction word format. Placing labels between instructions saves hardware but uses main processor time. The second method, attaching labels to instructions, is more attractive in light of present trends toward lower cost hardware and greater demand for processor speed. The transmission of labels can be performed in parallel with the execution of the associated instructions.

One possible implementation of SIC labels uses the address of the instruction as the label. This would be very close to ideal SIC since each instruction has a unique address. The disadvantage is the size of the label. The watchdog would have correspondingly large constants in the reference program. The use of short subfields of the address, e.g., the lowest 8 bits, instead of the entire address might be practical. The following method, however, is less difficult to implement and analyze.

SIC labels can be implemented by appending a relatively short label to the instruction word. For example, a short label could be 8 bits wide. Since a label of width w results in $2^{**}w$ distinct labels, some of the labels in a program may not be unique. One goal is to uniformly distribute the label values in order to minimize the chance of a random transfer to an instruction with the same label as the correct instruction. Another goal is to assign label values so that sequences of labels are likely to be distinct, thus maximizing structural error detection over sequences. These two goals can be approached easily by randomly and independently selecting each label from a uniform distribution. The first goal is achieved by this technique. The second goal is nearly attained since the probability of an erroneous length n sequence of labels going undetected is $2^{**}(-n*w)$, where w is the width

of the labels. The labels associated with different branches must be selected so that they are distinct. For CASE statements, the number of possible cases would be constrained to 2**w.

SIC is defined in terms of the high-level behavior of system components. The theoretical model of SIC and implementation techniques, such as labels taken from a uniform distribution, make it possible to evaluate SIC in these terms. The evaluation of SIC in lower level terms, such as sensitivity to circuit faults, is a subject for future research. Experiments and simulations can provide data for estimating the circuit fault coverage of SIC.

An experimental SIC Pascal preprocessor is being implemented by modifying an existing compiler, the UCODE Pascal compiler [17]. Choosing Pascal for both the input and output of the SIC software provides software portability. The Pascal preprocessor reads the source program and emits a modified program for the main processor, and a checking program for the watchdog processor. The modifications to the source program for the main processor implement the transmission of SIC labels during execution. The program for the watchdog processor implements the reception and checking of the labels.

In the experimental SIC Pascal preprocessor, the input source code is intercepted symbol-by-symbol (not character-by-character) and copied to the output for the main processor program. During the intercepting and copying of symbols, when the parsing indicates that a structured programming construct has been recognized, Pascal statements are injected into the main program and Pascal statements are generated for the watchdog program. The overall structure of the Pascal preprocessor is shown in Fig. 4. The modifications for the concatenation, selection, and repetition constructs have been made on the compiler. The resulting SIC Pascal preprocessor has been used to demonstrate SIC [14]. The modifications for the abstraction constructs known as procedure and function calls, and for user control of structural depth and resolution, are being planned. Table I shows examples of structured programming constructs and the corresponding statements in the main and watchdog programs. The examples do not cover all possible cases. For example, only the simplest form of function call is shown. The control information concerning conditional branches is transmitted from the main processor to the watchdog as negative label values, such as "$-x$." The notation, "⟨check declarations⟩" means that SIC is to be applied to the bodies of abstractions (procedures and functions) in the declarations. These examples are not strict specifications for SIC. It is possible to implement either more or less checking.

IV. Conclusion

Structural integrity checking (SIC) is a new method for using watchdog processors to monitor program execution. Unlike previous approaches, SIC uses high-level structural information embedded in the syntax of the program text. A theoretical model is given to define SIC in terms of formal languages and automata, and techniques for implementing SIC are presented. A Pascal compiler is being modified to implement SIC. Future goals include expanding the set of constructs checked, giving the user control over the depth and resolution of checking, extension to checking of execution semantics related to program control, evaluation of the sensitivity of this technique to lower level faults, and design of language features for SIC.

Watchdog processors and SIC were introduced in [11]. A summary of SIC and label compression was given in [12]. The experimental SIC Pascal preprocessor and a model for GOTO's in SIC were introduced in [13] and [14].

Acknowledgment

Advice given by Prof. E. J. McCluskey is gratefully acknowledged. Prof. S. S. Owicki and Prof. J. F. Wakerly also provided comments and suggestions. Prof. J. L. Hennessy provided access to the UCODE Pascal compiler. Discussions with W. Cory, D. Davies, D. Hill, and S. Wakefield were very helpful.

References

[1] J. M. Ayache, P. Azema, and M. Diaz, "Observer: A concept for on-line detection of control errors in concurrent systems," in *Proc. 9th Annu. Int. Conf. on Fault-Tolerant Comput.*, June 1979, pp. 79–86.

[2] S. Bologna and W. Ehrenberger, "Possibilities and boundaries for the use of control sequence checking," in *Proc. 8th Annu. Int. Conf. on Fault-Tolerant Comput.*, June 1978, p. 226.

[3] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming.* New York: Academic, 1972.

[4] E. W. Dijkstra, "Structured programming," in *Software Engineering Techniques*, J. N. Buxton and B. Randell, Eds. Rome, Italy: NATO, Oct. 1969, pp. 84–88.

[5] J. E. Donahue, *Complementary Definitions of Programming Language Semantics* (*Lecture Notes in Computer Science*, vol. 42). New York: Springer-Verlag, 1976, p. 47.

[6] R. S. Fabry, "Capability-based addressing," *Commun. Ass. Comput. Mach.*, vol. 17, pp. 403–412, July 1974.

[7] J. E. Hopcroft and J. D. Ullman, *Formal Languages and Their Relation to Automata.* Reading, MA: Addison-Wesley, 1969.

[8] J. J. Horning, H. C. Lauer, P. M. Melliar-Smith, and B. Randell, "A program structure for error detection and recovery," in *Lecture Notes in Computer Science*, vol. 16. New York: Springer-Verlag, 1974, pp. 171–187.

[9] K. Jensen and N. Wirth, *Pascal User Manual and Report, 2nd ed.* New York: Springer-Verlag, 1976.

[10] J. R. Kane and S. S. Yau, "Concurrent software fault detection," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 87–99, Mar. 1975.

[11] D. J. Lu, "Watchdog processors and VLSI," in *Proc. Nat. Electron. Conf.*, vol. 34, Chicago, IL, Oct. 1980, pp. 240–245.

[12] D. J. Lu, E. J. McCluskey, and M. Namjoo, "Summary of structural integrity checking," in *Proc. IEEE Distributed Data Acquisition, Comput., Contr. Symp.*, Miami Beach, FL, Dec. 1980, pp. 107–109.

[13] D. J. Lu, "Concurrent testing and checking in computer systems," Ph.D. dissertation, Stanford Univ., Stanford, CA, June 1981.

[14] ——, "Watchdog processors and structural integrity checking," CRC Tech. Rep. 81-5 (CSL TR 212), Cen. for Reliable Comput., Comput. Syst. Lab., Dep. Elec. Eng. Comput. Sci., Stanford Univ., Stanford, CA, July 1981.

[15] G. J. Myers, "Storage concepts in a software-reliability-directed computer architecture," in *Proc. 5th Annu. ACM–IEEE Symp. on Comput. Arch.*, Apr. 1978, pp. 107–113.

[16] J. S. Nowak, and L. S. Tuomenoksa, "Memory mutilation in stored program controlled telephone systems," in *Proc. IEEE 1970 Int. Conf. on Commun.*, vol. 2, June 1970, pp. 43-22–43-45.

[17] A. R. Rodriguez, "Ucode Pascal," S-1 Project, Dep. Comput. Sci., Stanford Univ., Stanford, CA, Dec. 1979.

[18] *SDS Sigma 7 Computer Reference Manual.* Santa Monica, CA: Scientific Data Systems, May 1966, p. 24.

[19] S. S. Yau and F.-C. Chen, "An approach to concurrent control flow checking," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 126–137, Mar. 1980.

Self-Stabilizing Programs: The Fault-Tolerant Capability of Self-Checking Programs

ALI MILI

***Abstract*—Self-checking programs are programs which meet the following condition. For any legal input, either they return the correct output or they return a message indicating that the output may be incorrect. Self-checking programs are capable of recognizing irregular conditions in their state space and reporting it. Self-stabilizing programs are programs which, in addition to**

Manuscript received September 28, 1981; revised January 7, 1982.

The author is with the Department of Computer and Information Sciences, Texas A & M University, College Station, TX 77843.

0018-9340/82/0700-0685$00.75