

Filed: December 9, 1994

IP05

Copy

9607119-6

PCT/US 94/05182
IPEA/US 09 DEC 1994

-1-

METHOD AND APPARATUS FOR THE AUTOMATIC
ANALYSIS OF COMPUTER SOFTWARE

REPLACED SHEETS		SG
☒	Description : paged 1 to paged 28	
☒	Claim(s) : paged 29 to paged 9	
☒	Drawing(s) : paged 1 to paged 25	
☒	Abstract : paged 48 to paged 48	

BACKGROUND OF THE DISCLOSURE

1. Field of the Invention

The invention relates generally to the field of fault identification and analysis in computer software. More particularly, the invention relates to a method and apparatus used for the automatic analysis of computer software.

2. Background of the Invention

Since the day the first computer program was written and to this day, errors or faults in computer software are an expected and commonplace occurrence. Even though such faults are expected, fault identification and analysis is a problem which has forever plagued programmers. This due in part, to the fact that the fault rate for a carefully constructed program can be about 5 faults per 1,000 lines of source code so that a one million-line program may contain 5,000 faults! Compound this with the fact that computer software faults are different from computer hardware faults in that the software faults do not obey the same simplified rules of proximity of cause and effect. That is to say, a computer software fault does not necessarily have to appear close to its failure, but instead can be located almost anywhere within a much larger software system. Therefore, even when a fault is known to exist, determining the origin and location of that fault is extremely difficult at best.

This Sheet Has Been Replaced



AMENDED SHEET

THIS SHEET HAS BEEN
REPLACED

FCT/US 94/05182
IPEA/US 09 DEC 1994

-2-

3. Description of the Prior Art

As a result of the above mentioned characteristics of computer software, the prior art has developed a number of methods and apparatus to assist a programmer in the development and analysis of software. The methods and apparatus of the prior art encompass a wide array of systems and techniques, each generally applicable to a different period of the overall computer software development process. Among these prior art methods and apparatus are: 1) structured design techniques; 2) language-sensitive development tools; 3) highly-efficient and informative language interpreters and compilers; and 4) an assortment of program profilers and debuggers and emulators which assist in identifying and locating faults in software during the debugging and testing of the software. Despite all these tools, techniques and related efforts, a large amount of programmer time is spent in trying to understand how a program works and many failures are tolerated out of fear of further "breaking" a software system.

There are many problems associated with existing tools and techniques for identifying and locating faults in computer software. Traditionally, programmer had to debug or analyze a program through the use of some kind of "hook" or output statement which was part of the programming language used for the software program. For example, in one particular language a "PRINT" statement would be inserted by a programmer into one or more locations of a program being analyzed so that upon encountering the PRINT statement, the program being analyzed would output to an appropriate output device the contents of some

This Sheet Has Been Replaced



AMENDED SHEET

94/05182
IPEA/US 09 DEC 1994

-3-

registers, variables, etc. In this manner, the programmer would examine the output of the executed PRINT statement searching for clues as to why the program failed. Needless to say, this method of program analysis and debugging is very tedious for the programmer and error prone as well.

Another method of program debugging and analysis involves the use of a special program, known as a program debugger which monitors the execution of a software program and oversees its execution. Such debuggers typically offer enhanced flexibility and more information to the programmer than possible with the simple PRINT method, particularly with respect to a user interface presented to the programmer. Utilizing a debugger, a programmer will normally specify one or more lines of the software program as "breakpoints", or points within a software program which when encountered suspend the execution of the program so that the programmer may examine processor registers or program variables. Breakpoints are not limited to specific lines of code, breakpoints may be used to interrupt execution when an expression changes value, or when a expression reaches a value. Despite apparent advantages over the PRINT method, program debugging and analysis using known tools suffer from many of the same infirmities i.e., they do not automate the understanding of cause-effect relationship between faults and failures (manifestation of fault).

The first problem associated with these above mentioned debugging and analysis methods and tools is where to begin. This is largely a hit or miss proposition for the programmer. Therefore, the programmer will typically insert a number of

This Sheet Has Been Replaced


AMENDED SHEET

PCT/US 94/05182
IPEA/US 09 DEC 1994

-4-

breakpoints or PRINT statements which are scattered throughout the program under test. As the programmer narrows down his understanding of the nature and location of program fault(s) responsible for a failure, unnecessary breakpoints or PRINT statements are normally removed or deactivated. This technique of randomly inserting breakpoints or PRINT statements throughout a program is highly inefficient, time-consuming, and error-prone.

Second, even if the programmer has some idea where within a program to place a breakpoint, the programmer must determine what to examine at that location. That is, a particular section of a program may use a large number of variables, registers or memory locations, thereby making the programmer's job of identifying the fault(s) more difficult. While debuggers and analyzers have certainly facilitated the examination of variables, registers or memory locations, they have by no means eliminated the problem of making a decision as to what variables, memory, or registers to examine and when to examine them. Furthermore, debuggers do not help to decide what is the meaning of the value of a variable and what other variables of what other statements need to be examined until the fault statement is found. As can be appreciated, the traditional techniques and tools used for debugging or analyzing a software program is not highly automated.

In an attempt to overcome these stated deficiencies in the art of program debugging and analysis, the prior art has developed additional tools and techniques.

AMENDED SHEET

This Sheet Has Been Replaced



-5-

United States Patent No. 4,853,851, "System
For Determining The Code Coverage Of A Tested Program
Based Upon Static And Dynamic Analysis Recordings"
issued to Borsch on August 1, 1989 teaches a system for
5 analyzing programs by measuring the degree of code
coverage of a program being tested during specific test
phases. The system disclosed utilizes a static and
dynamic instruction flow indicator which permits a
determination of the test cover results by correlating
10 the data of the static and dynamic instruction flow.
In this manner, the number of untested statements can
be determined during a test phase of a computer
program.

15 Although the more statements covered during
the test of a program the better, the fact that a
statement was executed does not provide any information
on the statement validity. In particular, the
statement may have been constructed wrong; the
20 statement may be executed at the wrong place at the
wrong time; or the statement may not contribute to the
observed result.

United States Patent No. 4,802,116,
25 "Programmed Controller" issued to Ward, et al on
January 31, 1989 teaches programmed controller which
controls a machine process and facilitates its
debugging. The application programs disclosed are
specifically designed to simulate specific control
30 schemes for sections of the machine by being organized
into one or more separate loops. A built-in control
mechanism records the states of these loops along with
the conditions responsible for the transitions between
those states. This teaching is limited in application
35 to only those processes that are specially coded by

AMENDED SHEET

This Sheet Has Been Replaced



-6-

very specific syntax rules as consisting of independent loops, where the means of tracing the states of these loops are pre-coded in the programs controlling the process.

5

A theoretical approach to the general problem of program analysis was disclosed by Thomas J. macabre in the article "A Complexity Measure" published in IEEE Transactions on Software Engineering, Vol. Se-2, No. 4, December 1976, pp. 308-320. Cyclomatic Complexity was proposed in this article as a measure of program complexity. This measurement is nothing more than the number of independent passes within a subroutine. This complexity measurement is useful to promote the design of less sophisticated software routines which are also easier to test. However, counting the number of independent passes does not eliminate their necessity

15

Despite these and other attempts made by the prior art, software debugging and analysis remains a difficult, time-consuming and error prone process. A critical shortcoming in the prior art is that there are no teachings for general methods or apparatus which are capable of automatically determining the location of logic faults in software programs even within a broad area of a program. This is due, in large part, to a general belief which exists in the art that such automation is not possible because software does not conform to the rules of proximate cause and effect and that a software program has no knowledge of what results are expected of it unless those expected results are pre-coded within the program. Such pre-coding is itself programming and therefore is also error-prone.

20

25

30

This Sheet Has Been Replaced



Therefore, a continuing need exists in the art for a method and apparatus which more highly automates the chore of software program debugging and analysis.

SUMMARY OF THE INVENTION

It is therefore the object of the present invention to provide a method and apparatus which automates the process of program debugging analysis and maintenance by allowing for automatic identification of an area of a program, which area contains a fault(s) responsible for specific failure or contains the statements to be modified in order to comply with the new requirements to the program. The mentioned area is automatically and continuously minimized while the invented system automatically learns about the target software program.

The invention presents, what will be referred as "Knowledge Induction Machine", that is an apparatus and method that build and propagate the knowledge about the internal processes within the target software program.

The invention thus presents the advantage that the method and device for debugging and analyzing software programs does so with precision that grows with the accumulation of knowledge about the target or analyzed program behavior. The invention successively accumulates this knowledge, and utilizes it to determine correct or uncertain behavior of the analyzed software program. The invention further shows the ability to partially "unlearn" previously acquired knowledge, thereby recognizing that correct results are



PCT/US 94/05182

IPEA/US 09 DEC 1994

-8-

a function of a user's expectations of the results which may change over time.

5 The invention operates by analyzing the results or effects produced by a software process and automatically traces back from these effects to their cause(s). In this manner failures, which are unexpected results, are automatically traced back to the fault(s) responsible. This is in contrast to
10 traditional methods, i.e., stepping forward through the software process to precipitate a failure. The analysis is accomplished by the invention of the present application by first building a model of the software process under analysis on the basis of a
15 static analysis performed under a set of strict rules. The model is constructed such that it serves as a framework for the further accumulation of knowledge which is acquired through a dynamic analysis of the software process.

20 The invention constructs the model of the target software process in such a manner that allows for the automatic determination of accessibility, or presence of a path, between any two elements of the
25 target software process. Accessibility is required for dependency, that is, if one statement within a software process is accessible from another statement within the process, then, and only then, the correctness of the other statement may affect the behavior of the one
30 statement under examination. If the second statement is not accessible from the first statement, then correctness/incorrectness of the first statement cannot affect the behavior of the second statement and therefore the second statement is not dependent upon
35 the first statement.

AMENDED SHEET

This Sheet Has Been Replaced



PCW/US 94/05182

IPEA/US 09 DEC 1994

-9-

During the construction of the model of the target software process, the invention extracts the control component of the target software process into a form which represents a control structure of that target process. Furthermore, the invention accumulates information about the behavior of the target software process thereby permitting the invention to determine the correctness of a software process element relative to the time at which the element is executed. Still further, this accumulated correctness information is propagated between target process elements (statements) thus automatically increasing the intelligence of the target process model. Through this accumulated knowledge that the invention acquires about the correctness of the software process behavior, the invention is able to automatically determine the location of fault(s) responsible for a failure.

These resources maintained by the inventive software analyzer, or "Knowledge Induction Machine", are presented to the user in an easy-to-use implementation. The preferred embodiment makes extensive use of an intuitive user interface as well as multimedia capabilities known in the art to further facilitate software debugging and analysis.

It is still another object of the present invention to introduce an original form of the target process object code, eX-Object code, and a method for its execution. The eX-Object code is more suitable for program analysis, both static and dynamic and for program maintenance. It eases and sometimes eliminates the traditional recompiling and re-linking of the target process code, which are traditionally necessary in order to implement modifications to the target

This Sheet Has Been Replaced



PCTAS 94/05182

IPEA/US 09 DEC 1994

-10-

process. The eX-Object code comprises the model
extracted from the target process. Therefore,
increasing the intelligence of the model performed by
the "Knowledge Induction Machine" thus increases the
intelligence of the resulting eX-Object code.

BRIEF DESCRIPTION OF THE DRAWINGS

The teachings of the present invention can be
readily understood by considering the following
detailed description in conjunction with the
accompanying drawings, in which:

FIG. 1 shows a flow chart of a sample target
process.

FIG. 2 shows an eX-Frame of the sample target
process of FIG. 1.

FIG. 3 shows how complex logic conditions are
reduced into two or more reduced logic conditions.

FIG. 4 shows a reduced flow chart of the
sample target process shown in FIG. 1.

FIG. 5. shows eX-Graph of the sample target
process shown in FIG. 1 and the eX-Binary addresses
assigned to the corresponding elements of the
eX-Graph..

FIG. 6 shows how typical IF-THEN-ELSE program
constructs are represented in eX-Frame and eX-Graph
form(s).

This Sheet Has Been Replaced



PCTN'S 94/05182
IPEA/US 09 DEC 1994

-11-

FIG. 7 shows the steps required to construct the eX-Model, eX-Repository and eX-Source presented next to the main components of the Knowledge Induction Machine (KIM).

FIG. 8 shows the main components of the Knowledge Induction Machine (KIM) operating in the traditional object code implementation.

FIG. 9 shows the construction of exit access indicators and the construction of propagated vision points.

FIG. 10 shows the construction of the Propagated Vision Point of a process terminal.

FIG. 11 shows a target process with faults and their potential origin.

FIG. 12 shows three different types of target process failure(s).

FIG. 13 shows how (KIM) minimizes the area of uncertainty of the position of a fault within a target process, which fault is responsible for a specific failure of the target process.

FIG. 14 shows terminated event definition space.

FIG. 15 shows the propagation of correct algorithm space.

AMENDED SHEET

This Sheet Has Been Replaced



FCI/US 94/05182

IPEA/US 09 DEC 1994

-12-

FIG. 16 shows the accumulation of knowledge about the correctness of a process element located within a specific process address.

5 FIG. 17 shows the knowledge propagation process implemented by constructing event definition spaces built only in specific directions and only up to the knowledge propagation terminals.

10 FIG. 18 shows the main components of KIM operating in the eX-Object code implementation, the two components which comprise eX-Object code and an alternative implementation of KIM through interpretation of the eX-Repository by the eX-Driver
15 controlled by the eX-Model.

FIG. 19 shows the construction of an analyzed calculation base.

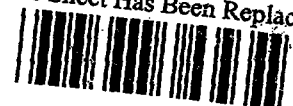
20 FIG. 20 shows an eX-Graph, eX-Repository of a sample process.

FIG. 21 shows a sample process and its eX-Source code.

25 FIG. 22 is a flow chart depicting the steps which comprise the Next Lowest Potential method or rule of parsing a target process.

30 FIG. 23 is a pseudocode listing which describes the steps necessary to create an eX-Frame of the target process by the next lowest potential method.

This Sheet Has Been Replaced



AMENDED SHEET

94/05182

IPEA/US 53 DEC 1994

-13-

FIG. 24 is a system block diagram showing the overall analysis system and the components parts which it comprises.

5 FIG. 25 shows the construction of event definition space terminated by knowledge propagation terminals.

10 FIG. 26 is a flow chart depicting the steps necessary to create an eX-Graph of a target process by the next lowest potential method.

15 FIG. 27 shows a sample format of Incremental System Output Process Synchronized map (ISOPS map).

To facilitate understanding, identical reference numerals have been used, where possible, to designate identical elements that are common to the figures.

20

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

After considering the following description, those skilled in the art will clearly realize that the teachings of this invention can be readily utilized in the analysis of software programs or processes.

30 FIG. 24 is a block diagram of the Knowledge Induction Machine (KIM) analysis system 240 which comprises the present invention. Some of the elements contained within the analysis system are well known in the art as they are typically found in conventional personal computers. Well known components of the analysis system includes, keyboard 257, a CRT display 35 258, CPU 241, memory 242, and some form of mass storage

AMENDED SHEET

This Sheet Has Been Replaced



PCT/US 94/05182
IPEA/US 09 DEC 1994

-14-

facility 272 (e.g., a hard disc drive). These systems communicate via and are frequently capable of receiving a wide array of options via interface bus 256.

5 It must be noted, that whether different components of the system reside on the mass storage or in the memory is often not important and is frequently controlled by the operating system.

10 A target software process 250 is defined by a target source program comprising one or more program statements. The target source is typically stored on mass storage device 272. Target process 250 is extracted from target source by eX-Builder program 259.
15 The eX-Builder 259 examines the loaded target process 250 and constructs an eX-Graph 262 of that target process 250.

20 An eX-Repository 246 of the target process is built by the eX-Builder 259 at the same time the eX-Graph 262 is constructed. The eX-Repository 246 so constructed is "control-less", in other words, it does not contain any process control statements which were present in the target process 250. Instead, process
25 control information is preserved within the eX-Model 243.

30 The eX-Model 243 comprises one or more elements, each element corresponds to an executable statement of the target process 250. Multiple eX-Models 243 are constructed when the target process 250 comprises multiple components, i.e., main, subroutines, functions, etc.

AMENDED SHEET

This Sheet Has Been Replaced


PCT/US 94/05182
IPEA/US 09 DEC 1994

-15-

After the eX-Model 243 and eX-Repository 246 are built, eX-Source Generator 249 constructs eX-Source code 247 from the information contained within eX-Model 243 and eX-Repository 246. The eX-Source 247 is representative of the original target process 250, since it is constructed from eX-Model 243 and eX-Repository 246 which in turn, were constructed from the target process 250.

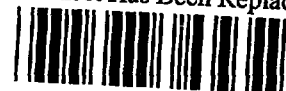
The eX-Source 247 is only created when eX-Analyzer 240 is used in its traditional object code embodiment. The eX-Source 247 permits the eX-Analyzer 240 to operate with the traditional, object code representation of target process 250, which should be very familiar to those skilled in the art. The eX-Source 247 so constructed contains "synchronization signals", i.e., CALL() statements or WRITE() statements which are inserted into eX-Source 247 by eX-Source Generator 249.

The synchronization CALL() statements reference the eX-Interface library 263. The eX-Source code 247 is compiled by compiler/linker 260 and the resulting code is linked with eX-Interface library 263 thereby resolving the CALL() references. The output code produced by the compiler/linker 260 operating on eX-Source code 247 and eX-Interface library is then executed by CPU 241. During the execution of this code, the synchronization statements embedded into eX-Source and now compiled/linked, are executed which, in turn invoke the execution of eX-Analytical Engine 244.

The creation of eX-Source 247 and compiling/linking it with eX-Interface library in order

AMENDED SHEET

This Sheet Has Been Replaced



-16-

to send synchronization signals to the eX-Model is not done in the embodiment utilizing the eX-Object code or in the embodiment utilizing a language interpreter operating on eX-Repository. In the two last mentioned
5 embodiments the model controls the execution of the target process.

In a first embodiment utilizing traditional object code and upon receipt of a later synchronization
10 signal from the eX-Interface 263, eX-Analytical Engine 244 traces the eX-Model 243 from the point at which a previous synchronization signal was received up to the position within the eX-Model 243 where the later
15 synchronization signal was received. In this manner, the eX-Analytical Engine 244 effectively "models" the target process by "executing" eX-Model 243.

In the other two embodiments, i.e. eX-Object Code 243 embodiment or eX-Repository 246 interpretation
20 by the Language Interpreter 268 embodiment, the eX-Model 243 is not "executed" after the target process by the eX-Analytical Engine 244, but instead, the eX-Model is "executing" the target process by controlling the
25 eX-Driver 269, that passes corresponding elements of Executable Component of eX-Object Code 248 for execution or corresponding statements of eX-Repository 246 for interpretation. In these two embodiments the
30 synchronization signals representing the state of the eX-Model 243 are passed to eX-Interface 263 by eX-Driver 269.

In all of the mentioned embodiments the eX-Analytical Engine 244 constructs the Analyzed
35 Calculation Base (ACB) 273 by tracing effects to their cause(s). This process is called in the present

AMENDED SHEET

This Sheet Has Been Replaced


PCTIUS

74/05182
IPEA/US 09 DEC 1994

-17-

invention a Knowledge Induction process. It is done in the direction opposite to the direction of the target process execution and only as far as defined by the Knowledge Propagation Terminals, that will be explained further. In this manner, the knowledge of the correctness or uncertainty of the effect of one statement execution event is propagated backwards to define the knowledge of the correctness/uncertainty of the effect of another statement execution event. The ACB 273 comprises Analyzed Calculation Words (ACW) which describe the knowledge of the correctness/uncertainty of target process elements (statements and parameters within the statements) as a function of time within the process.

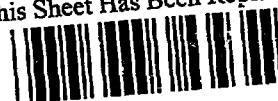
System Execution History, 253, System Input History 252 and System Result History 251 are all produced during the execution of the eX-Model 243. These structures will be described in greater detail later in this description.

The system receives user input through keyboard 257 or other input device and informs the user of analysis results and/or progress via CRT 258.

FIG. 1 is a flow chart representing a sample target source process 250 that is a single module although our results can be easily extended to a process comprising a system of modules. The module could be either a main program or a single subprogram (i.e., subroutine, function, or paragraph) with one entry point. Although the example shown has only a single exit, the invention will work in an identical manner for modules with multiple exits. Those skilled

AMENDED SHEET

This Sheet Has Been Replaced



-18-

in the art could readily extend our teachings to accommodate modules having multiple entry points.

During the initial examination of target source 250 by eX-Builder 259, eX-Model 243 may be built from two predecessor structures, eX-Frame 261 and eX-Graph 262. A representative eX-Frame, so constructed from the target source 250 is shown in FIG. 2.

In the following description of the construction of the eX-Frame, the below given designations and definitions will be used:

Table 1 eX-Object Set		
Designation	Object	Example
Process Entry	[	
End of process description	]	
Arithmetic or Assignment Statement	A	
Function Call other than Graphics Library	X	CALL, PERFORM
Logic Condition	L	IF
Loop Logic Condition	D	DO, WHILE, FOR
User Input Statement	E	READ
System Output Statement	W	WRITE(); CALL MOVETO(x,y,xy)
Null Statement	N	
Stop	=	STOP
Return	R	RETURN, EXIT

PCTNS 94/05182

IPEA/US 09 DEC 1994

-19-

Table 1 (continued)

Unconditional Branch	*	GO TO
Loop Terminal Statement	C	CONTINUE; ENDDO
Assigned GO TO	+	
ASSIGN	S	
Entry of GO TO	/	
ENDIF	!	
Opening of Logic Condition	{	
Closing of Logic Condition	}	
Logic Condition Alternative	V	ELSE; OR

Definition F1: A Reduced Logic Condition (LC) is a process element with one control input and two control outputs. One of the two control outputs are assigned to Boolean 0 (False) and the other control output is assigned to Boolean 1 (True) which correspond to a solution to the logic condition.

Definition F1.1: The Low Potential Solution of a logic condition is the "0" Boolean solution (False) to the logic condition.

Definition F1.2: The High Potential Solution of a logic condition is the "1" Boolean solution (True) to the logic condition.

Definition F2: The Process Terminals are comprised of the following members of the eX-Object set shown in Table 1: *, +, C, R, =.

Definition F3: The Main Branch of a process is the path traversed through a process from an origin of the main branch (process entry or high potential solution of logic condition) to a process terminal when

This Sheet Has Been Replaced



PCT/US 94/05182
IPEA/US 09 DEC 1994

-20-

all of the logic conditions which comprise that path are passed through their low potential solutions.

Referring to FIG. 9, the Main Branch is the path from entry 900, through logic conditions 904, 914 and 924, until terminal 930 is reached.

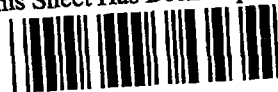
Definition F4: Exit Terminals result in an exit from a process and are comprised of the following members of the eX-Object set shown in Table 1: R, =. As shown in FIG. 9, terminal 960 is an exit terminal.

Definition F5: Rewind Terminals are terminals that rewind the process to an element that is in the same main branch or in a main branch to the left of the current main branch. An exception to this rule is when the rewind terminal is an exit of a loop. In such a case, the rewind terminal could reset the process to an element that is within a branch located to the right of the current branch. As shown in FIG. 9, terminals 908, 930, 944, 950, 952 and 958 are all rewind terminals.

Definition F6: A t-pass is a pass from the process entry to the process terminal. In any process there are (n+1) t-passes, where n is the number of logic conditions in the process. Each t-pass is uniquely identified by its terminal.

In FIG. 9 there is shown a single entry 900, and seven terminals; 908, 930, 944, 952, 950, 960, and 958. Of those seven, 930, 944, 952, 950, 958, and 908 are all rewind terminals while terminal 960 is an exit terminal. Shown further in FIG. 9 are six logic conditions; 904, 914, 924, 938, 946 and 956. A single

This Sheet Has Been Replaced



AMENDED SHEET

-21-

t-pass would be represented by entry at 900,
progressing along line 902, taking the high potential
path from logic condition 904 along line 906 until
terminal 908 is reached. This t-pass would be uniquely
5 identified as, for example, t-pass 908 where 908
identifies the terminal of the t-pass.

Definition F7: A branch of a logic condition is
that part of a t-pass which begins at that logic
10 condition. Each logic condition has at least one
0-branch and at least one 1-branch. Referring once
again to FIG. 9, a high-potential or 1-branch of logic
condition 904 is represented by line 906 and terminates
at terminal 908. There are two 1-branches of logic
15 condition 914 and they are ended by terminals 960 and
958.

Three of the four elements contained within
dotted block 964, namely 954, 956, and 960, constitute
20 a main branch of logic condition 914. The other
element contained within dotted block 964, namely 958,
constitutes a main branch of logic condition 956. The
combination of all the main branches of a process
necessarily and sufficiently represent all the elements
25 which comprise that process. Further, there could
exist many paths through a process element but that
process element can belong to only one main branch.

Definition F8: The 0-subfield of a logic
30 condition is a combination of all elements situated on
the 0-branch(es) of that logic condition.

Definition F9: The 1-subfield of a logic
condition is a combination of all elements situated on
35 the 1-branch(es) of that logic condition.

AMENDED SHEET

This Sheet Has Been Replaced



-22-

Definition F10: The field of a logic condition is a combination of the 0-subfield and 1-subfield of that logic condition.

5 Definition F11: The level of the logic condition field or logic level, is the numeric order in which that logic condition was encountered in the t-pass going through the logic condition.

10 Referring once again to FIG. 9, logic condition 904 is situated at logic level 1. Logic condition 914 is situated at logic level 2. Logic conditions 924 and 956 are both situated at logic level 3. Logic condition 938 is situated at logic level 4.
15 Logic condition 946 is situated at logic level 5.

Definition F12: A process segment is that part of a main branch which is positioned between process entry, process terminals and logic conditions.

20 Process terminals and logic conditions terminate process segments. They belong to the process segments that they terminate.

25 An eX-Frame is constructed in the present invention by representing the main branches of the target process in a special order along with representing the levels of the logic conditions and their corresponding fields.

30 Construction of the eX-Frame

The first step performed by eX-builder 259 is the construction of the eX-Frame which is
35 representative of the target process 250. This step is

This Sheet Has Been Replaced



AMENDED SHEET

-23-

not needed if the eX-Graph is constructed directly from the target process. The target process is processed by the eX-Builder through the "Next Lowest Potential" rule or alternatively, through the complementary "Next Highest Potential" rule. FIG. 22 is a flow chart depicting the steps associated with the analysis of a software process by the next lowest potential rule. The eX-Builder proceeds by beginning its analysis at start block 2200. Successive statements of the process are parsed by block 2202 until a logic condition or process terminal is met. If a logic condition is met as determined by block 2204, the low potential solution of this logic condition is taken as dictated by block 2206. When a rewind or exit terminal is encountered as determined by block 2208 and untraversed high potential paths remain in the software process as determined by block 2210, the block 2214 will be executed. The eX-Builder locates the logic condition, that was last passed through its low potential solution and which high potential path was not yet taken. eX-Builder then takes its high potential path and proceed via block 2202, therefore building the Main Branch of that logic condition. These steps repeat until no untraversed high potential paths are left in the target software process.

In order to provide the reader with additional clarity, FIG. 23 shows a PSEUDOCODE implementation of the steps involved in analyzing a target software process by the next lowest potential rule. Additionally, FIG. 23 shows at what points within the steps modifications are made to the eX-Frame under construction and at what points in the process statements are stored in the eX-Repository.

AMENDED SHEET

This Sheet Has Been Replaced



PCT/US 94/05182

IPEA/US 09 DEC 1994

-24-

FIG. 2 shows an eX-Frame 20 which was constructed from the flow chart of a target process shown in FIG. 1 by the next lowest potential rule. By inspection of FIG. 2, the reader can readily see that the eX-Frame 20 is built in the form of an algebraic expression containing several levels of parenthesis. In eX-Frame 20, the parenthesis are representative of the logic levels. The same general rule is applied to this eX-Frame 20 that applies to any algebraic expression, that is, the lower (outer) level logic field cannot be closed until the higher (inner) level logic field is first closed.

Referring once again to FIG. 2, the different levels of eX-Frame 20 become apparent. Dotted line 21 identifies that portion of eX-Frame 20 which is the first level. Line 22 represents the second level. Dotted lines 23 show the third level. Dotted line 24 shows the fourth level and dotted line 25 shows the fifth level. Within the eX-Frame, the fields of logic conditions are between the corresponding parenthesis and integers corresponding to the parenthesis represent the logic levels of those fields.

The labeling of the target process elements which are represented within the eX-Frame 20 proceeds using the eX-Object set members previously defined. Table 2 shows the eX-Object set members of the resulted eX-Frame and their respective process segments.

AMENDED SHEET

This Sheet Has Been Replaced



PCF#10
IPEA/US 94/051382

-25-

Table 2.	
Process Segments	eX-Object Set Representation
1	L1
2	/7, A1, /1, L2
3	A2, L3
4	X1, *1
5	A3, /2, L4
6	A4, /3, X2, *2
7	/4, A5, /6, L5
8	A6, *3
9	W1, *4
10	A7, L6
11	R1
12	*6
13	E1, *7

Upon completion of the construction of the
eX-Frame 20, an eX-Graph is built by the eX-Builder
259. An eX-Graph of target process 250 shown as a flow
chart in FIG. 1, is shown in FIG. 5(A). The eX-Graph
is constructed along two perpendicular axis which are
designated as k and b in the graph and are referenced
by 501(A) and 500(A) respectively. The origin of the
two perpendicular axis is at the top-left of the graph.

Main branches of the graph are directed
vertically down and represent tracks by which the
target process 250 execution progresses, unless the
track is switched as the result of the high potential
solution to a logic condition encountered in the track.
As shown in FIG. 5(A), every track has a unique b
coordinate. Every track of the graph is parallel to
one another.

AMENDED SHEET

This Sheet has Been Replaced



PCT/US 94/05182
IPEA/US 09 DEC 1994

-26-

The graph is constructed and populated such that an eX-Object set member representative of the target process 250 is inserted at appropriate k,b coordinate(s). For example, as shown in FIG. 5(A) at k,b coordinate 1,1 there is placed an eX-Object set member '[' (left brace) which is indicative of a process entry as defined previously in Table 1.

In the following description of the construction of the eX-Graph, the below given designations and definitions will be used:

Definition G1: A Reduced Flow Chart is an eX-Graph represented in the form of a flow chart. FIG. 4 shows a reduced flow chart for the target process represented by the flow chart of FIG. 1.

A reduced flow chart is presented along two perpendicular axis, k and b which are shown as 401 and 400 in FIG. 4, respectively. The origin of the two axis is at the top-left-most portion of the chart. The k axis increases as one proceeds down the chart and the b axis increases as one proceeds to the right of the chart.

All possible changes in the direction of control is reduced to one - to the right as the chart is traversed. In this manner, all high potential solutions to a logic condition are represented as a shift to the right in the graph. Only backward (up and left) jumps of unconditional control statements are possible when a software process is represented by a reduced flow chart. An exception to this general rule is an exit from a loop. Such an exit may be made forward.

AMENDED SHEET

This Sheet Has Been Replaced



PCT/US
IPEA/US 9/9 DEC 1994

-27-

Definition G2: The negative state of a logic condition is the state of a logic condition which, when evaluated results in its 0-solution or low potential solution. With reference to FIG. 5(A), logic condition L1 is represented at k,b coordinate 2,1. Traversing the eX-Graph shown in FIG. 5(A) through L1 with L1 in its negative state would result in relocating to k,b position 3,1 which is represented by eX-Object set member /7 in the graph.

Definition G3: The positive state of a logic condition is the state of a logic condition which, when evaluated results in its 1-solution or high potential solution. With reference to FIG. 5(A), traversing the eX-Graph shown in FIG. 5(A) through L1 with L1 in its positive state would result in relocating to k,b position 3,7 which is represented by eX-Object set member E1 in the graph.

Definition G4: The binary address (BA) of an eX-Graph element shows that elements position relative to process logic constructs and is represented by the integers 1 and 0 where the integers comprising the BA represent the solutions of the logic conditions in the t-pass up to that element with the process entry assigned the binary address of 1.

The binary address of those elements within the same process segment are the same. The binary address of a 0-segment of a logic condition is generated by shifting the address to the left by one bit position and placing 0 at the rightmost position of the binary address. The binary address of a 1-segment of a logic condition is generated by shifting the

This Sheet Has Been Replaced



AMENDED SHEET

94/05182
IPEA/US 09 DEC 1994

-28-

address to the left by one bit position and placing a 1 at the rightmost position of the binary address.

Referring once again to FIG. 5(A), the binary address of k,b position 8,1 (L3) is 100. This is constructed from the process address (binary address 1), the negative solution (0) to the logic condition L1 located at k,b position 2,1 and a negative solution to logic condition L2 located at k,b position 6,1 in the graph. As a further example the binary address of A7 (located at k,b position 7,5 in FIG. 5(A)) is 101. This is constructed from the process address (binary address 1), the negative solution (0) to the logic condition L1 located at k,b position 2,1 in the graph and the positive solution (1) to the logic condition L2 located at k,b position 6,1 in the graph.

Construction of the eX-Graph

The eX-Graph 262 which is representative of target process 250 is built by the same next lowest potential rule previously described. The eX-Graph 262 may be built from a previously constructed eX-Frame 261 or built directly from analyzing the target process 250.

FIG. 26 is a flow chart depicting the steps associated with the analysis of a software process by the next lowest potential rule and the construction of the resulting eX-Graph. The analyzer proceeds by beginning its analysis at start block 2600. The position on the graph is assigned to k,b position of 1,1 2601, and the entry binary address is assigned to 1 as indicated in block 2602. Successive statements of the process are examined by block 2603 with the

This Sheet Has Been Replaced

AMENDED SHEET



-29-

eX-Graph(k,b) position being populated with the
eX-Object set member representative of the examined
statement as required by block 2604. The repository is
updated in block 2605 and if the statement under
examination presently is a logic condition 2606, then
the low potential solution to the logic condition is
taken 2609, the present k,b position is updated by $k = k+1$
as stated in block 2612 and the statement
reexamination block 2603 is reentered.

If upon examination by block 2606 it is
determined that the statement is not a logic condition,
then the statement is tested in block 2607 to see if it
is a terminal. If the statement so tested is not a
terminal statement then the present k,b position is
updated by $k = k+1$ (block 2608) and the statement
reexamination is reentered.

If the statement examined by block 2606 is
not a logic condition and it is a terminal (block 2607)
then if it is a rewind terminal other than a loop
terminal the rewind entry is located. If it does not
exist, it is inserted in the eX-Graph (block 2614). A
determination is made at block 2616 as to whether any
further untraversed high potential paths exist (i.e., a
logic condition for which a main branch is not yet
built). If there are no remaining untraversed high
potential paths in the target process under
examination, then the graph is built and the eX-Builder
terminates 2615.

If there were untraversed high potential
paths as determined by block 2616, then the eX-Graph
position is updated to the last logic condition passed
through its 0-solution and not yet passed through its
1-solution (block 2617) and a new main branch of that

AMENDED SHEET

This Sheet Has Been Replaced



PCT/US 94/05182

IPFA/IIS 09 DEC 1994

-30-

logic condition is begun at eX-Graph position(nextk,
nextb) where nextk is assigned to k(logic condition) +
1 and next b is assigned to the last used b+1 (block
2618). The last used b is updated and the process then
5 continues with the examination statement 2603.

FIG. 5(A) shows an eX-Graph constructed by
the eX-Builder 259 using the above stated method of the
present invention. Once the target process 250 is
10 represented as eX-Graph 262 as constructed by the
eX-Builder 259, a number of useful characteristics of
eX-Graph 262 as shown in FIG. 5(A) emerge.

By examination of the eX-Graph shown in FIG.
15 5(A) it should be apparent that every element
represented within the eX-Graph may be addressed by a
unique coordinate pair (k,b). For example, logic
condition L1 is located at k,b position 2,1 in the
graph. Logic condition L6 is located at k,b position
20 8,5 in the graph. Table 3 shows the elements, their
corresponding and unique k,b coordinates and their
resultant binary addresses which are created during the
analysis of target process 250

In the present preferred embodiment, the
25 eX-Graph is built in only two directions - down and to
the right as one looks at the graph. When the eX-Graph
so constructed is traversed, the traversal proceeds
down the graph and can only be interrupted by shifting
30 to the right as a result of a high potential solution
to an encountered logic condition.

Referring once again to FIG. 5(A) the graph
traversal would begin at k,b position 1,1 (process
35 entry) and proceed through logic condition L1 (k,b

This Sheet Has Been Replaced



AMENDED SHEET

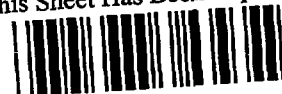
-31-

position 2,1), through label /7 (k,b position 3,1),
through arithmetic A1 (k,b position 4,1) and so on
until terminal *1 (k,b position 10,1) is reached. Such
a traversal would proceed in a single direction, down
the graph. If however, a high potential solution to a
logic condition was passed through during traversal
then a shift to the right would have taken place. For
example, during the above traversal, if the logic
condition L1 (k,b position 2,1) were passed through its
high potential solution then a shift in flow would take
place to E1 (k,b position 3,7).

Further, every segment is assigned a binary
address that uniquely identifies the logic position of
that segment within the target process 250. FIG. 5(B)
shows the binary addresses which are assigned to the
elements which are contained within eX-Graph depicted
in FIG. 5(A). For example, logic condition L1 (k,b
position 2,1) is assigned to binary address 1, logic
condition L2 (k,b position 6,1) is assigned binary
address 10, output statement W1 (k,b position 16,4) is
assigned binary address 100111. Table 3 shows the
binary address which correspond to the elements which
comprise target process 250.

AMENDED SHEET

This Sheet Has Been Replaced



PCT/US 94/0518:

IPEA/US 09 DEC 1994

-32-

Table 3		
eX-Object Set Member	(k,b) Position	Binary Address
[	1,1	1
L1	2,1	1
/7	3,1	10
A1	4,1	10
/1	5,1	10
L2	6,1	10
A2	7,1	100
L3	8,1	100
X1	9,1	1000
*1	10,1	1000
A3	9,2	1001
/2	10,2	1001
L4	11,2	1001
A4	12,2	10010
/3	13,2	10010
X2	14,2	10010
*2	15,2	10010
/4	12,3	10011
A5	13,3	10011
/6	14,3	10011
L5	15,3	10011
A6	16,3	100110
*3	17,3	100110
W1	16,4	100111
*4	17,4	100111
A7	7,5	101
L6	8,5	101
R1	9,5	1010
*6	9,6	1011

This Sheet Has Been Replaced



AMENDED SHEET

PCT/US 94/05182

IPEA/US 09 DEC 1994

-33-

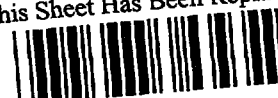
E1	3,7	11
*7	4,7	11

Lastly, once eX-Graph 262 construction is completed by eX-Builder 259, the accessibility between two elements of target process 250 may be effectively determined. Accessibility between two elements s1, and s2, is defined as the presence of a path from s1 to s2. That is, s2 is accessible from s1 if there is some possible input data combination that results in s2 being executed some time after the execution of s1. Accessibility between two process elements is required if one element is capable of effecting the result of the second element.

Referring once again to FIG. 5(A), it becomes obvious that statement A7 (k,b position 7,5) is not accessible from statement W1 (k,b position 16,4). That is because there is no possible path from statement W1 that would proceed through statement A7. Statement A7 is however, accessible from statement X1 (k,b position 9,1). That is because statement *1 (k,b position 10,1) repositions the path taken to /1 (k,b position 5,1) which immediately precedes statement L2 (k,b position 6,1). The high potential solution of L2 leads directly to statement A7. Therefore, the present invention is able to determine whether statement(s) or elements of target process 250 could be influenced by the correctness of a particular statement or statements within that process. When the target process 250 is represented in eX-Graph form as shown in FIG. 5(A) and is used subsequently for the construction of eX-Model 243, the invention is capable of evaluating the possibility for a statement s1 of target process 250 to

AMENDED SHEET

This Sheet Has Been Replaced



-34-

be responsible for the misbehavior of statement s2 of target process 250 by defining accessibility between s1 and s2 as a necessary condition for existence of cause-effect relationship between those two elements. Alternatively, the analyzer is capable of evaluating the potential for statement s2 to be effected by the modification of statement s1.

The accessibility between statements that comprise the eX-Graph may be defined with respect to three potentials: 1) the free fall potential within a main branch; 2) the shift potential of a logic condition; and 3) the rewind potential of a terminal.

Free fall potential is determined by one process element becoming active after another unless the direction of execution is changed by a logic condition in a high potential state. Quite simply and with reference to FIG. 5(A), a free fall potential exists between arithmetic statement A1 (k,b position 4,1) and arithmetic statement A2 (k,b position 7,1) because A2 follows the logic condition if target process 250 fell through the logic condition L2 (k,b position 6,1) in its low potential state until terminal *1 (k,b position 10,1) is reached.

The shift potential is the potential of a logic condition to increment the eX-Graph b coordinate as a result of a positive solution of the logic condition. With reference to FIG. 5(A), logic conditions L3 (k,b position 8,1), L4 (k,b position 11,2), L5 (k,b position 15,3) and L6 (k,b position 8,5) all have a shift potential of 1 because a positive solution to any of them results in a right shift by one of the b coordinate. Similarly, logic condition L2

AMENDED SHEET

This Sheet Has Been Replaced



PCT/US 94/05182

IPFA/US 09 DEC 1994

-35-

(k,b position 6,1) has a shift potential of 4 and logic condition L1 (k,b position 2,1) has a shift potential of 6.

In order to appreciate the significance of the eX-Graph structure and the information which the analyzer derives from its inventive structure, some additional definitions are required to facilitate understanding.

Definition G6: *bn will refer to the terminal with b-coordinate equal n. With reference to FIG. 5 *b3 refers to the GO TO terminal on the b=3 and *b5 refers to "exit" terminal on b=5.

Definition G7: Mbn will refer to a Main Branch built on the b-coordinate equal n. By way of example and with reference to FIG. 5, MB3 refers to the Main Branch built on coordinate b=3.

Definition G8: The shift potential of LC is the potential of LC to increment b-coordinate as a result of a positive solution of the LC.

Definition G9: The rewind potential of a terminal is the potential of that terminal change the next active Segment Address (SA).

Definition G10: The segment address (SA) is defined as a pair (BA,k), where BA is the binary address of the segment and k is a k coordinate within the segment. For example with reference to FIG. 5(A), the segment address of eX-Object set member /3 is 10010,13 Further, segment addresses can be compared by

This Sheet Has Been Replaced



AMENDED SHEET

-36-

the present invention in a manner similar to the comparison of binary addresses.

To compare binary addresses, the software analyzer left justifies the binary addresses, then corresponding bit position contents are compared from left to right until a mismatch occurs. When comparing binary address bit position contents, 1 > 0 > empty. By examining FIG. 5(A), 5(B) and Table 3, the comparison of binary addresses becomes apparent. The binary address of logic condition L2 (k,b coordinate 6,1 ; binary address 10) is greater than the binary address of logic condition L1 (k,b coordinate 2,1; binary address 1). The binary address of E1 (k,b coordinate 3,7; binary address 11) is greater than the binary address of L2 (binary address 10).

To compare Segment Addresses (SA), the binary address (BA) component of the SA is given greater priority, i.e., a segment address of 11,7 is greater than a segment address of 10,10 and segment address 11,7 is greater than a segment address of 11,6. Therefore, the segment address of L5 (FIG. 5) is greater than the segment address of A5 because their binary addresses are equal and the k value of L5 is greater than the k value of A5.

Definition G11: The vision of an element is the set of all elements which lie within any t-pass which passes through that element. An element s2 is within the vision of element s1 (or element s2 is seen from element s1) if the segment address (SA) of s2 is a descendant of the SA of s1. A statement s2 is within the vision of statement s1 if there is a path from s1 to s2 that is part of a t-pass.

AMENDED SHEET



-37-

Definition G12: A segment address of a statement is a descendant of the segment address of another statement if the binary address of the second statement is built from the binary address of the first statement by concatenating binary digits at the right, or if the BA(s1) is equal to the BA(s2) and the k coordinate of s2 is greater than the k coordinate of s1.

10 With reference to FIG. 4., A6 424 is within the vision of A3 412 since the segment address of 424 (100110) is a descendant of the segment address of 412 (1001). By way of an additional example, the binary address 1001 can be mapped onto 10011, therefore any
15 element within segment 10011 is within the vision of an element within segment 1001. Further, element X2 418 (binary address 10010) lies within the vision of element A4 415 (binary address 10010) since they have the same binary address and the k coordinate of X2 (14)
20 is greater than the k coordinate of A4 (12) when mapped to the eX-Graph.

Definition G13: The vision of a rewind terminal is defined by the vision of its rewind entry.
25 The vision of the exit terminal is empty. As an example, refer to FIG. 9. The vision of rewind terminal 1 930 is the vision of its rewind entry 912. The vision of exit terminal 960 is empty.

30 Definition G15: The access field of a process element is the combination of all the visions that the element successively owns. When an element sees a terminal, a vision of that terminal, which is the vision of its rewind entry may see some terminals
35 which were not originally seen. And so on...therefore

AMENDED SHEET

Sheet Has Been Replaced



-38-

expanding the access field of the examined process element.

5 Definition G16: The rewind binary address
(RBA) is the binary address of the rewind terminal
entry. The rewind binary address of the exit terminal
is 0. Referring to FIG. 5(A), the RBA of terminal *1
(k,b coordinate 10,1) is defined by its terminal entry
/1 (k,b coordinate 5,1) Therefore the RBA of terminal
10 *1 is the binary address of /1, which is 10. The
position of rewind terminal entry is identified by a
pair (BA,rk), where rk is the k coordinate value of the
rewind terminal entry. The rk value of the exit
terminal is zero.

15 Definition G19: An open terminal is an exit
terminal or rewind terminal that has access to an exit
terminal.

20 Definition G18: A closed terminal is a
rewind terminal that does not have access to an exit
terminal.

25 Definition G17: The exit access indicator
(EAI) is an indicator which is set to 0 for closed
terminals and set to 1 for open terminals.

30 Definition G20: A closed region is a
combination of adjacent main branches which are
identified by closed terminals. Once a target process
250 enters a close region, that process cannot
terminate. Referring to FIG. 9, terminals 944, 952 and
950 identify a closed region.

This Sheet Has Been Replaced



PCT/US 94/05182
IPEA/US 09 DEC 1994

-39-

Definition G21: The propagated vision point (PVP) of a process terminal is defined by the vision of all the terminals that the process terminal sees. As previously defined, the vision of a terminal is the vision of its rewind entry. If the rewind entry of a first process terminal can see a second terminal whose rewind entry can see the rewind entry of the first process terminal, then the vision of the second rewind entry becomes the propagated vision point for both the first and second terminal.

By way of example and with reference to FIG. 10, if rewind entry /n 1001 of process terminal *n 1004 can see another terminal *i 1002 whose rewind entry /i 1000 can see the rewind entry /n 1001, then the vision of *n 1004 is propagated by the vision of *i 1002 and the vision of *i 1002 becomes the propagated vision point of both terminals *i 1002 and *n 1004. Therefore the propagated vision point of terminals *i 1002 and *n 1004, is the rewind entry of /i 1000.

Further, the terminal *k 1005 has a rewind entry /k 1003 which sees another terminal *n 1004. The propagated vision point of terminal *k 1005 is therefore propagated to the vision of terminal *n 1004, that had been already established as propagated to /i 1000.

The propagated vision point of a terminal is represented by a segment address of a specific rewind entry. If a terminal vision is not propagated by the visions of other terminals, then the propagated vision point of that terminal is its own vision, i.e., the vision of its rewind entry.

This Sheet Has Been Replaced



-40-

When all of the terminals within the vision of a rewind entry have that entry as their propagated vision point, the result is a closed region (Data Independent Endless Loop). This is because none of the terminals within the vision of that rewind entry have had an access to an exit terminal since the vision of an exit terminal would have been zero.

Once the eX-Graph 262 is constructed, the exit access indicator (EAI) table is constructed. FIG. 9 shows a sample EAI table constructed from the terminal binary address, the rewind binary address and rk of each process terminal, all of which are known once the eX-Graph is constructed.

FIG. 9 shows an EAI table which is constructed according to the following method. First, the original value of the propagated vision point of a terminal is built from that terminal rewind entry, that is, the rk and RBA of the terminal. Second, the EAI of all rewind terminals are set to 0 (set closed) and the EAI of all exit terminals are set to 1 (set open). Lastly, and proceeding from $b = 1$ to the maximum b of the eX-Graph, each terminal PVP is compared with the Binary Address and PVP of every other terminal in an attempt to propagate (decrement) the PVP of the current terminal under examination and attempt to open that terminal if closed. If the PVP of a closed terminal can see any open terminal, then the examined terminal is defined as open by setting its EAI to 1.

By way of further explanation and with reference to the EAI table shown in FIG. 9, the construction of the EAI table proceeds as follows:

AMENDED SHEET

This Sheet Has Been Replaced

PCT/US 94/0518
IPEA/US 09 DEC 1994

-41-

1) Set the PVP of terminals to the segment address of their rewind entries:

BA of PVP of terminal *b1 (PVP*b1) is set to 10;

5

BA of PVP*b2 is set to 1001;

BA of PVP*b3 is set to 10010;

BA of PVP*b4 is set to 10011;

BA of PVP*b5 is set to 0, since *b5 is an exit terminal;

10

BA of PVP*b6 is set to 10011;

BA of PVP*b7 is set to 10;

2) Propagate the PVP and Set the EAI. If the PVP of a closed terminal can see any open terminal, then the examined terminal is defined as open by setting its EAI to 1.

15

a) PVP*b1 {10,rk1} sees *b2,*b3,*b4,*b5, but neither of those terminals PVPs can see PVP*b1.

20

Therefore

PVP*b1 is not changed;

PVP*b1 {10,rk1} sees open *b5 (1010);

open *b1: set EAI*b1 to 1.

25

b) PVP*b2 {1001,rk2} sees *b3,*b4 whose binary addresses are 100110 and 100111, but neither of these terminals can see the PVP*b2. Therefore,

PVP of *b2 is not changed;

30

PVP*b2 {1001,rk2} does not see any open terminals;

EAI*b2 is kept 0;

AMENDED SHEET

This Sheet Has Been Replaced



-42-

c) PVP*b3 {10010,rk3} sees *b2 {10010} and
PVP*b2 {1001,rk2} sees PVP*b3, that is, {10010, rk3}.
Therefore,

5 set PVP*b3 to PVP*b2: {1001,rk2};
 propagated PVP*b3 still does not see any
opened terminals;
 EAI*b3 is kept 0;

10 d) PVP*b4 {10011,rk4} sees *b3 {100110}, and
PVP*b3 {1001,rk2} can see PVP*b4 {10011,rk4};
 set PVP*b4 to {1001,rk2};
 propagated PVP*b4 still does not see any open
terminals;
 EAI*b4 is kept 0;

15 e) PVP*b5 is not changed, since it is an exit
terminal;

20 f) PVP*b6 {10011,rk6} sees *b3 and *b4, whose
current PVPs {1001,rk2} can see PVP*b6;
 set PVP*b6 to {1001,rk2};
 propagated PVP*b6 still does not see any
opened terminals;
 EAI*b6 is kept 0;

25 g) PVP*b7 {10,rk7} can see terminals from *b1
to *b6, but neither of them can see PVP*b7;

30 PVP*b7 is not changed;
PVP*b7 {10,rk7} can see open terminals {1000
and 1010};
 open *b7: set EAI*b7 to 1;

35 In some target processes and with reference
to FIG. 10(B), terminal *i 1006, that is propagated by

This Sheet Has Been Replaced



ADDITIONAL

-43-

the vision of *n 1007, is examined before the PVP of *n 1007 is finally set, since the PVP of *n 1007 may in turn be upgraded by the vision of the other terminals. Therefore, the only way to make sure, that all the possible upgrades of PVP and EAI were performed, is to repeat the step (2) of examining all the terminals in the order from *b1 to *bn once again. If no upgrades were done within the last pass, the process of upgrading the PVPs and EAIs is finished.

In the worst case, the number of required passes through the terminals, i.e. executions of step (2) is equal to the number of Main Branches, NB, since each pass will finally set the PVP and the EAI of at least one terminal. In the above example all upgrades were performed in one pass, and the second pass through *b1 to *b7 will not change any PVP or EAI.

Once the exit access indicator table is constructed, Propagated Vision Points built into this table allow one to define the accessibility between any two elements of the graph. As a consequence, all data independent endless loop constructs are located, as those constructs do not have accessibility to an exit terminal.

It is commonly accepted that the question of whether or not a program will terminate is proven to be unsolvable. We can divide this question of process "terminability", i.e. ability to terminate, into two categories: 1) data independent endless loops and 2) data dependent endless loops.

A data independent endless loop consists of process elements that do not have an access to "exit"

AMENDED SHEET

This Sheet Has Been Replaced


-44-

terminals. A data dependent endless loop is such a construction that after entering it the control will never reach an exit (STOP or RETURN) by the reason other than the absence of control passes to these exit terminals. The reason in this case is either: a) the absence of the manipulation of program variables, that could potentially change the control pass, on the specific control passes, or b) change of these program variables in the direction that takes the values of LCs further from changing their states between Low Potential State and High Potential State.

In both cases a and b above, the control will fall into an endless loop, going through the same terminals in the same sequence, without ever being able to reach an "exit" terminal.

As an example of data dependent endless loop and with reference to FIG. 9, if there are no elements on the pass between /1 912 and *1 930 that can redefine the values, being examined by L2 914 and L3 924, then terminal *b1 930 will produce a data dependent endless loop as soon as L2 914 and L3 924 will be resolved by their Low Potential solutions the first time (since the state of L2 914 and L3 924 will never change during the forward execution of the target process.)

As an example of a data independent endless loop and with reference to FIG. 9, a data independent endless loop construction is presented by Main Branches MB2, MB3, MB4. If L3 924 is ever solved positively, the process will go in the infinite loop, no matter, what data manipulation statements are coded within those branches.

AMENDED SHEET

This Sheet Has Been Replaced



PCT/US 94/05182

IPEA/US 09 DEC 1994

-45-

While process constructions that could allow for data dependent endless loops present a potential for process "non-terminability", they are not necessarily faulty constructions, since it could be so, that an input data can never be such, that would set LCs in the state of infinite loop.

Entry /q 922 in FIG. 9 does not correspond to any terminal and is shown only as an example of the fact, that the position of L3 924 in the negative subfield of L2 914 with rewind terminal *b1 rewinding the process control directly in front of L2 914 is not necessarily a fault, even if L2 914 and L3 924 are not dependent on the data manipulations that are coded on the path /1 912 to *1 930. This is, because that if L3 924 had received control through the entry /q 922, returning the control to the entry /1 912 will not necessarily result in L2 914 being solved negatively thus in creating an endless loop.

On the other hand the programs that contain Data Independent Endless Loops constructs are definite faults in the target process construction.

The detection of data independent endless loops is done by evaluating an accessibility between rewind terminals and exit terminals as it was explained on the previous example. The same process would allow evaluation of accessibility between any two elements of eX-Graph (for example between A2 and A7 in FIG. 9).

As a result we see: EAs of terminals *b2, *b3, *b4 and *b6 are 0, these terminals are closed. Terminals *b2, *b3, *b4 constitute a closed region (data independent endless loop). Terminal *b6 is also closed

This Sheet Has Been Replaced



PGDUS 94/01182
IPEA/US 09 DEC 1994

-46-

and its rewind entry is within the mentioned closed region [*b2 - *b4]. Dotted line 966 represents the vision field of an entry /1 {10,rk1}. This entry is PVP of *b1. This means, that any element of Main Branch b=1 can access any element within the vision of {10,rk1}.

An access field of A2 is represented by the dotted line 966. Dotted line 967 represents the vision field of an entry /2 {1001,rk2}. This entry is PVP of *b2, *b3, *b4 and *b6. This means, that any element of Main Branch MB2, MB3, MB4 and MB6 can access any element within the vision of {1001,rk2}.

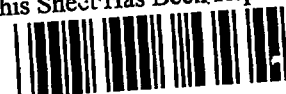
By definition, an access field of a process element is the combination of all the visions it successively owns. Therefore, an access field of A7 is combined by the fields represented by dotted line 964 and dotted line 967, as dotted line 964 is the vision of A7 and dotted line 967 is the vision of *b6 that is seen from A7.

From FIG. 9 we can see, that A2 can access A7, and A7 cannot access A2.

As an example from FIG. 9, examining the attributes (BA, RBA and PVP) that are built for every terminal, we can determine the following: a) A2 can not see A7, as 101 is not seen from 100. However, A2 can access A7, since one of the terminals, seen from A2, *b1, can see A7 through its PVP {10,rk1}, as 101 is a descendent of 10; b) A7 can not see A2, as 100 is not a descendent of 101; the terminals that are seen from A7 are ones with BAs 1010 and 1011; Terminal 1010 has no RBA or PVP, being an exit terminal; terminal 1011

AMENDED SHEET

This Sheet has Been Replaced



PCT/US 94/05182
IPEA/US 09 DEC 1994

-47-

has PVP {1001,rk1}, which can not see 100 either.
Therefore A7 can not access A2.

5 Data dependent endless loops could be
determined using our method of propagated vision points
in the following manner. Once the sequence of
terminals executed repeats many times it could become
useful to locate the propagated vision point of those
terminals and then to see whether or not there are any
10 data manipulation statements on the t-passes from this
PVP to those terminals which data manipulation
statements affect the variables checked by the logic
conditions located on the t-passes from this PVP to
those terminals. If no such statements exist, we have
15 a data dependent endless loop. If such statements
exist but are constructed in a way such that the
variables change in the direction that would take the
value calculated by the logic condition further from
the value that would change the status of the logic
20 condition. In this case we also have a data dependent
endless loop.

Determining Process Correctness

25 As a result of the structures which are
constructed by the eX-Analyzer it is possible to
evaluate the correctness of a target process. In order
for those skilled in the art to readily appreciate how
the eX-Analyzer operates, the following definitions are
30 necessary at this point.

Definition P1: The symbol (s), represents a
process element or individual statement of a program.

AMENDED SHEET

This Sheet Has Been Replaced


PCT/US 94/05182

IPEA/US 09 DEC 1994

-48-

Definition P2: The Process Address, (PA)
{k,b} of an element is the position of the element
within the reduced flow chart of the process or the
position of the element within eX-Graph of that
process. With reference to FIG. 5, the PA of an
element is defined by the k,b coordinate pair which
describes the position of the element within the
reduced flow chart. With reference to FIG. 5, element
A2 has a PA of 7,1. Each statement that comprises a
process has a unique PA.

Definition P3: The local time (p) parameter
is an integer value which represents the present
occurrence of a PA within a process execution. Local
time is started with the initial invocation of the
process. The local time can only be measured at a PA.
The eX-Analyzer does not utilize time as it is normally
used, i.e., a measurement of duration. Instead, time
as recorded by the eX-Analyzer is a measurement of the
number of occurrences of a particular statement.
Therefore each event or occurrence of a statement
execution is identified by its own time parameter.

Definition P4: A process event is an event of
a process statement execution. A process event is
defined by {k,b,p}, where k,b are the coordinate pair
which identify the process address and p is the local
time within this process address.

Definition P5: The Process time is
represented by process events {k,b,p}. As an example
and with reference to FIG. 5(A), when statement A5 is
executed the third time, that moment of execution is
referred to as the (13,3,3) event.

This Sheet Has Been Replaced



Admitted Sheet

-49-

Definition P6: A process event $\{k,b,p\}$, other than an output event, is correct if that event takes part in a chain of events that effect a correct output event. An output event that is an event of execution of system output statement is correct if it produces expected output result.

Definition P7: A Correct Process Address (CPA) contains a process element $\{s\}$ that had been executed as a correct event.

Definition P8: An Uncertain Process Address (UPA) contains a process element $\{s\}$ that has not yet been executed as a correct event.

A state of PA as UPA or CPA, is relative to some other time in the process. This state could be changed at some process time $\{ke,be,pe\}$ which is an event of execution of some other process address $\{ke,be\}$ which event could define $\{s\}$ event execution as correct or could make it uncertain.

As an example, consider a process defined by the chain of events $e_1, e_2, e_3, e_4, e_5, e_6$ and e_7 where e_7 is a correct output event, e_1 and e_3 are input events and e_1, e_2, e_3, e_4 and e_6 were involved in producing the correct output event e_7 . Therefore, events e_1 through e_7 except e_5 are correct. Event e_5 may be defined as correct by some other correct output event chain in which e_5 took place, but until such a time its correctness is uncertain.

Definition P9: The algorithm space of a process is the space defined by the combination of all process addresses $\{ki, bi\}$. With reference to FIG. 5,

RENDERED SHEET

This Sheet Has Been Replaced



PCT/US 94/05182
IPEA/US 09 DEC 1994

-50-

the algorithm space of the target process described by the reduced flow chart is bounded by the maximum k and maximum b shown.

5 Definition P10: The correct algorithm space of a process is the space of all process addresses which are presently defined as correct process addresses.

10 Definition P11: The uncertain algorithm space of a process is the space of all process addresses which are presently defined as uncertain process addresses.

15 In both of the above definitions the correctness or uncertainty of a space is relative to the present time within the process. This is due to the fact that the eX-Analyzer has the ability to later
20 redefine a correct event as uncertain (losing the knowledge). The process address may become uncertain if it does not any longer have a history of being executed as a correct event.

25 Definition P12: The event definition space of an event (e) consists of process addresses of the event's ancestor events.

30 Definition P13: An ancestor event is a prior event which took part in producing a later event by either: 1) producing a value of the later event (value ancestor event) or 2) producing a parameter for a conditional control event, i.e. an event of executing a conditional control statement (L or D of the ex_Object Set) in a path leading to the later event (control
35 ancestor event). If either of these conditions are

REMOVED SHEET

This Sheet Has Been Replaced


PCT/US 94/05182
IPEA/US 09 DEC 1994

-51-

present, then the later event is a descendent of the prior event.

By way of example and with reference to
5 FIG. 14, We(a) 1400 is a descendent of the events shown
below We(a) and within its branches. Further, e(d) is
a descendent of e(i) 1410, #*e(j) 1413, e(k) 1414,
*e(l) 1415, *e(m) 1418 and the other events that are
the ancestors of #*e(j) 1413, which are #e 1417, #e
10 1419, #e 1420, #e 1421.

The eX-Analyzer operates by preserving the
event definition space of correct events which, in
turn, preserves the elementary knowledge of process
15 element execution, i.e., the correctness or uncertainty
of this event. The eX-Analyzer continuously evaluates
the event definition space of new output events and
expands the correct algorithm space of a target process
250, when possible. On the other hand, when evaluating
20 the event definition space of output events found to be
incorrect, it may lose previously built knowledge of
correct algorithm space by losing the status of some
CPA to UPA.

25 The relationship between
A, L, D, E, X, *, +, C, !, /, R, =, S and W elements of the
eX-Object Set are such, that an approved event of a W
element execution activates a chain of secondary
processes. These secondary processes thereby approve
30 the execution events of those eX-Graph elements that
were involved in producing the approved event.

As a consequence of the principles we are
using, the fault(s) responsible for the failure of an
35 output result will be located within that part of the

AMENDED SHEET

This Sheet Has Been Replaced


PCT/US 94/05182
IPEA/US 09 DEC 1994

-52-

output result event definition space that lies within
the area of uncertain algorithm space. Likewise,
modifications made to a program as a result of new
system requirements must be located within that part of
the output result event definition space that lies
within the area of uncertain algorithm space. If there
is no uncertainty within the event definition space of
the results that must be modified, then the new
branches should be built in order to preserve correct
algorithm space.

Definition K1: The approval of an event is
the process of validating that events correctness by
the eX-Analytical Engine.

Definition K2: A control event is any event
which may potentially change the direction of a process
from falling down the main branch.

Definition K3: An unconditional control event
is an execution of rewind or exit terminals as defined
in the eX-Object set by (*,+,C,R,=).

Definition K4: A conditional control event is
an execution of a logic condition (L or D elements of
the eX-Object Set).

A positive solution of a conditional control
event results in a control shift of a process to the
right as viewed within an eX-Graph or reduced flow
chart. A negative solution to a conditional control
event results in the direction of a process to proceed
downward as viewed within an eX-Graph or reduced flow
chart.

AMENDED SHEET

This Sheet Has Been Replaced



-53-

Definition K5: A data manipulation event is an execution of an (A) or (E) or the arithmetic part of a (D) element of the eX-Object Set.

5 Definition K6: An initial data manipulation event" is one of the following: E-event, A-event of data initialization or D-event of the loop index initialization.

10 Definition K7: A Secondary Data Manipulation event is any data manipulation event other than an initial data manipulation event.

15 Definition K8: An elementary knowledge of a process element located within $PA(k,b)$ is obtained, when one of its execution events is approved. Accumulating knowledge about a process is performed by accumulating information about the correctness or uncertainty of the process element's execution events.

20 Definition K9: Elementary knowledge propagation is a process by which a elementary knowledge of an event is applied to different process addresses. Elementary knowledge propagation is
25 performed by locating and analyzing all of the ancestor events up to a knowledge propagation terminal events.

30 Definition K10: A knowledge propagation terminal (KP terminal) is defined as an event that either has no ancestors within the target process (i.e., an initial data manipulation event) or is a previously approved event.

35 Terminating knowledge propagation at previously approved events stabilizes the process of

AMENDED SHEET

This Sheet Has Been Replaced



-54-

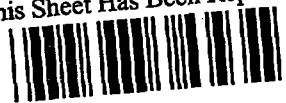
knowledge propagation and prevents the knowledge propagation process from becoming increasingly complex with the accumulation of ancestor events during the forward execution of a target process. When a
5 knowledge propagation process encounters a previously approved event, all of the ancestor events of the approved event were previously approved as well.

FIG. 14 shows events We(a) 1400 and We(n)
10 1401 above a tree of their ancestors. The process direction is indicated within FIG. 14 by arrow 1422, therefore event We(a) 1400 occurs at some time before event We(n) 1401. In this example, *e represents a KP terminal event, that is an initial data manipulation
15 event and #*e represents a KP terminal that is a previously approved event.

In FIG. 14, the value of We(a) 1400 was calculated from three parameters whose three values
20 were produced by the three events e(b) 1402, #*e(c) 1403 and e(d) 1404, respectively. Further, a value associated with e(b) 1402 was calculated from two parameters whose two values were produced by the two events *e(f) 1407 and #*e(g) 1408 respectively. Event
25 *e(f) 1407 does not have any ancestors, since it is an initial data manipulation event and is therefore a KP terminal. Event #*e(g) 1408 was previously approved (as indicated by #) and therefore it is also a KP terminal.

30 There is no attempt by the eX-Analyzer to locate the ancestors of a KP terminal. This is because such ancestors either: 1) are not present (as is the case with *e(f) 1407) or 2) have been previously
35 approved (#*e(g) 1408).

AMENDED SHEET

This Sheet Has Been Replaced


-55-

With reference to FIG. 17, event e(0) 1700 is shown as occurring at some process time (k0,b0,p0). Further, if event e(0) 1700 were the first approved event of s0 1701, then the PA of s0 1701 is upgraded from UPA to CPA as indicated by arrow 1710. This example shown in FIG. 17 indicates that a value of variable y is a function of three other parameters, x1 1702, x2 1703 and x3 1704.

The immediate ancestors of s0 1701 are the events which produced the value of the parameters involved in the s0 1701 calculation. The value of the parameters of s0 1701, namely x1 1702, x2 1703 and x3 1704 are determined by ancestor events #*e1 1705, #*e2 1706 and e3 1707. If events #*e1 1705 and #*e2 1706 were previously approved as shown, they are treated as KP terminals and we do not examine their ancestors.

Event e3 1707, which calculates parameter x3 1704 was not previously approved. The CPA status 1708 which was a state of the process address of statement s3 1709 during the process time (k0,b0,p0), means that some other event(s) of this PA execution was previously approved. Therefore, it is necessary to examine all of the ancestors of the event e3(k3,b3,p3) 1707 until a KP terminals are met during such examination on each direction.

Definition K11: The immediate owner of an event is another event which approved the event.

Definition K12: A W-owner of an event is a descendent of the event which originated the chain of

This Sheet Has Been Replaced



PCTA 94/05182
IPEA/US 09 DEC 1994

-56-

elementary knowledge propagation that approved the event.

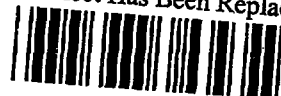
Each event has only one W-owner event and only one Immediate Owner event. In some situations, the W-owner of an event and the immediate owner of an event are one and the same.

As an example of event approval and with reference to FIG. 16, a process address (k,b) 1600 is shown during five different process times t1 1605, t2 1606, t3 1607, t4 1608 and t5 1609, respectively. These process times are defined by the execution of events at some address or addresses that could be other than (k,b). At time t1 1605, the PA 1600 had no occurrences of its execution. At time t2 1606, there has been two occurrences of PA 1600, e1 1610 and e2 1611, neither of which has been approved. At time t3 1607, there has been three occurrences of PA 1600, e1 1610, #e2 1611 and e3 1612. Further, event #e2 1611 has been approved during the interval which occurred between t2 1606 and t3 1607. The status of Process address 1600 therefore, is known to be CPA at time t3 1607. The W-owner attribute of CPA helps to maintain the current knowledge of the target process 250. When a previously approved W-event is disapproved, the target process knowledge is dropped to the level determined by the position of the disapproved W-owner event in the target process time.

The process of approving an event generates a secondary process which approves the immediate ancestors of the event unless the immediate ancestors are already approved or do not exist. Therefore, the propagation of elementary knowledge of the process

AMENDED SHEET

This Sheet Has Been Replaced



-57-

continues in the direction of all ancestors until a knowledge propagation terminal encountered in each direction.

5 Building the eX-Model of the Target Process

After the eX-Builder 259 finishes construction of the eX-Graph previously described, an eX-Model 243 of target process 250 is built. The invention constructs a different eX-Model for each distinct software process analyzed where the process is procedure i.e., program, subroutine, function, paragraph. Although the preferred embodiment of eX-Model 243 discussed in this detailed description is a three dimensional model, those skilled in the art could readily adapt the model of the present invention to additional dimensions. Two of the three dimensions on which the eX-Model is constructed have already been described, namely the k,b coordinates of the eX-Graph 262.

The third dimension of the constructed eX-Model comprises a structure, that is populated with attributes of process addresses. The process address of an element is the position (k,b) of that element within the reduced flow chart, or alternatively within the eX-Graph 262 of the target process 250. Table 4 is an example of the implementation of the structure constituting the third dimension of eX-Model for the elements of the eX-Object set *, C or +.

AMENDED SHEET

This Sheet Has Been Replaced



PCT/US 94/05182

IPEA/US 03 DEC 1994

-58-

Table 4									
Attributes									
*	ind	re-	loc.	BA	tar.	rep.	eX-S	RBA	
C		wind	time		rec#	rec#	rec#		
+		ID							

5 Different attributes are assigned to
different process addresses, depending upon the type of
process element (i.e., A,L,D,W,E) that is positioned at
this process address. Referring to Table 4, one member
of the structure is populated with the type of process
element. Another member of the structure contains ind,
10 which is the index of the corresponding process element
in the eX-Frame and eX-Graph.

Each element of the eX-Model corresponds to
an executable statement and contains three distinct
attributes which establish a relationship between the
15 target process 250, the eX-Model 243, the eX-Repository
246 and eX-Source 247. Those attributes providing this
relationship are: 1) a record number of the
corresponding statement within the target process; 2) a
record number of the corresponding statement within the
20 eX-Repository; and 3) a record number of the
corresponding statement within the eX-Source.
Referring once again to Table 5, these three attributes
are present in the structure members.

25 For each executable process element, the
eX-Model keeps an attribute of local time. Local time
is an integer value of the occurrence of a statement
within a target process. For example, the first time
that a statement is executed during a program

This Sheet Has Been Replaced



AMENDED SHEET

PCT/IS 94/05182

IPEA/US 09 DEC 1994

-59-

5 execution, the local time would be 1. The second time
that the same statement is executed during a program
execution, the local time would be defined as 2. The
local time attribute is present as a structure member
of the sample eX-Model structure shown in table 4.

10 The binary address attribute is assigned to
each process address of the eX-Model. In Table 4, the
binary address is shown as a structure member.

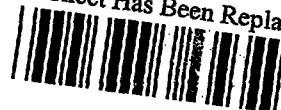
15 Structures representing rewind terminals
*,+,C also possess an attributes of Rewind ID and
Rewind Binary Address that represent the Process
Address and BA of the position in the eX-Graph, that
will be active after that statement execution.

20 The ID value is an alternative representation
of (k,b) coordinate pair. Each process element
represented by a (k,b) coordinate pair in the eX-Graph
could be alternatively represented by one integer value
ID, where the integer is the numerical order of the
element as it is encountered during a traversal of the
two dimensional table on which the eX-Graph is built in
order of traversing columns, left to right. The ID of
25 any particular element with process coordinates (k,b)
could be defined by the following formula:

$$ID = ((b-1) * k_{max}) + k$$

30 where k_{max} is the maximum value of the k
dimension represented on the eX-Graph.

This Sheet Has Been Replaced



AMENDED SHEET

-60-

From the other side knowing the ID of element s (k,b) and knowing the $k_{\max}$ of the model, one could define (k,b) by the following formulas:

$$\begin{aligned} b &= (ID - 1) / k_{\max} + 1 \\ k &= ID - (b-1) * k_{\max} \end{aligned}$$

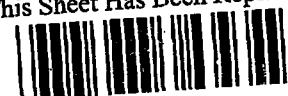
Referring to eX-Graph shown in FIG. 5(A), graph element *1 (k,b coordinate 10,1) has an ID of 10. Element L6 (k,b coordinate 8,5) has an ID of 76.

Table 5 shows the representative eX-Model structure for a process element represented by the following eX-Object set members, R and = .

Table 5					
Attributes					
R	ind	loc.	BA	tar.	eXS.
=		time		rec#	rec#

Exit terminals, when represented in the eX-Model, and in Table 5 do not contain the following attributes of rewind terminals; Rep.rec#. Rewind ID and Rewind BA.

Process addresses containing process elements represented by the eX-Objects A,L,D,E,W,X and S each have an ACB Pointer attribute. The ACB Pointer attribute is a pointer to a corresponding Analyzed Calculation Base construction that contains information on the history of approved and uncertain execution



-61-

events of the current process elements in relation to the knowledge of their effects correctness.

5 In addition, structures representing L and D
elements have an additional attribute which represents
the increment in the b coordinate when traversed
through their positive, or high potential solution.
This corresponds to the shift potential previously
described with respect to the eX-Graph. For example,
10 element L1 (k,b position 2,1) shown in FIG. 5(A) would
have a shift potential of 6 recorded in its shift
potential attribute field.

15 The Analyzed Calculation Base (ACB) 273 is a
collection of structures, each corresponding to a
different elements (statements) of the target process
250. The information recorded within the ACB indicates
the knowledge of correctness or its uncertainty of the
effects of the different process address execution
20 events at different times within the process.

FIG. 19 shows a number of ACB structures with
each ACB construction corresponding to an executable
process address of the target process 250. Every
25 member or element of an individual ACB structure
represents an event of a target process element
execution.

30 In a present embodiment, the ACB pointer
element of the eX-Model previously described points to
a binary file that is divided into two parts. The
first part of the ACB structure is a control area that
contains two distinct fields; Next Parameter Word (NPW)
and Uncertainty Displacement Word (UDW). The NPW is
35 only used for elements that have parameters, i.e., E

This Sheet Has Been Replaced



AMENDED SHEET

POT/12 94/05182

IPEA/US 09 DEC 1994

-62-

and X. Referring once again to FIG. 19, an ACB structure for element A1 is shown as 1900. This ACB structure contains three fields, 1901, 1902 and 1903. The NPW is shown as 1901 and the UDW is shown as 1902.

5 The second part of the ACB structure, shown in FIG. 19 as 1903, is the information area which stores knowledge state bits. Each execution event of the element that is represented by this ACB structure is represented by a bit in the information area 1903. 10 The position of a particular bit within the information area corresponds to the local time of the event. That is to say, the 20th bit of the information area 1903 corresponds to a local time of 20, or the 20th time 15 that particular element was executed. The state of the binary digits (0 or 1) comprising the information area correspond to the current knowledge of the events effects correctness, where 1 represents a correct effect of an event and zero corresponds to an uncertain 20 effect of an event. When an event is represented by zero in the ACB structure existing at some process time, it means that at that process time, the effect of the examined event is not known to be correct.

25 Those skilled in the art can readily appreciate that the monitoring of a target process 250 can generate enormous volumes of data even when recorded in a structure as simple as the ACB. While many forms of compression could be used, the present 30 implementation UDW 1902 stores the relative displacement of the first bit that is in a zero state. For example, UDW 1902 contains the value 2000. This represents that the first bit position stored in information area 1903 is for a local time value of 2001 35 and that all 2000 previous events of that element

This Sheet Has Been Replaced



AMENDED FIGS

-63-

execution were approved or disregarded as a blank events, i.e., events which did not produce any results which were used subsequently.

5 Each parameter of an input statement (E) is represented by a ACB structure. Referring once again to FIG. 19, two parameters of an input statement are shown represented by two ACB structures 1905, and 1906. The NPW of 1905 points to structure 1906.

10 Similarly, each parameter of a call statement (X) is represented by an ACB structure. FIG. 19 shows three ACB structures 1907, 1908 and 1909 which correspond to three parameters of the call statement. The NPW of 1907 points to structure 1908. The NPW of 1908 points to 1909. The NPW of 1909 is empty since there is no next parameter in this CALL() statement.

20 Building the eX-Repository

25 The eX-Repository 246 of target process 250 is built at the same time as the eX-Graph 262 or the eX-Frame 261. Each element of the main branch of the target process that corresponds to an executable statement, i.e., represented by eX-Object set members A, L, D, E, W, X and S, is stored in the eX-Repository 246.

30 The size of the eX-Repository 246 is less than the corresponding target process source code. This is because the eX-Repository 246 does not contain any process control statements, therefore the eX-Repository does not contain any unconditional branches, stop, return, then, else, endif, etc. The control structures are instead, preserved by the

35



PCTAID 94/05182
IPEA/US 09 DEC 1994

-64-

eX-Builder 259 in the eX-Model 243. To demonstrate how particular statements of target process 250 are represented in the eX-Repository, consider the following examples.

If target process source 250 contained the statement:

```
if (d < 10) go to label
```

The eX-Repository 246 would contain the statement:

```
if (d < 10)
```

Alternatively, if target process source 250 contained the following statements:

```
if (a .gt. b) then
  c = d
else
  e = f(g)
endif
return
```

The eX-Repository 246 would contain:

```
if (a .gt. b)
  c = d
  e = f(g)
```

Each statement of the eX-Repository is referenced by the repository record number (rep rec.#) attribute of the corresponding process address within the eX-Model. For example if the eX-Repository is kept as a Relative Record Data Set (RRDS) then the

AMENDED

This Sheet Has Been Replaced



PCT/10 94/05182

IPEA/US 09 DEC 1994

-65-

repository record attribute of the process address within the eX-Model must contain the relative record number of the corresponding statement within eX-Repository 246.

5

Building the eX-Source Code

Subsequent to the construction of the eX-Graph 262, the eX-Model 243 and the eX-Repository 246, eX-Source 247 is generated by the eX-Source generator 249. This step is only done if the eX-Analyzer is implemented in an embodiment which uses traditional object code.

The eX-Source 247 provides a synchronization mechanism between the target process 250 and the eX-Analytical Engine 244. This synchronization mechanism is implemented through the use of synchronization signals, which are inserted within the eX-Source 247, by the eX-Source Generator 249. In addition to the inserted synchronization signals, the eX-Source 247 may differ from the target process 250 in that the eX-Source 247 will not contain any unreachable or "dead" code.

25

The eX-Source code 247 is built by the eX-Source Generator 249 from two predecessor structures, the eX-Repository 246 and the eX-Model 243. Since the eX-Source is built from the eX-Model and the eX-Repository which were built from the target process 250, there exists a functional equivalence between the target process 250, the eX-Model 243 and the eX-Source 247.

30

AMENDED SHEET

This Sheet Has Been Replaced



PCT/US 94/05132
IPEA/US 09 DEC 1994

-66-

During the creation of the eX-Source 247 by
eX-Source generator 249, the following steps are
performed by the eX-Source generator: 1) the contents
of each executable statement which comprises the
eX-Source is taken from the eX-Repository 246; 2) the
control statements contained within the target process
250 which are now represented within the eX-Model 243
are extracted from the eX-Model; and 3) the target
process 250 - eX-Model 243 synchronization signals are
inserted into the eX-Source 247.

Those skilled in the art can readily
appreciate that the synchronization signals can be
implemented through a variety of means. The
synchronization signals inserted into the eX-Source may
consist of CALL() or WRITE() statements, which should
be immediately familiar to those skilled in the art.
The inserted synchronization signals provides the
eX-Interface 263 with three parameters. The three
parameters are defined as (k,b,m) where k,b are the k,b
coordinates that define the process address, and m
defines the model of the target process.

Alternatively, this eX-Interface 260 could be
implemented with only two parameters. For example,
instead of the (k,b,m) parameters one could pass only
(ID,m) where ID is the unique identification of the
element's process address which is calculated from the
k,b coordinate as previously described:

The synchronization signal is inserted into
the eX-Source 247 by the eX-Source Generator 249 during
the construction of the eX-Source for each statement
representing a process terminal. As an example,
consider the reduced flow chart shown in FIG. 4 which

This Sheet Has Been Replaced



AMENDED SHEET

-67-

represents target process 250. The flow chart shows a terminal 411 at k,b coordinate 10,1 which represents a FORTRAN GOTO statement contained within target process 250.

When the eX-Source Generator 249 constructs the eX-Source for terminal 411 the following program statements are inserted within the program code:

10 call XPDI(10,1,1)
 GOTO 'LABEL a'

15 The call XPDI() statement is made with three parameters as identified above. By way of an additional example, consider terminal 431 shown in FIG. 4. When the eX-Source Generator constructs eX-Source for terminal 431, the following program statements are inserted within the program code:

20 call XPDI(9,6,1)
 GOTO 'LABEL e'

25 When CPU 241 executes either of the above synchronization signals, the three parameters are passed to the eX-Interface 263. When the eX-Interface receives the signal, it records the process address or ID of the calling terminal and the model identifier (m) within the system execution history 253.

30 While the present embodiment passes the three above parameters with the synchronization signal, an alternative embodiment would pass only two parameters with the synchronization signal. Since the element ID characteristic previously described uniquely identifies
35 a process element and the ID is derived from the k,b

AMENDED SHEET

This Sheet Has Been Replaced



PCT/US 94/051
IPEA/US 09 DEC 19'

-68-

coordinates of the process element, the process ID could be passed as a parameter to the synchronization signal instead of the k,b coordinate pair.

5 When implemented in this manner, the sequence of executed process terminals is sufficient information to allow the analyzer to recreate the target process 250 behavior for any given model of a target process.

10 This is because, within the model, there is only one pass between the START and a specific terminal or between an entry of a Rewind ID of a Rewind ID of the previous terminal and the next terminal. The Rewind ID has been stored as an attribute of each
15 terminal represented within the eX-Model 243.

 The terminal which sent the synchronization signal is then located by the eX-Interface in the eX-Model through its k,b address.

20 When eX-Interface 263 working in the traditional object code embodiment of Knowledge Induction Machine is signaled and the parameters associated with the signal are sent, the eX-Interface
25 invokes eX-Analytical Engine 244 which retraces on the eX-Model a part of the pass executed within the target process 250. The part retraced is from the Rewind ID of the previously executed terminal (or from the START, if the currently executed terminal is the first since
30 the process was entered) to the current terminal. Each pass through a process address represents an execution event for that process address and the local time of that address is incremented by one.

AMENDED SHEET

This Sheet Has Been Replaced


PCT/US 94/05182

IPEA/US 09 DEC 1994

-69-

By way of example, consider the execution where the following three CALL() instructions were made to the eX-Interface during the course of an analysis.

CALL XPDI(10,1,1)
CALL XPDI(10,1,1)
CALL XPDI(9,5,1)

With reference to FIG. 5(A), these signals correspond to the execution of terminals mb1, mb1, and mb5 respectively. From this information, the analyzer determines that the target process 250 was executed through the path identified by the following elements in this order Entry, L1, /7, A1, /1, L2, A2, L3, X1, *1, L2, A2, L3, X1, *1, L2, A7, L6, R1 (Exit).

To represent repetition of the same terminal, as is often the situation with control loops structures comprising DO, FOR, While, etc., a negative number notation is used. For example, a notation 27 -5 42 would mean that a loop was passed through 6 times (reached the terminal with IDT=27) before exiting from the loop.

As may be appreciated by those skilled in the art, the synchronization statements could have been invoked after every element or after every exit of a logic condition. Our preferred embodiment minimizes the number of signals utilized to effectively represent the target process.

These above examples demonstrate that recording the target process execution history by recording the sequence of the executed process terminals is sufficient and the absolute minimum

This Sheet Has Been Replaced



Amended to 547-07

PCT/US 94/0518

IPEA/US 09 DEC 1994

-70-

recording required to characterize the target process
250 dynamics.

5 In addition to the synchronization signal
previously described that synchronize the execution
(retracing) of eX-Model necessary in this embodiment
with the execution of the target process, two other
types of synchronization signals are inserted into the
eX-Source by the eX-Builder. Those other two signals
10 allow the construction of the incremented system output
process synchronized map and optionally additional
synchronization between the target process and eX-Model
execution processes.
Those two types signals are inserted immediately
15 following system output statements (W) and immediately
before system input statements (E). Synchronization of
the system output does not invoke the eX-Analytical
Engine. Process synchronization at the system input is
optional. Process synchronization at the system input,
20 when performed, invokes the eX-Analytical Engine and is
performed in our preferred embodiment only because it
is a convenient place to catch up with the target
process by the eX-Analytical Engine since the system
input interrupts the target process anyway by the
25 operator or user "think time".

With reference to FIG. 21A and FIG. 21B the
entry for the Incremental System Output Process
Synchronized (ISOPS) map corresponding to the system
30 output 2151 will be performed by the call to the eX-
Interface XPDI 2152. Process synchronization at the
system input 2154 is performed by 2153, process
synchronization at the terminal 2161 is performed by
2160, at the terminal 2159 by 2158 and at the terminal
35 2157 by 2156. Entry for the System Input Process

AMENDED SHEET

This Sheet Has Been Replaced



-71-

Synchronized (SIPS) map corresponding to the system input 2154 will be recorded by 2155.

Incremented System Output Process Synchronized Map

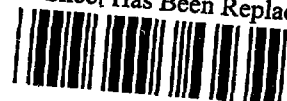
The incremented system output process synchronized map (ISOPS) provides a mechanism to associate an output of a target process with the process events which produced that output. The ISOPS map is constructed out of two elements, Incremental Image Monitor (IIM) 267 and ISOPS parameters 264 that in turn in its present embodiment consists of five fields, each containing an entry which defines a characteristic of the map.

In reference to FIG. 27 entry 1 defines a W-event as a set of (k,b,p,m) process parameters. (k,b) are the synchronization parameters associated with the signal sent to eX-Interface 263. Parameter (p) is derived from the eX-Model 243 of the target process 250. It is the current time attribute of the process element which corresponds to the (k,b) coordinate pair. The time attribute is obtained from the local time attribute kept by the eX-Model incremented by one since this W event was not yet executed (retraced) in the eX-Model in the traditional object code embodiment. In the eX-Object code embodiment, where the eX-Model drives the execution, the local time attribute will already be up to date and the retracing is not performed. Parameter (m), that is also present within the synchronization signal, is eX-Model identifier.

Entry 2 defines the screen line and column position (sl,sc) at the beginning of each system output

AMENDED SHEET

This Sheet Has Been Replaced



-72-

signal. The {sl,sc} entry is {1,1} for the very first
W-event entry. When eX-Interface 263 receives a
synchronization signal sent by a W-event, eX-Interface
5 obtains the current screen position that is in effect
after the execution of the W-event. The screen
position so obtained is stored and later used as
(sl,sc) entry for a next W-event.

10 Entry 3 defines the smallest rectangular area
of the system output screen 258 (l1,c1,l2,c2) which
represents the area of the screen changed by the
W-event execution. This rectangular area is determined
by comparing the previous screen image 255 that is
saved at the process time after the previous W-event,
15 and the current system output screen buffer 265.

Entry 5 contains the contents of the sysout
screen buffer 254 that corresponds to the area defined
in Entry 3. Entry 5 is contained in a variable length
20 field and is stored in the present embodiment as a
separate file, the Incremental Image Monitor File 267.

Entry 4 is an integer which defines the
displacement of Entry 5 from the beginning of IIM file
25 267.

Building System Input Process Synchronized Map

30 A system input process synchronized map
(SIPS) map is built by the eX-Interface during the
course of an execution of the target process to be
later used for analysis of a specific input to the
system made at some process time. The SIPS map is
stored in system input history file 252. The entry made

AMENDED SHEET

This Sheet Has Been Replaced


PCT/US 94/0518;

IPEA/US 09 DEC 1994

-73-

to this file consists of two records. The first record is the set of parameters {k,b,p,m} which describe the process time {k,b,p} and the model {m} corresponding to the current input string. The second record is the actual input string.

The eX-Interface 263 creates the entry made to the system input history file 252 from the synchronization parameters {k,b,p,m} associated with a signal made to eX-Interface 263 and the system input buffer .

Alternatively, from taking the system input values for the SIPS map from the system input buffer, these values could be taken from some other buffer where they were temporarily stored. They could be placed there by the statement inserted immediately after the system input statement and constructed from this input statement by simply changing READ to WRITE directed to the buffer. The remainder of the statement, i.e., its parameters and format, remain unchanged. As shown in FIG. 21(B), the READ(*,002)CH is followed by a WRITE(M,002)CH.

FIG. 21(B) shows an example of {ID,m} process synchronization call 2153 that is made before an input from the terminal. The ID of this synchronization signal (21) is translated by eX-Interface to k,b of 5,3 which corresponds to the process address containing the element E1 (2154 in FIG. 20(A)). If done as shown in the example, the system input is recorded by this means of statement 2155 to the input monitor buffer file 254. There could be several system input strings recorded in the system input monitor buffer by several E elements before the next terminal is met. During the next

AMENDED SHEET

This Sheet Has Been Replaced



-74-

process synchronization signal performed at the process terminal (which in our example could be performed at = 2157 or C1 2159 in FIG. 20(A)), the system input string entries are taken from the system input monitor buffer 254 and recorded in the system input history file 252. After that time, the system input monitor buffer could be reused.

Alternatively, if the keyboard buffer is used to create an entry in the system input history file (ISOPS file) the presence of the system input monitor buffer is not needed as well as inserting statements 2155 (FIG. 21(B)) within the eX-Source. Instead, the signals which would read the keyboard buffer and make an entry to the SIPS map would be needed after each system input statement 2154 in FIG. 21(B).

Operational Examples of Knowledge Induction Machine (KIM) Analyzer

Delayed Sequential execution of the eX-Model is implemented by one or more CPU(s) which execute the target process and the eX-Analytical Engine. Synchronization between the execution of the target process and the eX-Analytical engine takes place through synchronization signals that are placed within the target process by the eX-Source generator. When the CPU executes such a synchronization signal, program control is transferred to the eX-Analytical Engine. The target process execution is suspended for the duration of the signal. Target process execution resumes upon return from the signal.

Delayed parallel execution of eX-Model is implemented through the use of multiple CPUs, where one

ALTERNATE SHEET

This Sheet Has Been Replaced


-75-

of the multiple CPUs executes the target process and another of the multiple CPUs executes the eX-Analytical Engine at the same time. Synchronization between the multiple CPUs takes place through any one of a variety of methods known in the art. The present invention only requires communication of the parameters (k,b,m) between CPU(s) working on the target process and CPU(s) working on the eX-Analytical Engine analyzing the target process.

Both, methods of execution of the eX-Model are implemented through the use synchronization signals from process terminals (idt,m) and optionally, system input (ide,m). The synchronization signals (idt,m) and optionally (ide,m), bring the eX-Model "up to date" with the target process. The synchronization signal (idw,m) is designed to bring ISOPS map "up to date" with the target process and Process Time within eX-Model. The SIPS map is created either: 1) at the terminal synchronization points if the system input monitor buffer is used; or 2) by the synchronization signals performed after each system input statement.

When eX-Interface 263 receives a synchronization signal that is implemented in the form (id,m), the eX-Interface translates the (id) parameter of the corresponding model (m) into (k,b). The process address (k,b) will point to one of the following: a process terminal (idt); a system output signal (idw); or an input from the terminal (ide).

If the synchronization signal received by eX-Interface 263 originated from system output element (idw,m), eX-Interface 263 initiates the process previously described which updates ISOPS map.



PCTA/IS 94/05182

IPEA/US 09 DEC 1994

-76-

If eX-Interface 263 receives an {idt} synchronization signal, the {idt} parameters are written into system execution history file 253.

5 In the traditional object code embodiment, if the process synchronization signal received by eX-interface 263 is {idt,m} or in the alternative embodiment {ide,m}, eX-Interface 263 then initiates eX-Analytical Engine 244. Once initiated,
10 eX-Analytical Engine 244 will retrace, i.e. execute, the eX-Model 243 from the point of the model at which a previous synchronization signal {k,b} was sent or from the entry point of the eX-Model to the {k,b} of the current synchronization signal. As each element of the
15 eX-Model is traced, the time attribute {p} of that element is incremented by one, thereby keeping the eX-Model 243 time current.

20 When during the model execution in any of the embodiments, traditional object code embodiment, eX-Object Code embodiment or embodiment using interpreting of the eX-Repository, a process element is met that is a system output element {W}, eX-Analytical engine 244 initiates a knowledge induction process. Alternatively
25 the same Knowledge Induction Process could be initiated directly from the ISOPS map synchronization signal.

30 The eX-Repository entry corresponding to the current {W} element is analyzed and each parameter of the W element initiates a cycle of knowledge induction chains. Each of these chains traces back events to their ancestors, up to knowledge propagation terminals.

35 Tracing backwards is analogous to, but in a different direction from the earlier explained

APPENDED SHEET

This Sheet Has Been Replaced



PCT/US 94/05182

IPEA/US 09 DEC 1994

-77-

principle of tracing the target process forward. It is done from the present position of the synchronization signal within the target process eX-Model to the previous position of the synchronization signal within the target process eX-Model. It is possible within the present eX-Model construction which knows the rewind entry of each terminal.

The construction of terminated event definition space of a W element that is terminated by knowledge propagation terminals is called knowledge induction.

The terminated event definition space is built with reference to FIG. 25 or alternatively with the reference to FIG. 17. When the W element statement stored in the eX-Repository is examined all of its variable parameters are identified. In the example shown in FIG. 25, there are four parameters, 2511, 2512, 2513, and 2514, that have defined the value of some w {k,b,p} event, which value was saved on the ISOPS map. The branch 2501 represents these four parameters. Starting from the last one, 2514, the branch 2501 is retraced backwards, towards 2511. During the retracing, the eX-Model is traced against the direction of execution between two events. Originally the first event is the event {k,b,p}. The second event is the event that defined the value of one of its parameters. In this example, 2514. Further with respect to FIG. 25, the parameter 2514 was produced by the knowledge propagation event that is either a previously approved event or an event that had no ancestors, i.e., system input event or initial data manipulation event. Therefore, this parameter 2514 did not produce a new branch.

AMENDED SHEET

This Sheet Has Been Replaced



-78-

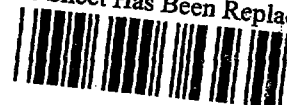
Retracing the current branch backwards,
parameter 2513 is encountered. The process event which
produced the value of this parameter is identified by
tracing the eX-Model against the direction of execution
5 from the event {k,b,p}. With reference to FIG. 25,
this value was produced prior to {k,b,p} event by some
statement s2 executed at the process time {k2,b2,p2}.
The examination of the eX-Repository entry of this
statement identified the presence of two variables,
10 2521 and 2522.

Tracing back the model from {k2,b2,p2} to the
events that defined the values of both parameters 2522
and 2521, the eX-Analytical Engine finds that in the
15 example of FIG. 25, the value of both of the parameters
2522 and 2521 prior to {k2,b2,p2} event, were defined
by knowledge propagation terminals previously
described. Therefore, new branches were not built for
those parameters 2522 and 2521 and therefore the
20 knowledge propagation processes were not activated in
those directions.

The branch 2502 therefore was the only branch
representing the event definition space of the
25 event {k2,b2,p2}. The eX-Analytical Engine is brought
back to the branch 2501 to continue the examination
backwards against the direction in which the branch was
built. The next parameters examined would then be
2512. In our example the value of this parameter 2512
30 was determined to be produced by some knowledge
propagation terminal event (KPT event) and therefore
the branch that would represent its event definition
space was not built.

AMENDED SHEET

This Sheet Has Been Replaced



PCT/US 94/05]

IPEA/US 09 DEC 1994

-79-

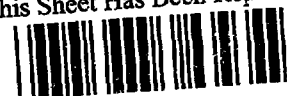
eX-Analytical Engine then retraces the current branch back to the parameter, 2511. In our example its value was also defined by KPT event.

5 The analysis of parameter 2511 concludes the analysis of the value of W-event (k,b,p) event definition space which in the example shown in FIG. 25, is defined by events that define the value of parameters located on the two branches 2501 and 2502.
10 If, during the construction of event definition space we pass an event which was previously uncertain, (in our example it was the event (k2,b2,p2)) its status is upgraded within the corresponding ACB structure.

15 Every time that an execution event status is upgraded from uncertain to correct by the knowledge induction process, a corresponding bit in the Analyzed Calculation Base structure 273 is updated from a value of 0 to a value of 1, thus increasing the knowledge of
20 the ACB structure and therefore the knowledge of target process eX-Model.

 A user of the eX-Analyzer views the system output that is recreated from the system result history
25 by the Incremental Image Extractor/Constructor. Operationally, the user positions a cursor, mouse or other pointing device at the system result. The system result so identified, corresponds to (k,b,p,m) parameters of the ISOPS map which defines the event of
30 W element execution. By analyzing the pointing device position within the area defined by entry 3 of the ISOPS map, the corresponding parameter of that W element (system output statement) is defined..

This Sheet Has Been Replaced



AMENDED SHEET

PCTUS

94/051

IPEA/US 09 DEC 1994

-80-

If the so identified system output parameter result is defined by the user as unexpected and therefore incorrect, the eX-Analytical Engine will start the knowledge deduction process.

5

Knowledge Deduction Process

Knowledge deduction is performed by a process identical to the knowledge induction process which builds the event definition space for the output events. The output events are traced back to their ancestors. This tracing is terminated in each direction by reaching knowledge deduction terminals which are the same as knowledge induction terminals, i.e., previously approved events or events that have no ancestors.

15

Unlike knowledge induction, there is no upgrading to the ACB structure knowledge during the knowledge deduction process. Instead, when a previously approved W-event is disapproved, the target process knowledge is dropped to the level, determined by the position of the disapproved W-owner event in the target process time as previously described.

20

25

There are two effects of the knowledge deduction process. First, there could be a loss of knowledge which is represented by changing the status of some process addresses from CPA to UPA. With reference to FIG. 16, when W owner of element k,b was defined, its position on the ISOPS map was recorded as one of the attributes of that k,b address. When some other position within the ISOPS map is identified as a failure, eX-Analyzer will change the status of all addresses previously defined as CPA to UPA if their

30

35

This Sheet Has Been Replaced



AMENDED 11/89

-81-

attributes representing W owner position within ISOPS map point to a place within the ISOPS map representing a later or equal time as compared with the reference failure. This works because the ISOPS map is built in chronological order in one direction and the process time of the W event chronology is preserved in the ISOPS map.

Second, the position of possible faults or position of required modifications is determined as an area of uncertainty within the definition space of the system output result. This area of uncertainty is defined as those process addresses of this results event definition space that lie outside the correct algorithm space.

As can be fully appreciated by those skilled in the art, a target process is fully represented by its eX-Graph and eX-Model. In that regard, once the eX-Graph, eX-Model and eX-Repository of a target process are generated by the eX-Analyzer, there is no longer a need for the eX-Source Generator, eX-Source, the Compiler/Linker. The Knowledge Induction Machine can execute and analyze the target process without these above-mentioned components and therefore without the necessity for synchronization signals.

There are two embodiments which could be done. One embodiment uses an interpreter 268 that interprets statements contained within eX-Repository 246 (shown in FIG. 18(C)). The order of the interpretation is controlled by eX-Driver which is tracing, i.e. executing, the eX-Model 243 and therefore is being controlled by the eX-Model.

This Sheet Has Been Replaced



-82-

Another embodiment uses eX-Object code which comprises the translated to machine code elements of the eX-Repository 246. The resulting code, therefore does not contain any control elements such as logic conditions, loop conditions, conditional or unconditional GOTO, RETURN, STOP, or loop terminals. The conditional statements are represented in eX-Object code only by the expressions to be evaluated. In this embodiment as well as the embodiment which uses an interpreter, the control of the target process is performed by the eX-Driver tracing (executing) the eX-Model. In the eX-Object code embodiment, the eX-Model becomes a part of the object code as it is traditionally understood since the eX-Model performs the control functions.

Since the eX-Model is controlling the process execution the eX-Model does not need to be informed of the active event in the target process. Therefore, the process synchronization signals are not needed. The only function of the eX-Interface in these two embodiments is to create the ISOPS and SIPS maps.

With reference to the eX-Object code embodiment shown in FIG. 18(A) and FIG. 18(B), two components of the eX-Object code are shown. Shown in FIG. 18(B) are a control component 1800 and an executable component 1801. The control component 1800 has a pointer 1802 to a segment 1804 within the executable component. Each segment within the executable component represents a statement of the eX-Repository that has been translated into object code.



PC/US 94/05182

IPEA/US 09 DEC 1994

-83-

The Executable Component Address (ECA) attribute 1802 is added to each executable element PA of the eX-Model. Each segment i.e., 1804, is followed by the Statement Separator Byte (SSB) 1803. eX-Driver
5 269 reads the value of ECA from eX-Model 243, reads the corresponding segment of object code 1803 and passes it to the cpu 241 which is located within 1806, that runs the executable component of eX-Object Code.

As one of different methods for monitoring
10 system input, the following is implemented: when module 1807 translates a statement representing input from the system terminal into Executable Component of eX-Object Code, it adds a next statement, that reverses the action of an input from a system terminal to output
15 to the input monitor buffer file 254. For example and with reference to FIG. 21(B), READ (* ,002)CH 2154 is followed by WRITE(M,002)CH 2155.

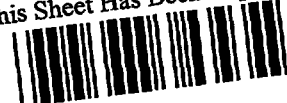
eX-Driver then has an option to activate or deactivate this second statement in order to control
20 monitoring of the system input while selecting the next code fragment to be passed to the module 1806.

Since the eX-Object code architecture consists of two components, control and control-less, this
25 architecture allows for easier modification of the target process. When the change needed is a control change, the control component is changed. When the change needed is a non-control component, a new entry is made in the executable component file and the ECA
30 reference in the control component is updated.

The resolution of the link between the control and non-control components through the ECA is done at a run time. Therefore compiling, recompiling

AMENDED SHEET

This Sheet Has Been Replaced



-84-

or interpreting is needed only for newly created or modified statements of the non-control component of eX-Object Code. It is even possible for the interpreter to be invoked only for the newly created or modified statements. This includes those situations where the newly created or modified statement is just a single statement.

The control component of eX-Object Code can be rearranged by insertion, deletion or modification to the eX-Model element corresponding to the newly created eX-Repository element. In some situations, when only the control structure needs to be rearranged, a modification to eX-Object Code can be made without any changes being made to the non-control component. This is done by rearranging the order of the elements within eX-Model.

Modification to the eX-Object code can be done in some situations by deactivating the control component and, therefore, bypassing the corresponding element of the non-control component. Further, if the resolution of the links between the Control and non-control eX-Object code components is performed at run time, linking or re-linking of a group of two or more target processes implemented by eX-Object Code is done by linking only the control components of the eX-Object Code.

Result Oriented Programming

The eX-Analyzer can address any event of any statement execution through the reference to the system results by using a system pointing device or cursor. Therefore, while building the event definition spaces

AMENDED SPEC

This Sheet Has Been Replaced



-85-

of the system results, there exists a way to automatically modify the target process code.

5 The event definition space of system result events or of the event definition space of an intermediate value calculation event could be reused to create new system results or conditions for new system results. The system results could be rearranged in the process time that corresponds to the system results.

10 This serves as the basis for the method of Result Oriented Programming (ROP) that is, a very high level language for forward development of a software process. In fact the level of this language grows with the forward development of the target process, since there are more and more events to analyze and more event definition spaces to reuse.

Program Stethoscope

20 Audio analysis, combined with traditional video analysis allows for better dynamic understanding of programs or system of programs. Execution of any target process can be seen as it is traditionally done by tracing the graph dynamically, but it also can be heard, if one generates sound pitches which correspond to statement positions within the eX-Graph. Generated this way, "music" would be defined by the process algorithm and is a function of the input data combination.

35 During the cognitive process of program understanding we are faced with a task of "playing computer". This is by definition a dynamic process. On the other hand a graph (picture) is by definition a

This Sheet Has Been Replaced



REMOVED PAGE

PCT/US 94/05182
IPEA/US 09 DEC 1994

-86-

static object. The difference between a graphical and sound representation of dynamic process may be understood by comparing two ways of recreating music:
5 1) by reading the music score (graphic representation);
and 2) by actually playing the music, or hearing it when looking at the score.

Audio effect during a program analysis can be achieved as follows. During parallel or post-life program emulation by applying next recorded terminal
10 address {k,b} to the eX-Model of the program we can trace the program execution on the eX-Graph of the program algorithm.

The audio effect is added by calling the
15 hardware internal function that allows the generation of an audio frequency and send it to the speaker. A head phone will be a more appropriate addition to the computer when analyzing a system by the described method, and will not affect other programmers.

20 The frequency can be calculated as a function of two parameters, {k,b}, where a single increment in {k} increments the frequency by one discrete step, and a single increment of {b} increments a frequency by a
25 larger step, for example by an octave. A branch back will accordingly decrement the frequency.

The duration of beep, corresponding to the tracing of one individual element of the eX-GRAPH can
30 be set as such that it would be recognized by the human ear. The effect of such audio execution emulation is such, that each process of program execution has its own "music", depending on that process algorithm structure and input data combination. In the case of
35 emulating a loop having of large number of iterations,

AMENDED SHEET

This Sheet Has Been Replaced


PCT/US IPEA/US 94/05182
09 DEC 1994

-87-

a code signal can be set to indicate such a loop in order to eliminate a necessity of tracing all the iterations. This code can be for example represented by longer duration signal, followed by different
5 duration signals, each representing let say 10 iterations of the loop. In addition, a graph depicting the process may be made and as each element is executed, the corresponding portion of the graph is highlighted and its characteristic tone is played.

10 When a particular output result is not being observed for a long time, this analysis can be activated to hear: 1) whether the system is dead; 2) whether the program in loop; 3) or is it waiting for an
15 input. When the last executed element was an input statement a special audio tone can be generated

Most stressful places of the program's algorithm can be heard as a most common frequency and
20 therefore easily identified and the difference in program behavior with different input data combinations can be heard.

A special tone can indicate a call to a child
25 function and a special tone can indicate a return from the call. The audio analysis can be limited to only those signals originally and expanded optionally to indicate process execution within one specific
30 function, if functionality within the function is to be studied.

The Audio analysis can be recorded during the process emulation (alive or post-life) and then
35 replayed without the actual mode emulation in both directions forward and backward.

AMENDED SHEET

This Sheet Has Been Replaced



94/05182
11/11/05 09 Dec 1992

-88-

It can be replayed synthesized with the
replay of the system result and thus give a filling of
the intensity of the processing necessary for those
results achievement;

5

It can be recorded and replayed for different
input data combinations for the same process in order
to analyze the difference in intensity of calculations
for those different input situations. Given that the
potential for the program fault is a function of the
number of the elementary processes involved, this
analysis will aid in selection simpler
(input-process-output) combination during system
evolution to a new specifications.

10

15

Although various embodiments which
incorporate the teachings of the present invention have
been shown and described in detail herein, those
skilled in the art can readily devise many other varied
embodiments that still incorporate these teachings.

20

This Sheet Has Been Replaced



AMENDED SHEET

PCT/US 94/05182

IPEA/US 09 DEC 1994

-89-

I claim:

1 1. A knowledge induction analysis method for
2 determining the location of faults contained within a
3 target process having control and non-control
4 components, wherein the method generates an object
5 program for a computer having a main memory in which
6 the analysis takes place, the method comprising the
7 steps of:

8 first analyzing said source process and
9 generating a model comprising said control components
10 of said source process;

11 second analyzing said source process and
12 generating a repository comprising said non-control
13 components of said source process;

14 executing said model and said repository
15 while recording an elementary knowledge about said
16 target process wherein said elementary knowledge
17 indicates the correctness of a process element
18 execution within the target process.

1 2. The method according to claim 1 further comprising
2 the steps of:

3 executing said model and said repository
4 while recording an elementary knowledge about said
5 target process wherein said elementary knowledge
6 indicates an uncertainty of a process element execution
7 within the target process.

1 3. The method according to claim 2 wherein said
2 elementary knowledge is recorded within an analyzed
3 calculation base, said analyzed calculation base
4 comprising a plurality of analyzed calculation words,



This Sheet Has Been Replaced



AMENDED SHEET

-90-

5 one analyzed calculation word for each parameter within
6 a statement of the target process, each analyzed
7 calculation word comprising a control area and an
8 information area; the information area comprises a
9 sequence of indicators which indicate a knowledge of
10 correctness of an effect of an event of a target
11 software process statement parameter at different
12 points in time, and the control area contains a value
13 which defines a first event indicator represented in
14 the information area which does not have a knowledge of
15 correctness of an effect of an event.

1 4. The method according to claim 1 wherein the first
2 analyzing step produces a bi-directed, terminated
3 eX-Graph having two perpendicular axis, with one axis
4 representing a zero-potential of a logic condition and
5 the other axis representing a one-potential of a logic
6 condition and an element recorded at a coordinate pair
7 of the two axis is represented by a binary address
8 which uniquely identifies that element.

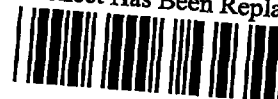
1 5. The method according to claim 1 further comprising
2 the steps of:
3 identifying a potential dependency between
4 elements of the target process wherein said potential
5 dependency is determined by ascertaining whether an
6 element of the target process is accessible from
7 another element of the target process.

1 6. The method according to claim 5 further comprising
2 the steps of:
3 identifying a data independent endless loop
4 contained within the target process.

AMENDED SHEET



This Sheet Has Been Replaced



PCT/US 94/05182

IPEA/US 09 DEC 1994

-91-

1 7. The method according to claim 3 further comprising
2 the steps of:
3 propagating the elementary knowledge recorded
4 about the correctness of an effect of an event of a
5 target software process to an ancestor event up to a
6 knowledge propagation terminal.

1 8. The method according to claim 1 further comprising
2 the steps of:
3 generating a plurality of audio tones during the
4 execution of the model where one tone of said plurality
5 of tones is generated during the execution of a process
6 element and another tone of said plurality of tones is
7 generated during the execution of another process
8 element thereby producing a continuous output of tones
9 as the model is executed through each individual
10 process element.

1 10. The method according to claim 1 wherein the
2 execution of the model is performed sequentially with
3 an execution of the target process.

1 11. The method according to claim 1 wherein the
2 execution of the model is performed in parallel with an
3 execution of the target process.

1 12. The method according to claim 5 further comprising
2 the steps of:
3 identifying a process element vision by comparing
4 a binary address of the element where the binary
5 address of an element is constructed from binary digits
6 representing logic solutions of the target process.

This Sheet Has Been Replaced



* C L M S *

PCT/US 94/01182

IPEA/US - 09 DEC 1994

-92-

1 13. A knowledge induction machine having a processor,
2 memory and input/output for analyzing target software
3 processes comprising:
4 a model comprising control components of the
5 target software process said model being contained in
6 the memory of the knowledge induction machine;
7 a repository comprising non-control components of
8 the target software process, said repository being
9 contained in the knowledge induction machine memory;
10 an analytical engine contained within the
11 knowledge induction machine memory;
12 a system execution history contained within the
13 knowledge induction machine memory;
14 a system input history comprising a system input
15 process synchronized map, said system input history
16 being contained within the knowledge induction machine
17 memory;
18 a system result history comprising an incremental
19 system output process map, said system result history
20 being contained within the knowledge induction machine
21 memory;
22 an ex-interface contained within the knowledge
23 induction machine memory, said ex-interface interfaces
24 the target process with the analytical engine and
25 constructs in memory the system execution history, the
26 system input history and the system result history;
27 an incremental image extractor/constructor
28 contained within the memory of the knowledge induction
29 machine, said incremental image extractor/constructor
30 compares a previous screen image with a present screen
31 image and extracts that portion of the present image

This Sheet Has Been Replaced



AMENDED SHEET

PCT/US 94/05182

IPEA/US 09 DEC 1994

-93-

32 which is different from the previous image, and places
33 the image portion so extracted, coupled with a
34 plurality of process synchronization parameters, into
35 the system result history; and

36 an analyzed calculation base contained within the
37 memory of the knowledge induction machine, said
38 analyzed calculation base comprising a plurality of
39 analyzed calculation words, one analyzed calculation
40 word for each parameter within a statement of the
41 target process, each analyzed calculation word
42 comprising a control area and an information area; the
43 information area comprises a sequence of indicators
44 which indicate a knowledge of correctness of an effect
45 of an event of a target software process statement
46 parameter at different points in time, and the control
47 area contains a value which defines a first event
48 indicator represented in the information area which
49 does not have a knowledge of correctness of an effect
50 of an event,

51
52 wherein the target software process sends a plurality
53 of synchronization signals to the ex-interface which in
54 turn builds the system execution history, system input
55 history and system result history and initiates the
56 analytical engine which in turn executes the model
57 while examining the repository and initiates a
58 knowledge induction process which constructs the
59 analyzed calculation base.

1 14. A knowledge induction machine having a processor,
2 memory and input/output for analyzing target software
3 processes comprising:

This Sheet Has Been Replaced



AMENDED SHEET

-94-

4 a model comprising control components of the
5 target software process said model being contained in
6 the memory of the knowledge induction machine;
7 a repository comprising non-control components of
8 the target software process, said repository being
9 contained in the knowledge induction machine memory;
10 an analytical engine contained within the
11 knowledge induction machine memory;
12 a system execution history contained within the
13 knowledge induction machine memory;
14 a system input history comprising a system input
15 process synchronized map, said system input history
16 being contained within the knowledge induction machine
17 memory;
18 a system result history comprising an incremental
19 system output process map, said system result history
20 being contained within the knowledge induction machine
21 memory;
22 an ex-interface contained within the knowledge
23 induction machine memory, said ex-interface interfaces
24 the target process with the analytical engine and
25 constructs in memory the system execution history, the
26 system input history and the system result history;
27 a language interpreter contained within the memory
28 of the knowledge induction machine;
29 a driver contained within the memory of the
30 knowledge induction machine;
31 an incremental image extractor/constructor
32 contained within the memory of the knowledge induction
33 machine, said incremental image extractor/constructor
34 compares a previous screen image with a present screen
35 image and extracts that portion of the present image

AMENDED



This Sheet Has Been Replaced



PCT/US 94/05182
IPEA/US 09 DEC 1994

-95-

36 which is different from the previous image, and places
37 the image portion so extracted, coupled with a
38 plurality of process synchronization parameters, into
39 the system result history; and

40 an analyzed calculation base contained within the
41 memory of the knowledge induction machine, said
42 analyzed calculation base comprising a plurality of
43 analyzed calculation words, one analyzed calculation
44 word for each parameter within a statement of the
45 target process, each analyzed calculation word
46 comprising a control area and an information area; the
47 information area comprises a sequence of indicators
48 which indicate a knowledge of correctness of an effect
49 of an event of a target software process statement
50 parameter at different points in time, and the control
51 area contains a value which defines a first event
52 indicator represented in the information area which
53 does not have a knowledge of correctness of an effect
54 of an event;

55
56 wherein the driver executes the model and extracts a
57 corresponding component from the repository and
58 presents the component to the language interpreter where
59 the component is interpreted, said driver further sends
60 a plurality of synchronization signals to the
61 ex-interface, said ex-interface in turn builds the
62 system execution history, system input history and
63 system result history and initiates the analytical
64 engine which examines the repository and initiates a
65 knowledge induction process which constructs the
66 analyzed calculation base.

This Sheet Has Been Replaced



AMENDED SHEET

PCT/US 94/05182

IPEA/US 09 DEC 1994

-96-

1 15. A knowledge induction machine having a processor,
2 memory and input/output for analyzing target software
3 processes comprising:

4 a model comprising control components of the
5 target software process said model being contained in
6 the memory of the knowledge induction machine;

7 a repository comprising non-control components of
8 the target software process, said repository being
9 contained in the knowledge induction machine memory;

10 an analytical engine contained within the
11 knowledge induction machine memory;

12 a system execution history contained within the
13 knowledge induction machine memory;

14 a system input history comprising a system input
15 process synchronized map, said system input history
16 being contained within the knowledge induction machine
17 memory;

18 a system result history comprising an incremental
19 system output process map, said system result history
20 being contained within the knowledge induction machine
21 memory;

22 an ex-interface contained within the knowledge
23 induction machine memory, said ex-interface interfaces
24 the target process with the analytical engine and
25 constructs in memory the system execution history, the
26 system input history and the system result history;

27 a driver contained within the memory of the
28 knowledge induction machine;

29 an ex-object code contained within the memory of
30 the knowledge induction machine;

31 an incremental image extractor/constructor
32 contained within the memory of the knowledge induction

This Sheet Has Been Replaced



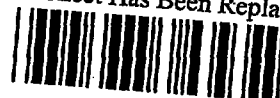
AMENDED SHEET

PCTIUS 94/NETR
IPEA/US 09 DEC 1994

-97-

33 machine, said incremental image extractor/constructor
34 compares a previous screen image with a present screen
35 image and extracts that portion of the present image
36 which is different from the previous image, and places
37 the image portion so extracted, coupled with a
38 plurality of process synchronization parameters, into
39 the system result history; and
40 an analyzed calculation base contained within the
41 memory of the knowledge induction machine, said
42 analyzed calculation base comprising a plurality of
43 analyzed calculation words, one analyzed calculation
44 word for each parameter within a statement of the
45 target process, each analyzed calculation word
46 comprising a control area and an information area; the
47 information area comprises a sequence of indicators
48 which indicate a knowledge of correctness of an effect
49 of an event of a target software process statement
50 parameter at different points in time, and the control
51 area contains a value which defines a first event
52 indicator represented in the information area which
53 does not have a knowledge of correctness of an effect
54 of an event;
55
56 wherein the ex-object code represents the repository
57 statements translated into a machine code of a target
58 machine and the driver executes the model and extracts
59 a corresponding component of the ex-object code and
60 presents the component so extracted to a computer
61 operating system for execution.

This Sheet Has Been Replaced



PCT/US 94/05182

IPEA/US 09 DEC 1994

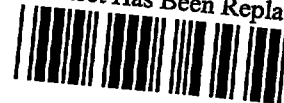
SUBSTITUTE ABSTRACT

-98-

Abstract of the Disclosure

A method and apparatus for the analysis of a computer process is disclosed. The method consists of the steps of analyzing a target process and constructing a model and repository which are
5 representative of the control and non-control components contained within the original target process respectively. The resultant model and repository are executed, and their execution is representative of the original target process. Structures created during the
10 execution of the model and repository assist a user to identify failures which occur during the execution, and then locate the fault(s) responsible for the failures as well as places within the target source where modifications due to new requirements are needed. An
15 alternative form of object code is proposed which consists of two components, control and non-control components. Previously described model representing the control component constantly increases the intelligence of the resulting object code in respect to
20 the target process being executed through the processes of propagation of elementary knowledge about the correctness or uncertainty of the target process elements execution.

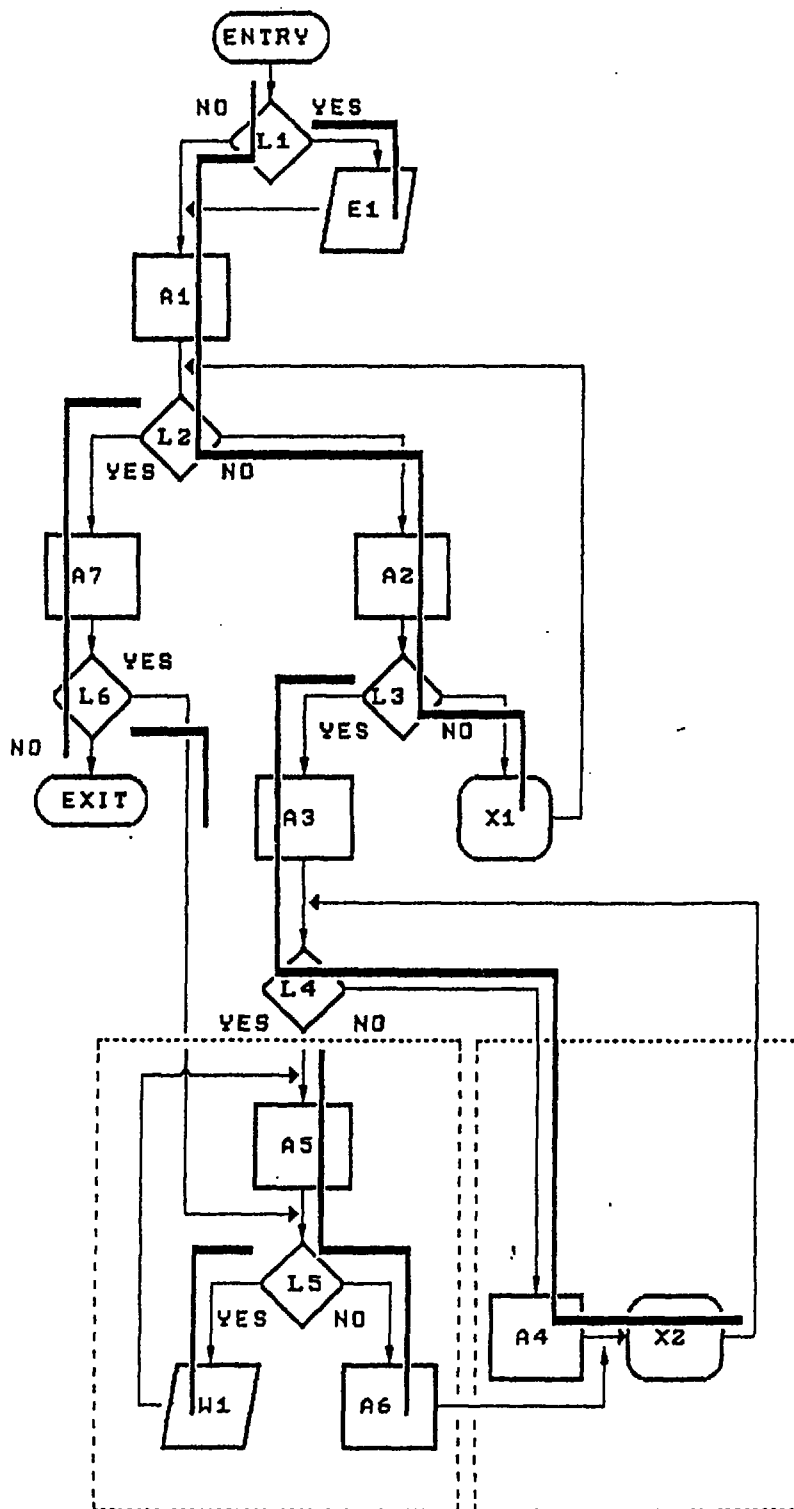
This Sheet Has Been Replaced



REMOVED SHEET

PCT/US 94/05182
IPEA/US 09 DEC 1994

FIGURE 1



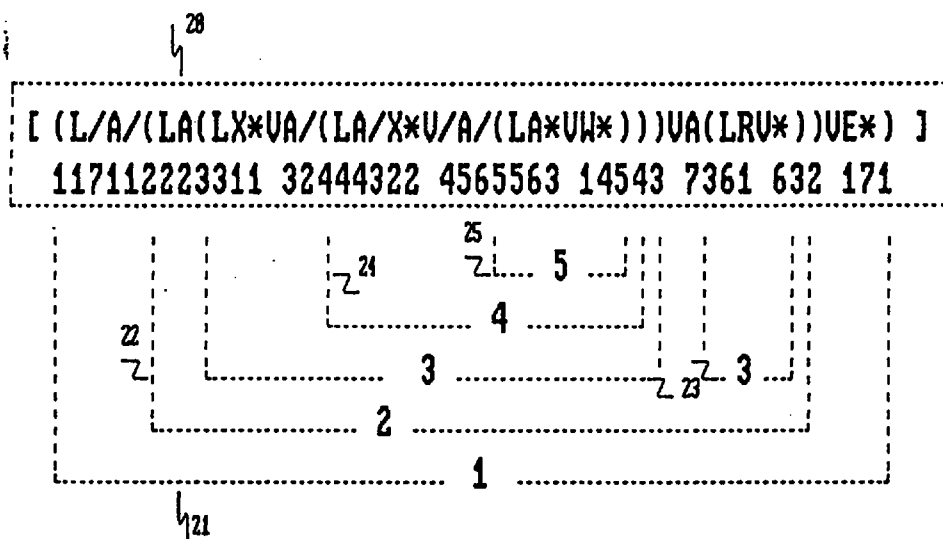
This Sheet Has Been Replaced



PCT/US 94/05182

IPEA/US 09 DEC 1994

FIGURE 2



This Sheet Has Been Replaced



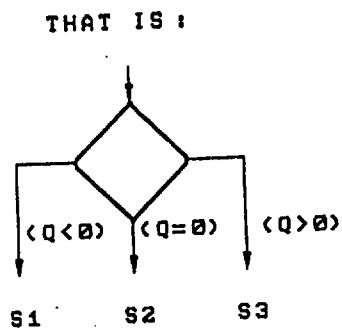
94/05102

IPEA/US 09 DEC 1994

FIGURE 3

EXAMPLE:

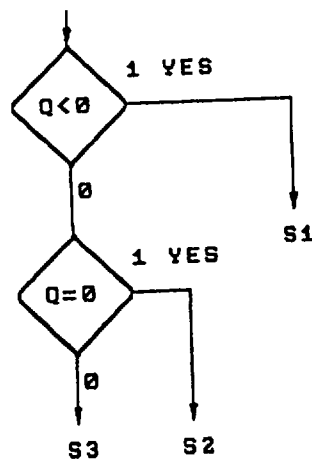
IF (Q) S1,S2,S3



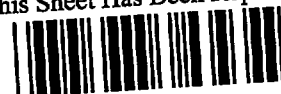
CAN BE REDUCED TO:

IF(Q.LT.0) GO TO S1
IF(Q.EQ.0) GO TO S2
GO TO S3

IS REDUCED TO :



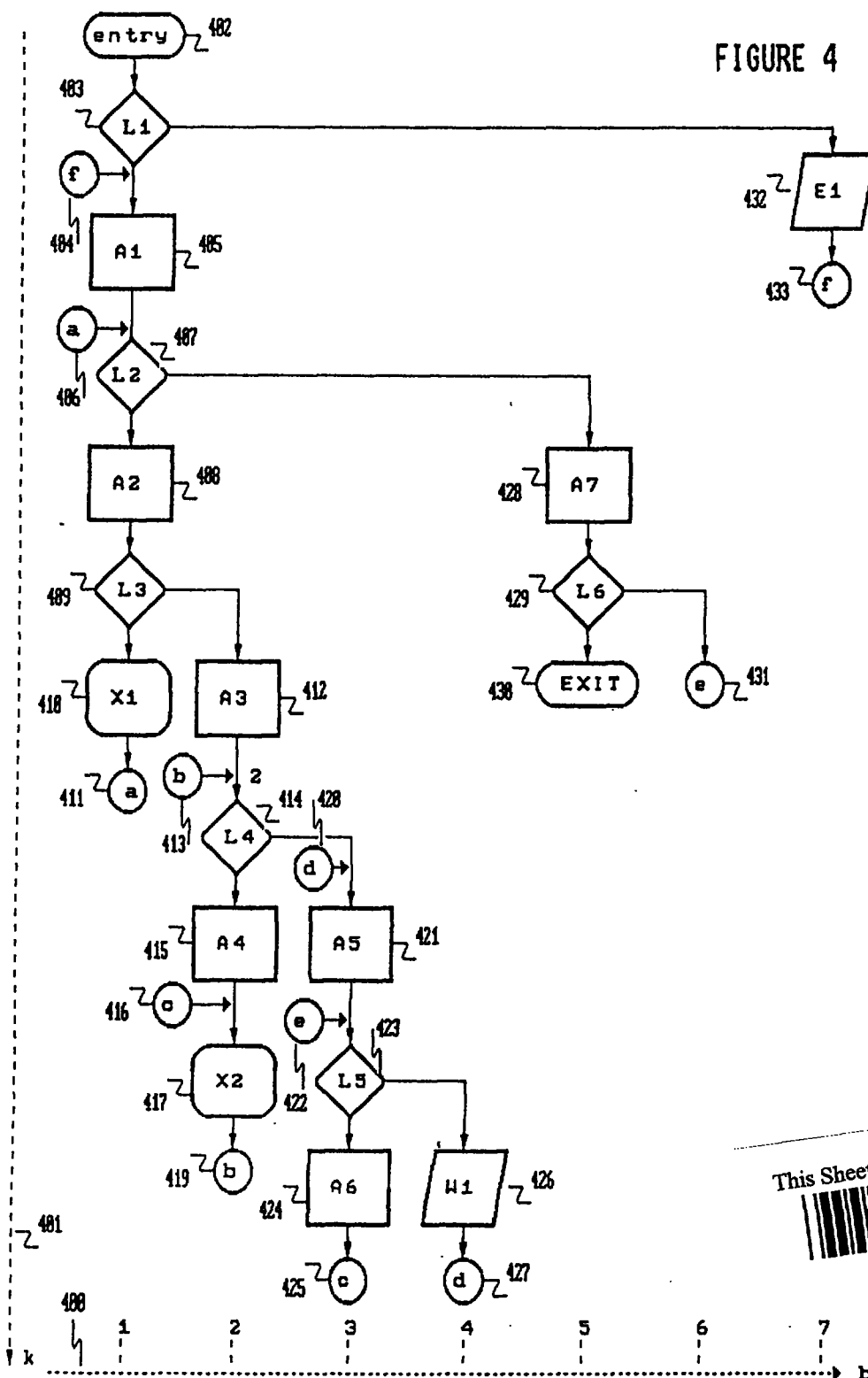
This Sheet Has Been Replaced



94/05182

IPEA/US 25 JUL 1994

FIGURE 4

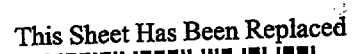


This Sheet Has Been Replaced



IPEA/US 03 DEC 1994

F.I GURE 5 8



IPEA/IIS 09 DEC 1994

```

graph TD
    subgraph Left_Flowchart
        In1(( )) --> P1[P1]
        P1 --> L1{L1}
        L1 -- YES --> P2[P2]
        L1 -- NO --> P3[P3]
        P2 --> Join1(( ))
        P3 --> Join1
        Join1 --> P4[P4]
        P4 --> Out1((1))
    end

    subgraph Right_Flowchart
        In2(( )) --> P1[P1]
        P1 --> L1{L1}
        L1 -- YES --> P2[P2]
        L1 -- NO --> P3[P3]
        P2 --> L2{L2}
        P3 --> L2
        L2 -- YES --> P2
        L2 -- NO --> P4[P4]
        P4 --> Out2((1))
    end
    
```

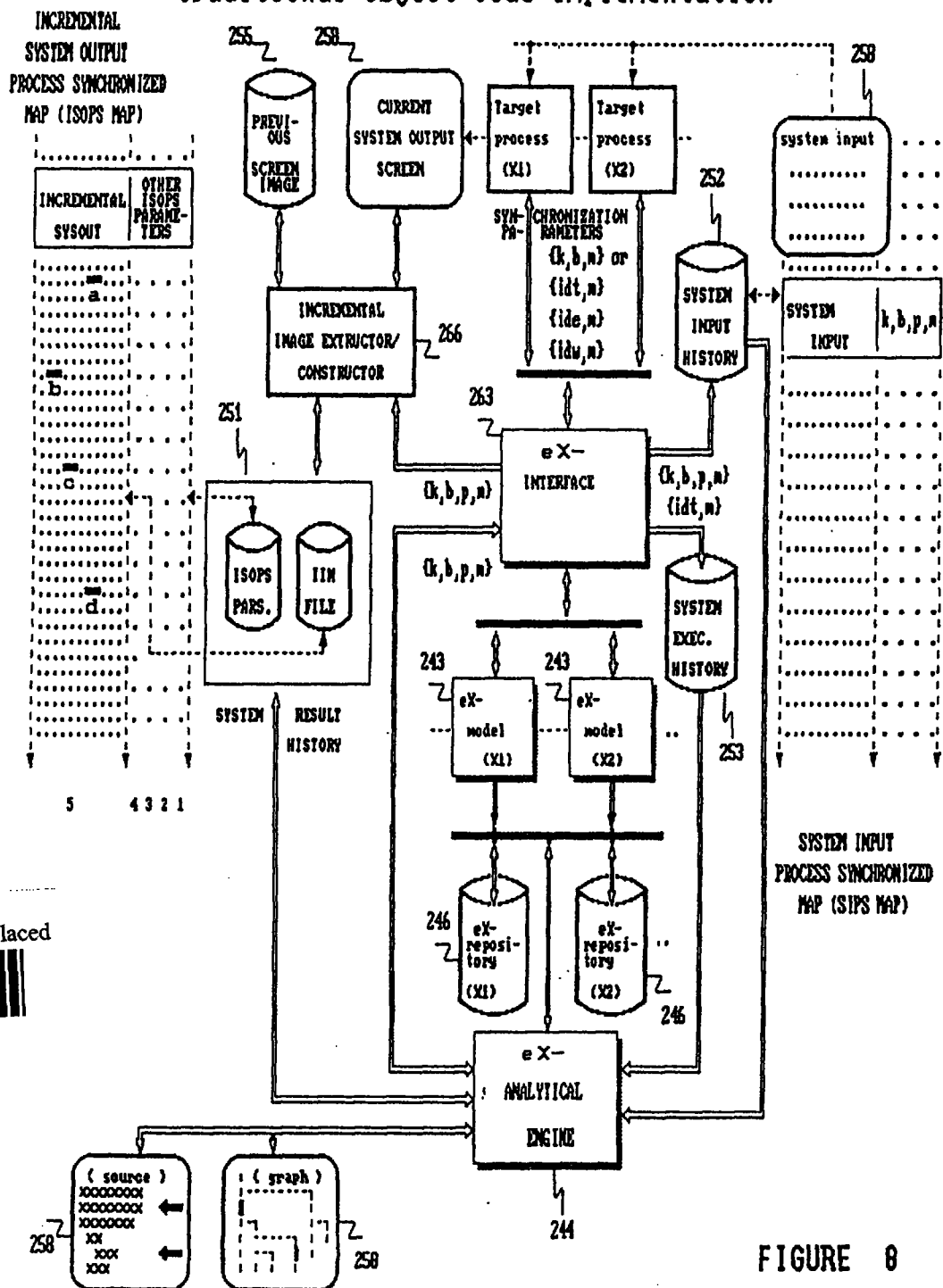
P1 (L1 P2 !2 P4 *1 V P3 *2)
1 1



PCTAIG 94/05182

IPEA/UIS 09 DEC 1994

Knowledge Induction Machine operation traditional object code implementation



This Sheet Has Been Replaced

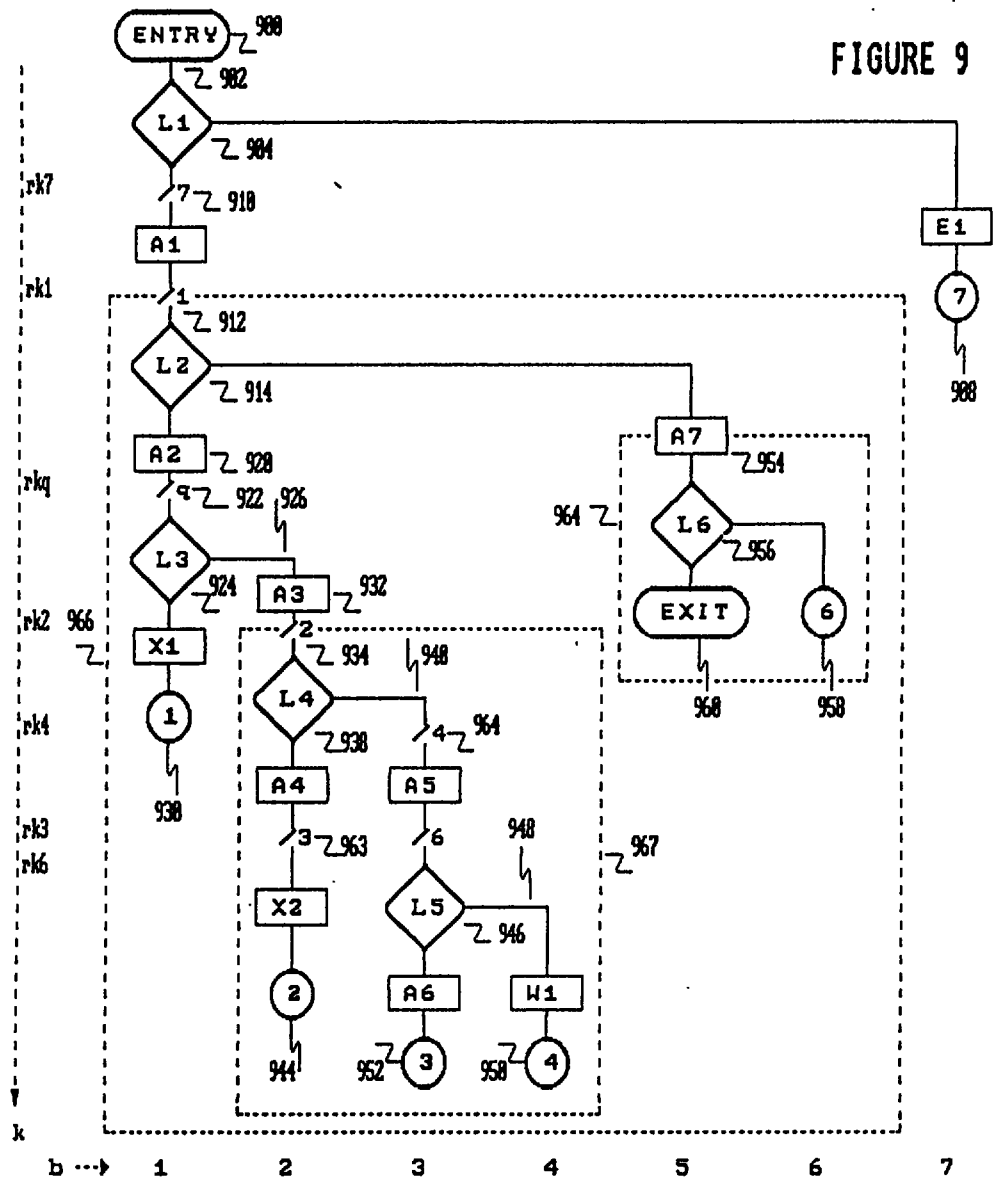


FIGURE 8

PCT/US 94/0518

IPEA/US 09 DEC 1994

FIGURE 9



EXIT:		orig.	0	0	0	0	1	0	0
ACC:		end	1	0	0	0	1	0	1
IND:		orig.	rk1	rk2	rk3	rk4	0	rk6	rk7
		rk end	rk1	rk2	rk2	rk2	0	rk2	rk7
PUP		orig.	10	1001	10010	10011	0	10011	10
		BA end	10	1001	1001	1001	0	1001	10
RBA			10	1001	10010	10011	0	10011	10
TBA			1000	10010	100110	100111	1010	1011	11

This Sheet Has Been Replaced



AMENDED SHEET

FIGURE 10 A

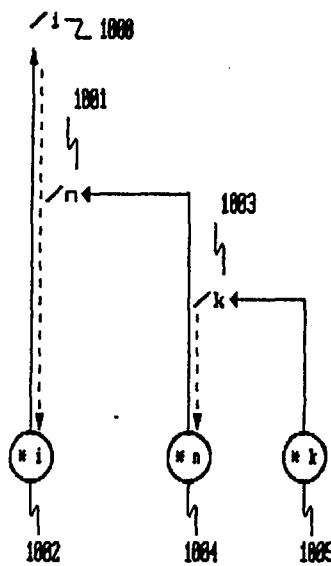
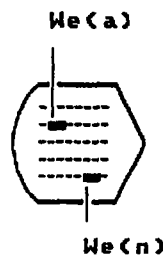
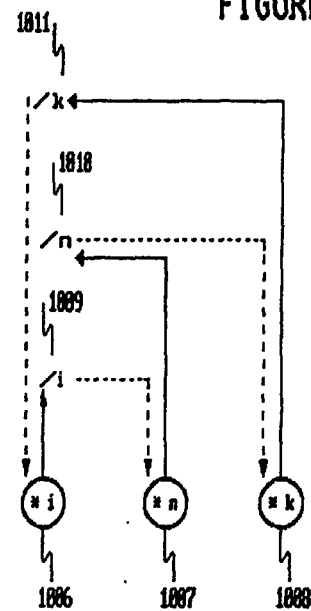
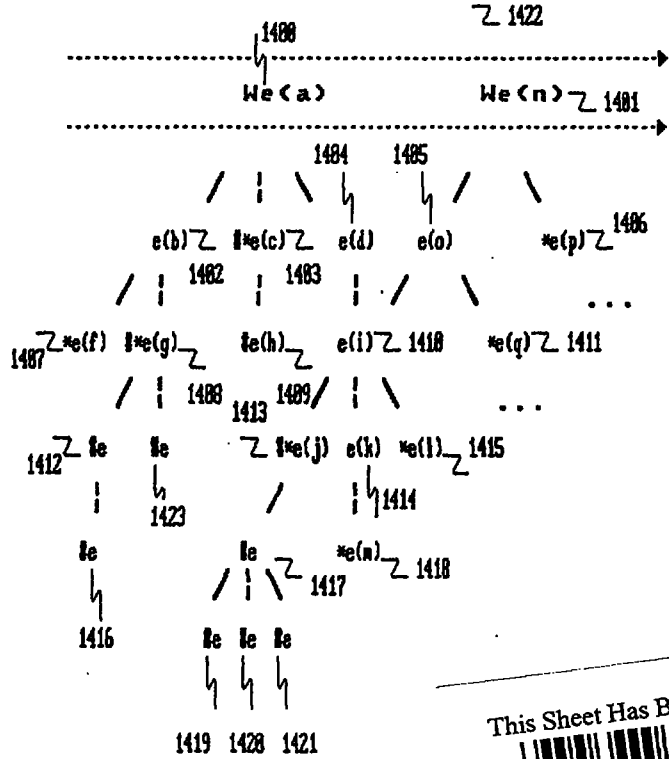


FIGURE 10 B



Process direction →

FIGURE 14



This Sheet Has Been Replaced

FIGURE 11

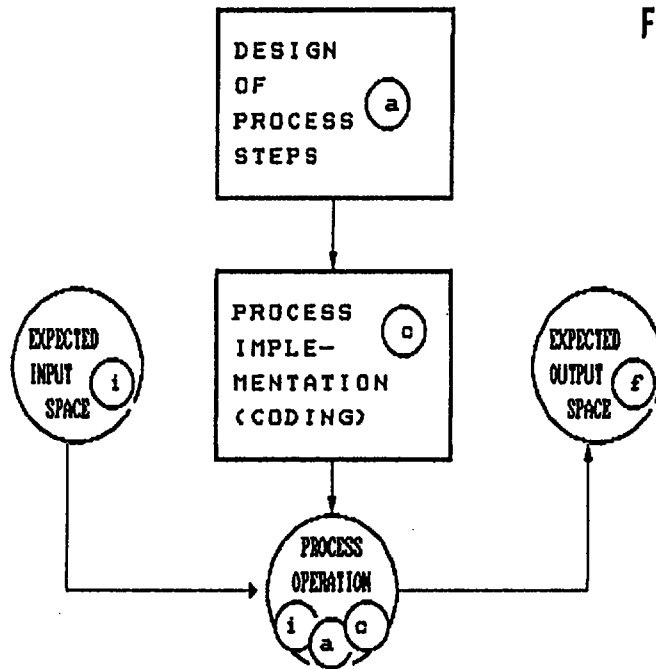
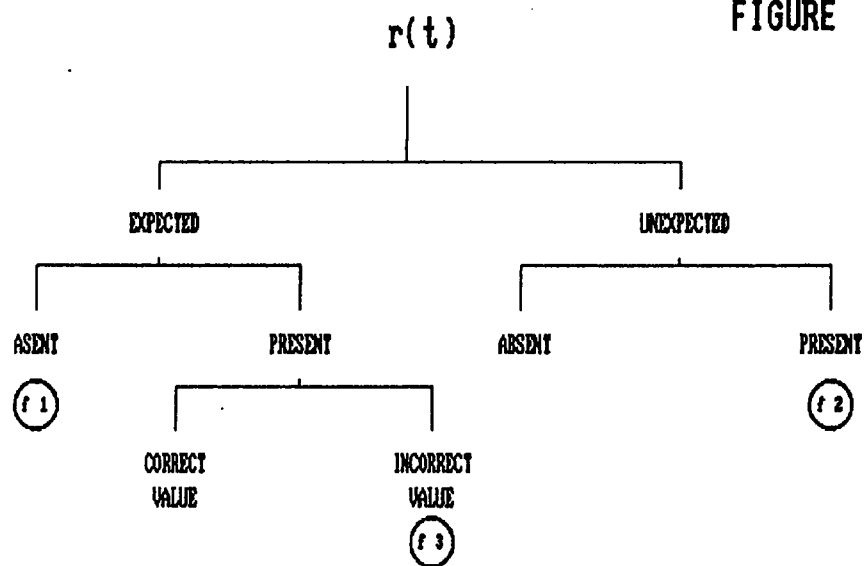


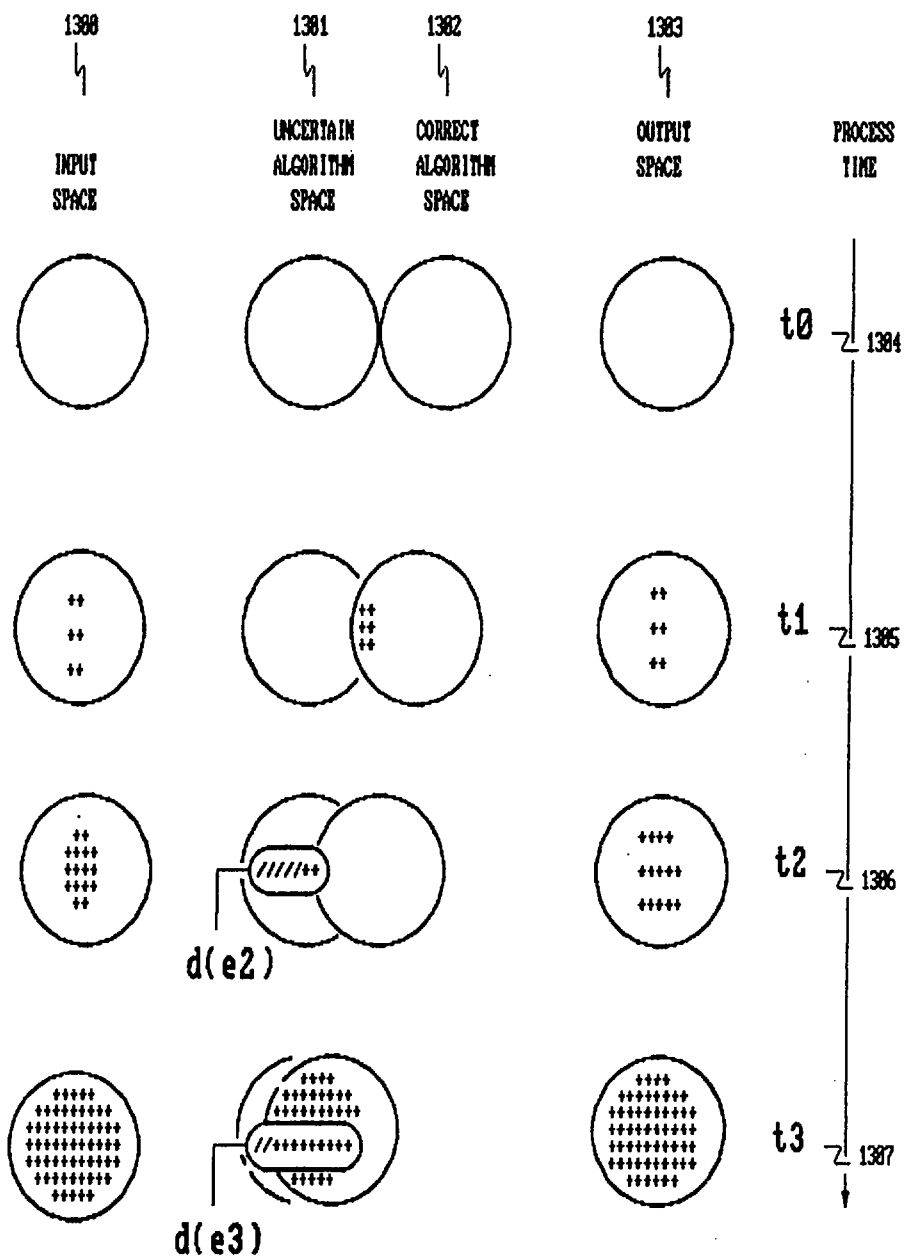
FIGURE 12



This Sheet Has Been Replaced

AMENDED SHEET

FIGURE 13



This Sheet Has Been Replaced

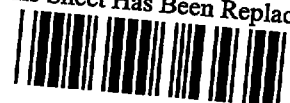


FIGURE 15

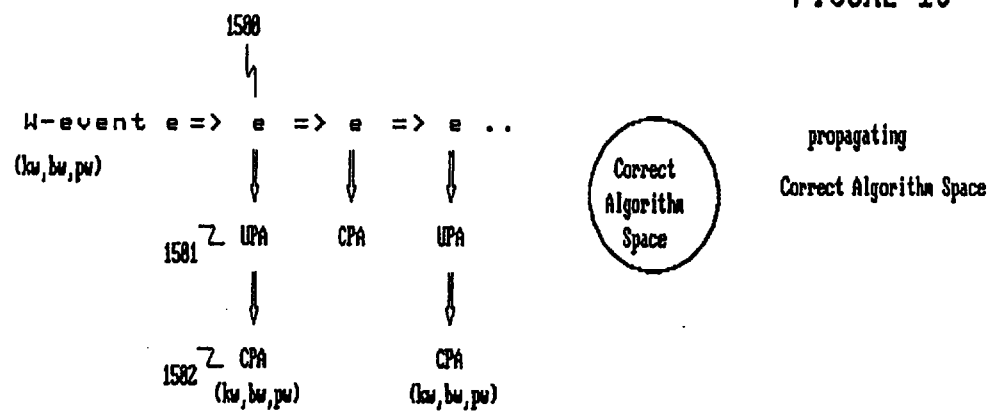
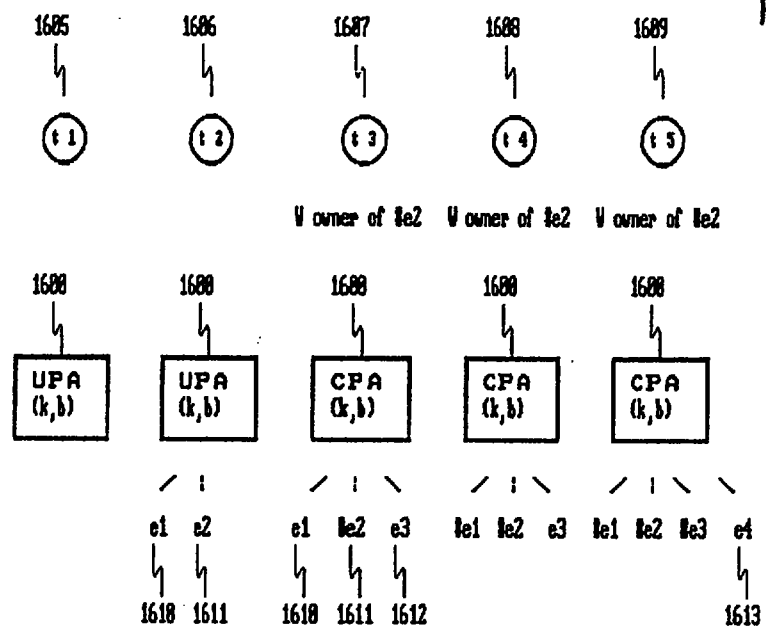


FIGURE 16



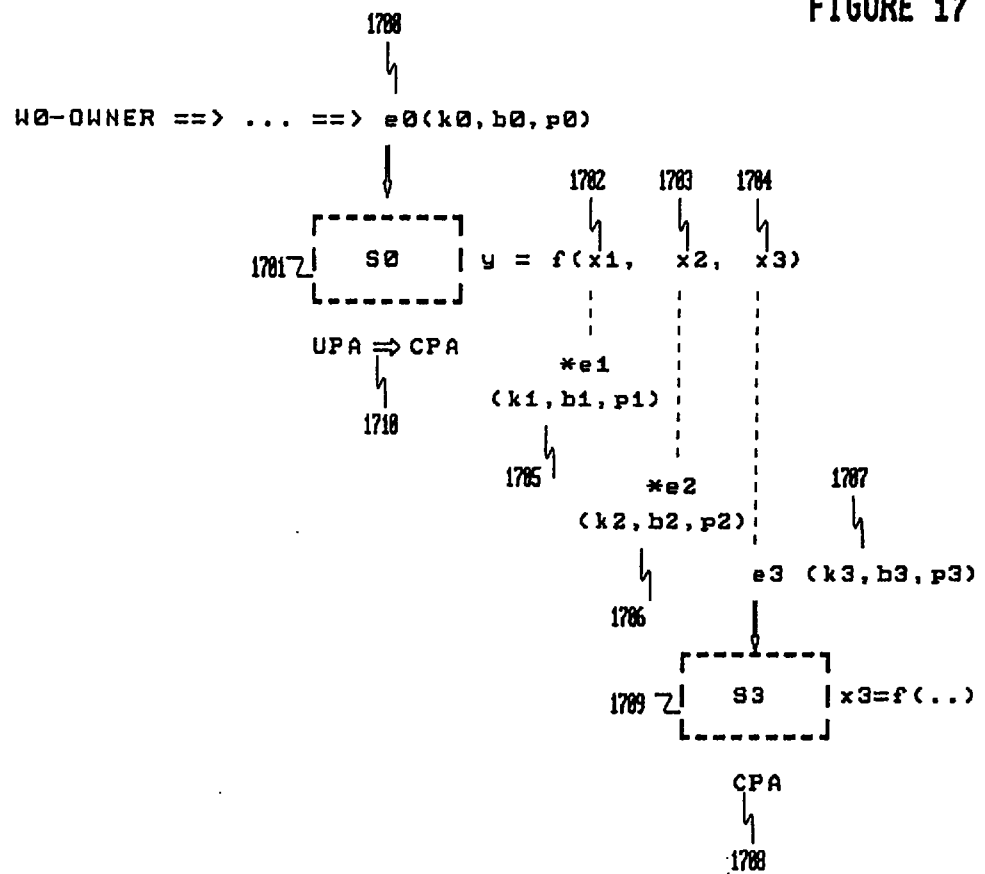
This Sheet Has Been Replaced



PCT/US 94/05182

IPEA/US 09 DEC 1994

FIGURE 17



This Sheet Has Been Replaced



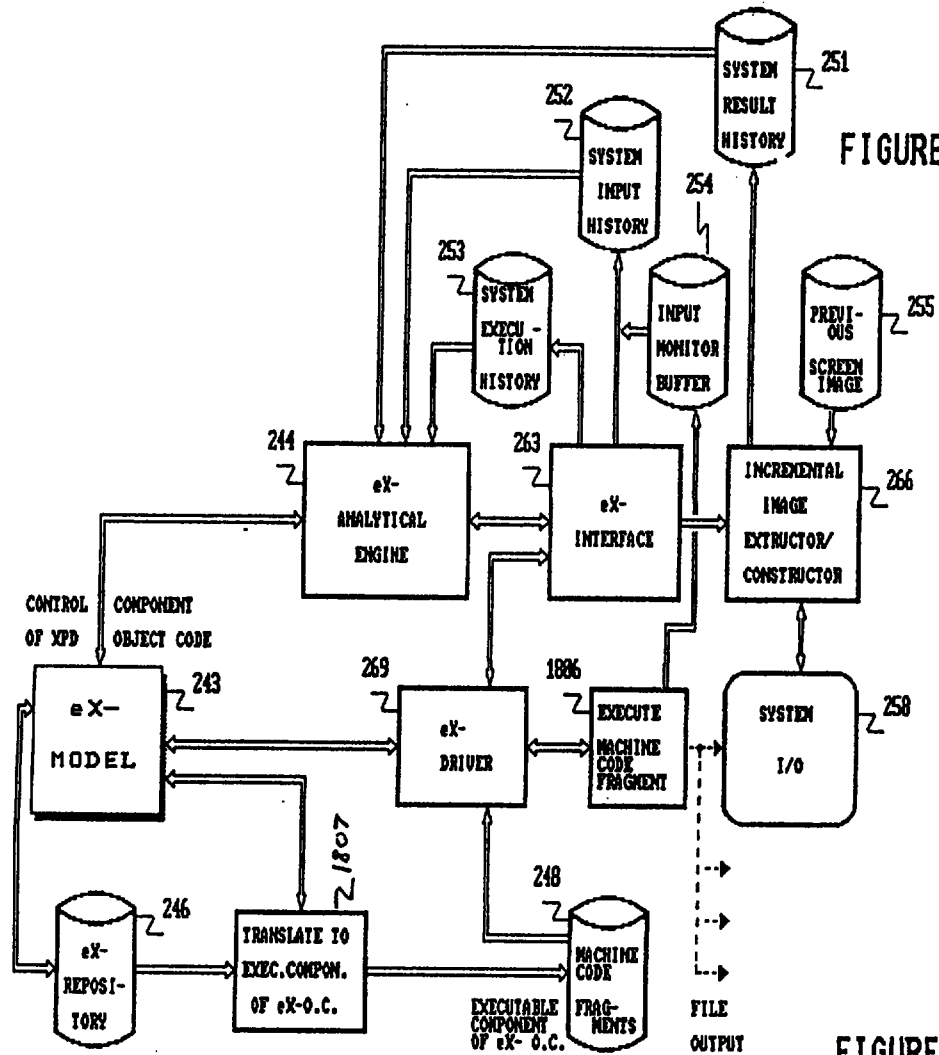


FIGURE 18 A

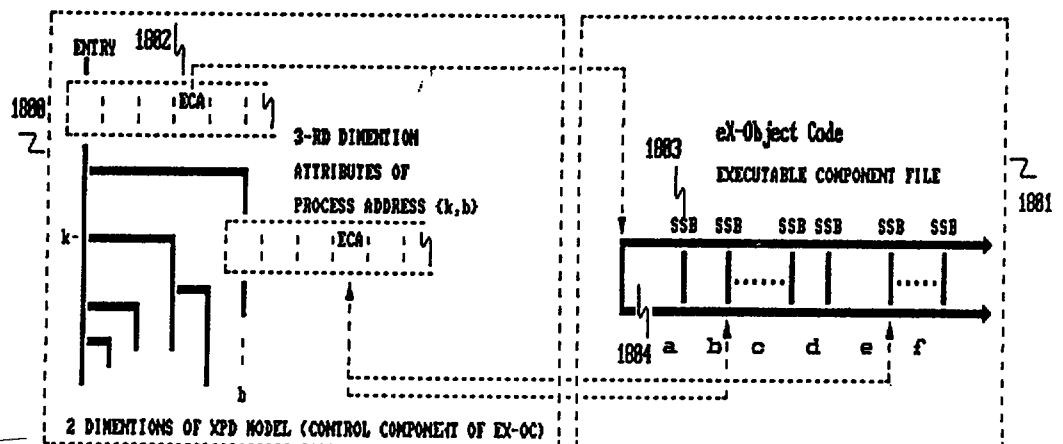


FIGURE 18 B



PCT/US 94/05282

IPEA/US 03 DEC 1994

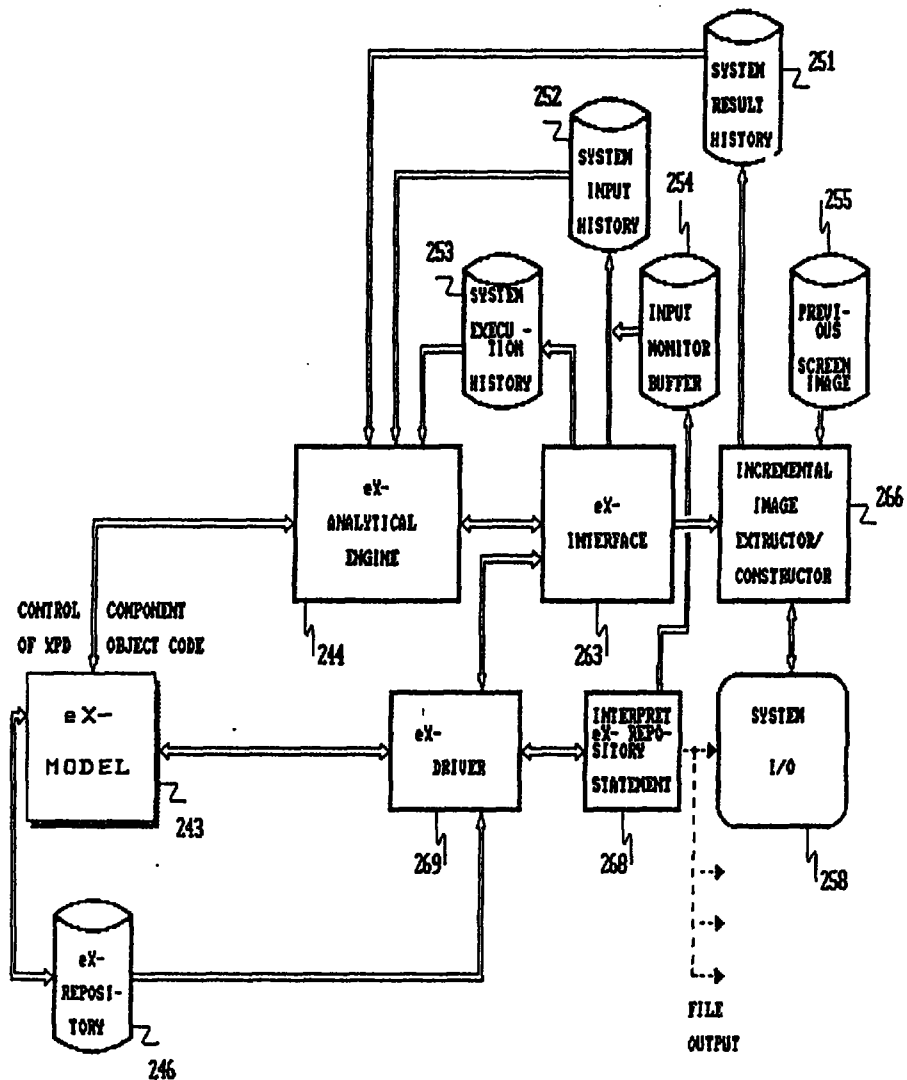


FIGURE 18 C

This Sheet Has Been Replaced



APPENDED SHEET

PCT/US 94/05182
IPEA/US 09 DEC 1994

FIGURE 19

	Control Area	Information Area
	NPW 1981	UDW 1982
		1983
1988 Z A1	2000 ₁₀	00011010010001101111
A2	127 ₁₀	00011010010001101111
D1	0 ₁₀	01101110010111101111
1985 Z E1-1	*E1-2 804 ₁₀	00111001100010010001
1986 Z E1-2	0 ₁₀	01111100011110010001
1987 Z X1-1	*X1-2 97 ₁₀	01001010110111001111
1988 Z X1-2	*X1-3 104 ₁₀	01111111111110010001
1989 Z X1-3	0 ₁₀	01000000110111001111

This Sheet Has Been Replaced



AMENDED SHEET

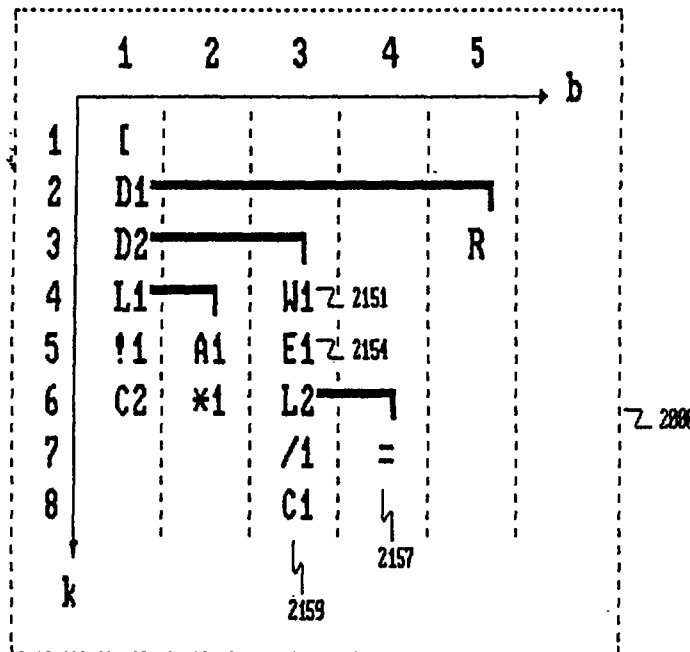


FIGURE 20 A

```

INTEGER IN(3)
CHARACTER IA(3), IALF(18), CH
DATA IALF/'8','1','2','3','4','5','6','7','8','9'/
001  FORMAT(3X,'IN(1)=',I3,' DOESNOT MATCH INTEGER FROM 8 TO 9 ',
        C' TYPE "Y" TO CONTINUE')
002  FORMAT(A)
D1:   I=1,3
D2:   J=1,18
L1:   IF(IN(I).EQ.J-1)
A1:   IA(I)=IALF(J)
W1:   WRITE(*,001)IN(1)
E1:   READ(*,002)CH
L2:   IF(CH.NE.'Y'.AND.CH.NE.'y')

```

FIGURE 20 B

This Sheet Has Been Replaced



FIGURE 21 A

2100

```

SUBROUTINE 12A(IN,IA)
  INTEGER IN(3)
  CHARACTER IA(3),IALF(18),CH
  DATA IALF/'0','1','2','3','4','5','6','7','8','9'/
  DO 10 I=1,3
    DO 55 J=1,18
      IF(IN(I).EQ.J-1)THEN
        IA(I)=IALF(J)
        GO TO 10
      ENDIF
55    CONTINUE
      WRITE(*,100)IN(I)  Z 2151
100    FORMAT(3X,'IN(I)=' ,I3,' DOES NOT MATCH INTEGER ',
C      'FROM 0 TO 9 TYPE "Y" TO CONTINUE')
      READ(*,20)CH  Z 2154
20    FORMAT(A)
      IF(CH.NE.'Y'.AND.CH.NE.'y') STOP Z 2157
18    CONTINUE  Z 2159
      RETURN  Z 2161
      END

```

FIGURE 21 B

2150

```

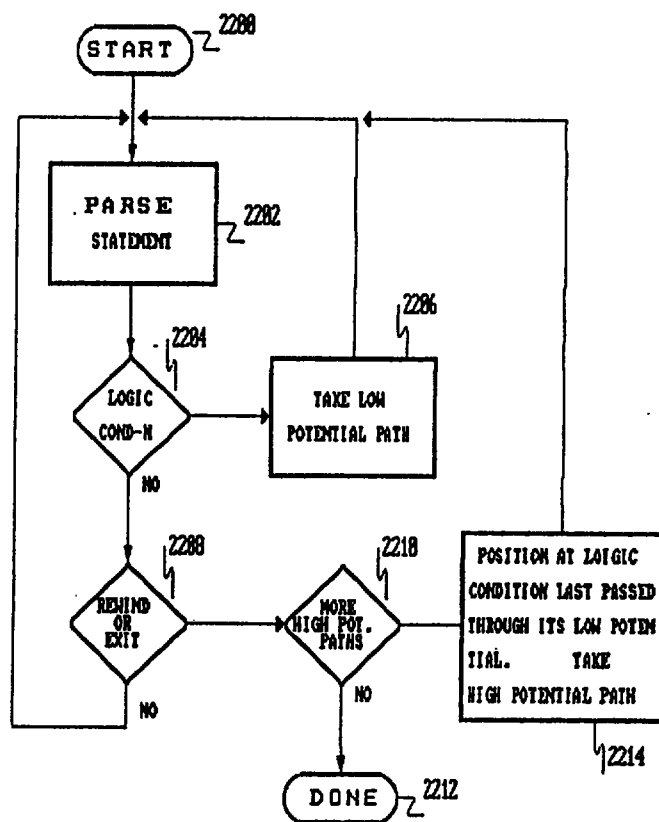
SUBROUTINE 12AXXX(IN,IA)
  INTEGER IN(3)
  CHARACTER IA(3),IALF(18),CH
  $INCLUDE: 'INCL-XPDI'
  DATA IALF/'0','1','2','3','4','5','6','7','8','9'/
  001  FORMAT(3X,'IN(I)=' ,I3,' DOES NOT MATCH INTEGER ',
C      'FROM 0 TO 9 TYPE "Y" TO CONTINUE')
  002  FORMAT(A)
  DO 3001 I=1,3
    DO 3002 J=1,18
      IF(IN(I).EQ.J-1)THEN
        IA(I)=IALF(J)
        0      CALL XPDI(14,2)
        GO TO 23001
      ENDIF
      0      CALL XPDI(6,2)
  3002  CONTINUE
      WRITE(*,001)IN(I)  Z 2151
      0      CALL XPDI(20,2)  Z 2152
      0      CALL XPDI(21,2)  Z 2153
      READ(*,002)CH  Z 2154
      0      WRITE(*,002)CH  Z 2155
      IF(CH.NE.'Y'.AND.CH.NE.'y') THEN
        0      CALL XPDI(31,2)  Z 2156
        STOP  Z 2157
      ENDIF
  23001  CALL XPDI(24,2)  Z 2158
  3001  CONTINUE  Z 2159
  0      CALL XPDI(35,2)  Z 2160
  RETURN  Z 2161
  END

```

This Sheet Has Been Replaced



FIGURE 22



This Sheet Has Been Replaced



AMENDED SHEET

PGI/US 94/05182
IPEA/US 09 DEC 1994

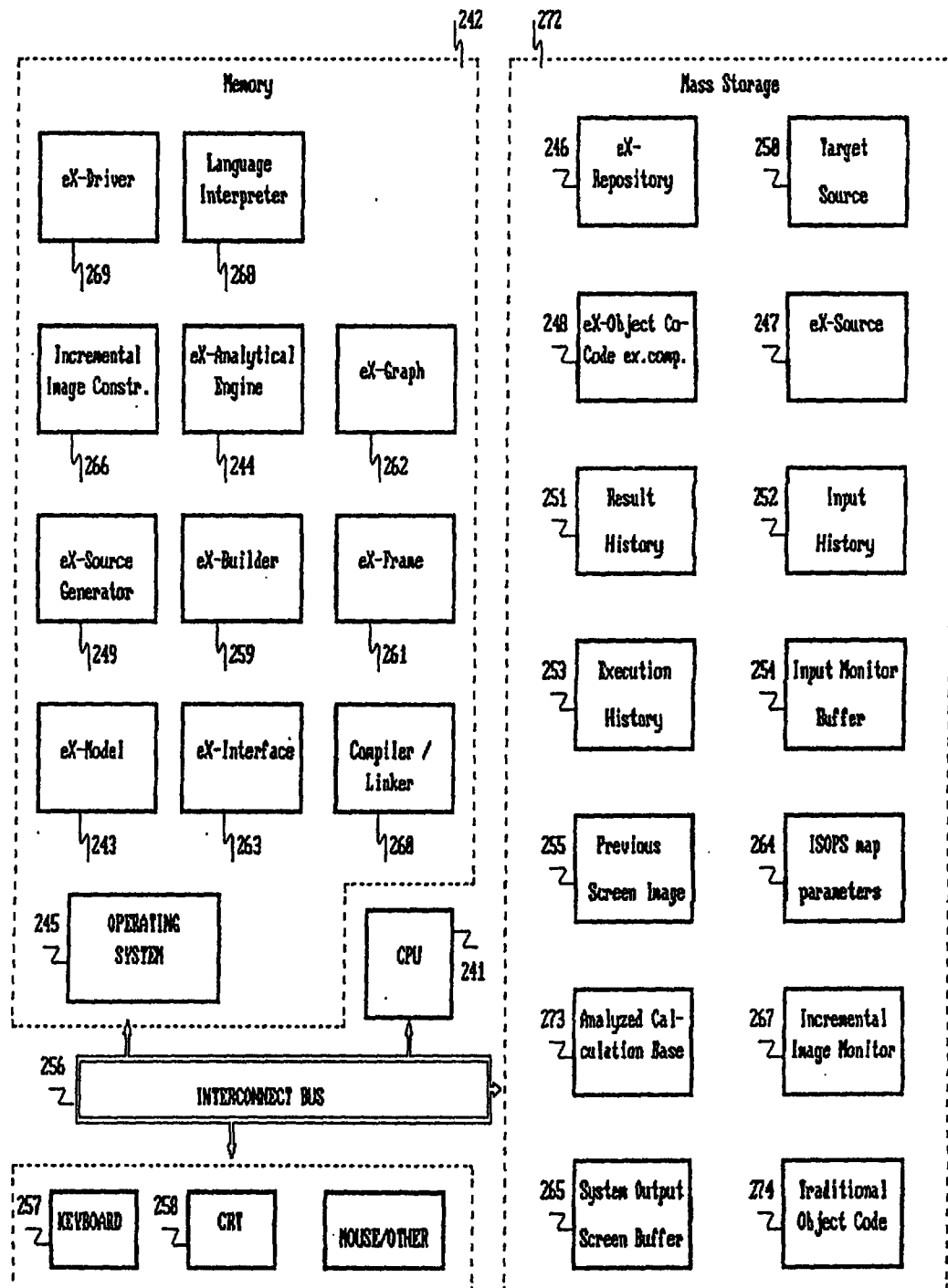
FIG. 23

```
Assign 0 to the active logic level. Set position to the Process Entry.
2310 Pass through the Main Branch
    Assign each passed executable statement to eX-Object
    Store assignment in eX-Repository
    IF Logic Condition
        Take Low Potential Solution
        Increment Active Logic Level by 1
        Represent opened logic level in the frame by opening parenthesis
            along with the integer representing Active Logic Level
        Start building its 0 subfield by repeating step 2310 (GOTO 2310)
    ENDIF
2320 IF reached Process Terminal
    The segment is terminated
    IF Rewind Terminal
        IF corresponding rewind entry position not yet exist in eX-Frame
            Assign next Rewind Terminal index
            Insert new rewind entry in corresponding
                eX-Frame position
        ENDIF
    ENDIF
2330 Examine Active Logic Level
    IF an Active Logic Level is 0
        eX-Frame is built
        STOP
    ELSE
        Logic Subfield is closed
2340 Examine Closed Subfield
        IF the closed subfield = 0-Subfield
            Place "V" at the next position of eX-Frame and therefore open
                1-Subfield of the corresponding Logic Condition
            Position at the High Potential solution of the corresponding Logic
                Condition
            Start building 1-Subfield by repeating step 2310 (GOTO 2310)
        ELSE
            The closed subfield was 1-subfield; both subfields closed;
            Close Active Logic field by placing closing parenthesis
                with corresponding logic level at the next position
                of eX-Frame
            Decrease active logic level by 1
            The Segment which terminal was designated in the
                eX-Frame by the corresponding logic condition is now
                terminated
            Repeat step 2330 (GOTO 2330)
        ENDIF
    ENDIF
```

This Sheet Has Been Replaced

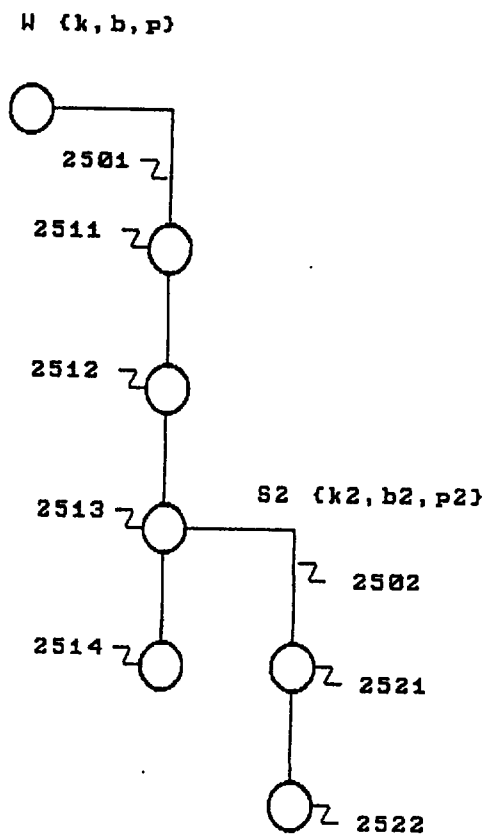


FIGURE 24



DCT/US 94/05182
IPEA/US 09 DEC 1994

FIGURE 25



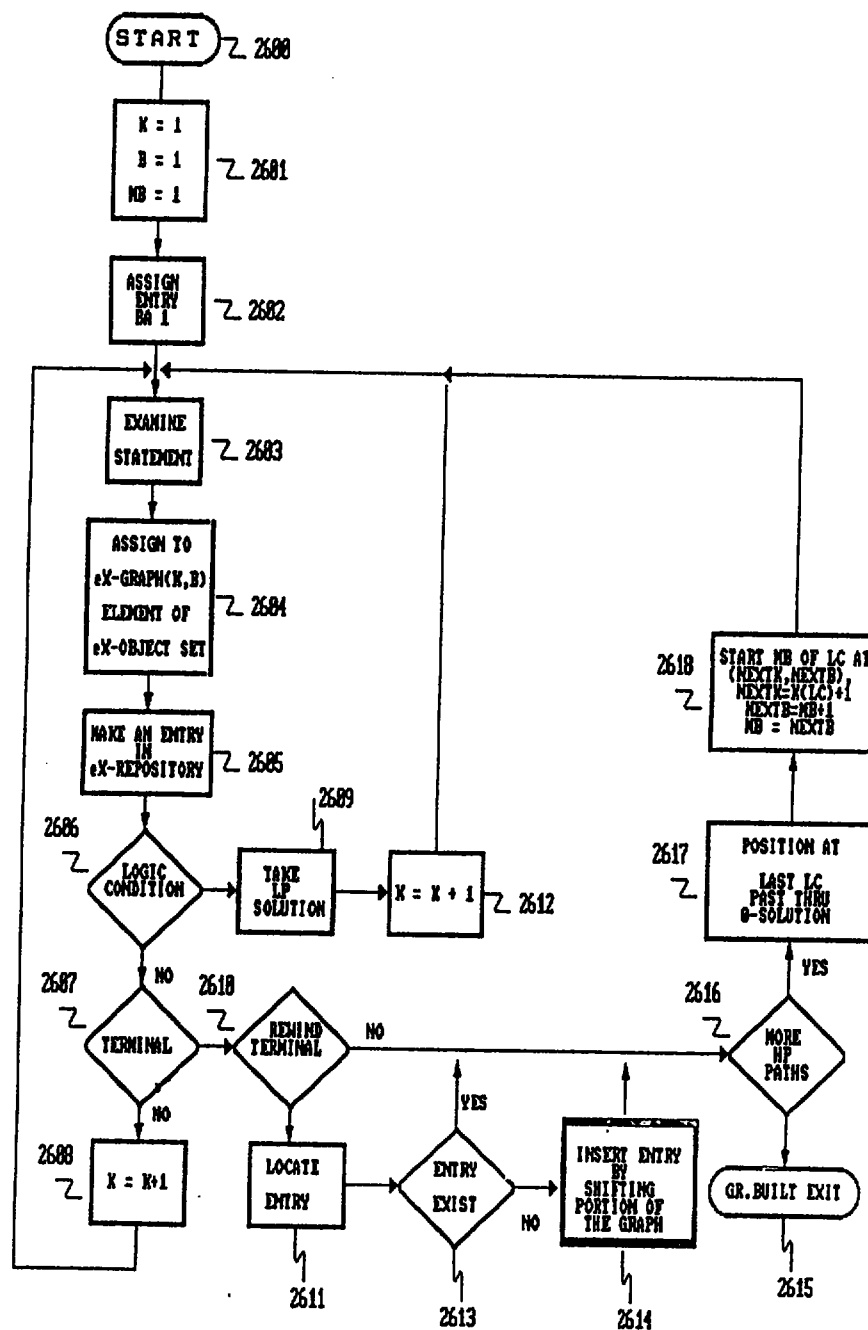
This Sheet Has Been Replaced



AMEND

PCT/US 94/05182
IPEA/US 09 DEC 1994

FIGURE 26



This Sheet Has Been Replaced



45 OF 407 (12) 5

PGI/US 94/05182

IPEA/US 09 DEC 1994

ISOPS map sample format:

1.	2.	3.	4.	5.
W- event	screen position	screen area changed	displ. in image monitor file	incremental image monitor file

_____ISOPS parameters file_____/_IIM file__/\

{k,b,p,m}	{sl,sc}	{l1,c1,l2,c2}	DDDDDD	'xxxxxxxxxxxxxxxxxxxx'
{k,b,p,m}	{sl,sc}	{l1,c1,l2,c2}	DDDDDD	'xxxxxxx'
{k,b,p,m}	{sl,sc}	{l1,c1,l2,c2}	DDDDDD	'xxxxxxxxxxxxxxxxxxxx'
{k,b,p,m}	{sl,sc}	{l1,c1,l2,c2}	DDDDDD	'xxxxxxx'
{k,b,p,m}	{sl,sc}	{l1,c1,l2,c2}	DDDDDD	'xxxxxxxxxxxx'

FIGURE
27

This Sheet Has Been Replaced



12 NOV 1997

- 1 -

METHOD AND APPARATUS FOR THE AUTOMATIC ANALYSIS OF COMPUTER SOFTWARE

BACKGROUND OF THE DISCLOSURE

1. Field of the Invention

The invention relates generally to the field of fault identification and analysis in computer software. More particularly, the invention relates to a method and apparatus used for the automatic analysis of computer software.

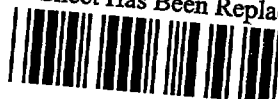
2. Background of the Invention

Since the day the first computer program was written and to this day, errors or faults in computer software are an expected and commonplace occurrence. Even though such faults are expected, fault identification and analysis is a problem which has forever plagued programmers. This due in part, to the fact that the fault rate for a carefully constructed program can be about 5 faults per 1,000 lines of source code so that a one million-line program may contain 5,000 faults! Compound this with the fact that computer software faults are different from computer hardware faults in that the software faults do not obey the same simplified rules of proximity of cause and effect. That is to say, a computer software fault does not necessarily have to appear close to its failure, but instead can be located almost anywhere within a much larger software system. Therefore, even when a fault is known to exist, determining the origin and location of that fault is extremely difficult at best.

3. Description of the Prior Art

As a result of the above mentioned characteristics of computer software, the prior art has developed a number of methods and apparatus to assist a programmer in the development and analysis of software. The methods and apparatus of the prior art encompass a wide array of systems and techniques, each generally applicable to a different period of the overall computer software development process. Among these prior art methods and apparatus are: 1) structured design techniques; 2) language-sensitive development tools; 3) highly-efficient and informative language interpreters and compilers; and 4) an assortment of program profilers and debuggers and emulators which assist in identifying and locating faults in software during the debugging and testing of the software. Despite all these tools, techniques and related efforts, a large amount of programmer time is spent in trying to understand how a program works and many failures are tolerated out of fear of further "breaking" a software system.

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

REPLACEMENT SHEET

12 NOV 1997

- 2 -

There are many problems associated with existing tools and techniques for identifying and locating faults in computer software. Traditionally, a programmer had to debug or analyze a program through the use of some kind of "hook" or output statement which was part of the programming language used for the software program. For example, in one particular language a "PRINT" statement would be inserted by a programmer into one or more locations of a program being analyzed so that upon encountering the PRINT statement, the program being analyzed would output to an appropriate output device the contents of some registers, variables, etc. In this manner, the programmer would examine the output of the executed PRINT statement searching for clues as to why the program failed. Needless to say, this method of program analysis and debugging is very tedious for the programmer and error prone as well.

Another method of program debugging and analysis involves the use of a special program, known as a program debugger which monitors the execution of a software program and oversees its execution. Such debuggers typically offer enhanced flexibility and more information to the programmer than possible with the simple PRINT method, particularly with respect to a user interface presented to the programmer. Utilizing a debugger, a programmer will normally specify one or more lines of the software program as "breakpoints", or points within a software program which when encountered suspend the execution of the program so that the programmer may examine processor registers or program variables. Breakpoints are not limited to specific lines of code, breakpoints may be used to interrupt execution when an expression changes value, or when an expression reaches a value. Despite apparent advantages over the PRINT method, program debugging and analysis using known tools suffer from many of the same infirmities i.e., they do not automate the understanding of cause-effect relationship between faults and failures (manifestation of fault).

The first problem associated with these above mentioned debugging and analysis methods and tools is where to begin. This is largely a hit or miss proposition for the programmer. Therefore, the programmer will typically insert a number of breakpoints or PRINT statements which are scattered throughout the program under test. As the programmer narrows down his understanding of the nature and location of program fault(s) responsible for a failure, unnecessary breakpoints or PRINT statements are normally removed or deactivated. This technique of randomly inserting breakpoints or PRINT statements throughout a program is highly inefficient, time-consuming, and error-prone.

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

12 NOV 1997

- 3 -

Second, even if the programmer has some idea where within a program to place a breakpoint, the programmer must determine what to examine at that location. That is, a particular section of a program may use a large number of variables, registers or memory locations, thereby making the programmer's job of identifying the fault(s) more difficult. While debuggers and analyzers have certainly facilitated the examination of variables, registers or memory locations, they have by no means eliminated the problem of making a decision as to what variables, memory, or registers to examine and when to examine them. Further more, debuggers do not help to decide what is the meaning of the value of a variable and what other variables of what other statements need to be examined until the fault statement is found. As can be appreciated, the traditional techniques and tools used for debugging or analyzing a software program is not highly automated.

In an attempt to overcome these stated deficiencies in the art of program debugging and analysis, the prior art has developed additional tools and techniques.

United States Patent No. 4,853,851, "System For Determining The Code Coverage Of A Tested Program Based Upon Static And Dynamic Analysis Recordings" issued to Borsch on August 1, 1989 teaches a system for analyzing programs by measuring the degree of code coverage of a program being tested during specific test phases. The system disclosed utilizes a static and dynamic instruction flow indicator which permits a determination of the test cover results by correlating the data of the static and dynamic instruction flow. In this manner, the number of untested statements can be determined during a test phase of a computer program.

Although the more statements covered during the test of a program the better, the fact that a statement was executed does not provide any information on the statement validity. In particular, the statement may have been constructed wrong; the statement may be executed at the wrong place at the wrong time; or the statement may not contribute to the observed result.

United States Patent No. 4,802,116, "Programmed Controller" issued to Ward, et al on January 31, 1989 teaches programmed controller which controls a machine process and facilitates its debugging. The application programs disclosed are specifically designed to simulate specific control schemes for sections of the machine by being organized into one or more separate loops. A built-in control mechanism records the states of these loops along with the conditions responsible for the transitions between those states. This teaching is limited in application to only those processes that are specially coded by very

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

12 NOV 1997

- 4 -

specific syntax rules as consisting of independent loops, where the means of tracing the states of these loops are pre-coded in the programs controlling the process.

A theoretical approach to the general problem of program analysis was disclosed by Thomas J. macabre in the article "A Complexity Measure" published in IEEE Transactions on Software Engineering, Vol. Se-2, No. 4, December 1976, pp. 308-320. Cyclomatic Complexity was proposed in this article as a measure of program complexity. This measurement is nothing more than the number of independent passes within a subroutine. This complexity measurement is useful to promote the design of less sophisticated software routines which are also easier to test. However, counting the number of independent passes does not eliminate their necessity

Despite these and other attempts made by the prior art, software debugging and analysis remains a difficult, time-consuming and error prone process. A critical shortcoming in the prior art is that there are no teachings for general methods or apparatus which are capable of automatically determining the location of logic faults in software programs even within a broad area of a program. This is due, in large part, to a general belief which exists in the art that such automation is not possible because software does not conform to the rules of proximate cause and effect and that a software program has no knowledge of what results are expected of it unless those expected results are pre-coded within the program. Such pre-coding is itself programming and therefore is also error-prone.

Therefore, a continuing need exists in the art for a method and apparatus which more highly automates the chore of software program debugging and analysis.

SUMMARY OF THE INVENTION

It is therefore the object of the present invention to provide a method and apparatus which automates the process of program debugging analysis and maintenance by allowing for automatic identification of an area of a program, which area contains a fault(s) responsible for specific failure or contains the statements to be modified in order to comply with the new requirements to the program. The mentioned area is automatically and continuously minimized while the invented system automatically learns about the target software program.

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

12 NOV 1997

- 5 -

The invention presents, what will be referred as "Software Understanding Machine", that is an apparatus and method that build and propagate the knowledge about the internal processes within the target software program.

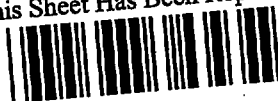
The invention thus presents the advantage that the method and device for debugging and analyzing software programs does so with precision that grows with the accumulation of knowledge about the target or analyzed program behavior. The invention successively accumulates this knowledge, and utilizes it to determine correct or uncertain behavior of the analyzed software program. The invention further shows the ability to partially "unlearn" previously acquired knowledge, thereby recognizing that correct results are a function of a user's expectations of the results which may change over time.

The invention operates by analyzing the results or effects produced by a software process and automatically traces back from these effects to their cause(s). In this manner failures, which are unexpected results, are automatically traced back to the fault(s) responsible. This is in contrast to traditional methods, i.e., stepping forward through the software process to precipitate a failure. The analysis is accomplished by the invention of the present application by first building a model of the software process under analysis on the basis of a static analysis performed under a set of strict rules. The model is constructed such that it serves as a framework for the further accumulation of knowledge which is acquired through a dynamic analysis of the software process.

The invention constructs the model of the target software process in such a manner that allows for the automatic determination of accessibility, or presence of a path, between any two elements of the target software process. Accessibility is required for dependency, that is, if one statement within a software process is accessible from another statement within the process, then, and only then, the correctness of the other statement may affect the behavior of the one statement under examination. If the second statement is not accessible from the first statement, then correctness/incorrectness of the first statement cannot affect the behavior of the second statement and therefore the second statement is not dependent upon the first statement.

During the construction of the model of the target software process, the invention extracts the control component of the target software process into a form which represents a control structure of that target process. Furthermore, the invention accumulates information about the behavior of the target software process thereby permitting the invention to determine the correctness of a software process element relative to the time at which the

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

12 NOV 1997

- 6 -

element is executed. Still further, this accumulated correctness information is propagated between target process elements (statements) thus automatically increasing the intelligence of the target process model. Through this accumulated knowledge that the invention acquires about the correctness of the software process behavior, the invention is able to automatically determine the location of fault(s) responsible for a failure.

These resources maintained by the inventive software analyzer, or "Software Understanding Machine", are presented to the user in an easy-to-use implementation. The preferred embodiment makes extensive use of an intuitive user interface as well as multimedia capabilities known in the art to further facilitate software debugging and analysis.

It is still another object of the present invention to introduce an original form of the target process object code, SUM-Object code, and a method for its execution. The SUM-Object code is more suitable for program analysis, both static and dynamic and for program maintenance. It eases and sometimes eliminates the traditional recompiling and re-linking of the target process code which are traditionally necessary in order to implement modifications to the target process. The SUM-Object code comprises the model extracted from the target process. Therefore, increasing the intelligence of the model performed by the "Software Understanding Machine" thus increases the intelligence of the resulting SUM-Object code.

BRIEF DESCRIPTION OF THE DRAWINGS

The teachings of the present invention can be readily understood by considering the following detailed description in conjunction with the accompanying drawings, in which:

FIG. 1 shows a flow chart of a sample target process.

FIG. 2 shows an SUM-Frame of the sample target process of FIG. 1.

FIG. 3 shows how complex logic conditions are reduced into two or more reduced logic conditions.

FIG. 4 shows a reduced flow chart of the sample target process shown in FIG. 1.

FIG. 5. shows SUM-Graph of the sample target process shown in FIG. 1 and the SUM-Binary addresses assigned to the corresponding elements of the SUM-Graph..

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

12 NOV 1997

- 7 -

FIG. 6 shows how typical IF-THEN-ELSE program constructs are represented in SUM-Frame and SUM-Graph form(s).

FIG. 7 shows the steps required to construct the SUM-Model, SUM-Repository and SUM-Source presented next to the main components of the Software Understanding Machine (SUM).

FIG. 8 shows the main components of the Software Understanding Machine (SUM) operating in the traditional object code implementation.

FIG. 9 shows the construction of exit access indicators and the construction of propagated vision points.

FIG. 10 shows the construction of the Propagated Vision Point of a process terminal.

FIG. 11 shows a target process with faults and their potential origin.

FIG. 12 shows three different types of target process failure(s).

FIG. 13 shows how (SUM) minimizes the area of uncertainty of the position of a fault within a target process, which fault is responsible for a specific failure of the target process.

FIG. 14 shows terminated event definition space.

FIG. 15 shows the propagation of correct algorithm space.

FIG. 16 shows the accumulation of knowledge about the correctness of a process element located within a specific process address.

FIG. 17 shows the knowledge propagation process implemented by constructing event definition spaces built only in specific directions and only up to the knowledge propagation terminals.

FIG. 18 shows the main components of SUM operating in the SUM-Object code implementation, the two components which comprise SUM-Object code and an alternative implementation of SUM through interpretation of the SUM-Repository by the SUM-Driver controlled by the SUM-Model.

FIG. 19 shows the construction of an analyzed calculation base.

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

12 NOV 1997

- 8 -

FIG. 20 shows an SUM-Graph, SUM-Repository of a sample process.

FIG. 21 shows a sample process and its SUM-Source code.

FIG. 22 is a flow chart depicting the steps which comprise the Next Lowest Potential method or rule of parsing a target process.

FIG. 23 is a pseudocode listing which describes the steps necessary to create an SUM-Frame of the target process by the next lowest potential method.

FIG. 24 is a system block diagram showing the overall analysis system and the components parts which it comprises.

FIG. 25 shows the construction of event definition space terminated by knowledge propagation terminals.

FIG. 26 is a flow chart depicting the steps necessary to create an SUM-Graph of a target process by the next lowest potential method.

FIG. 27 shows a sample format of Incremental System Output Process Synchronized map (ISOPS map).

To facilitate understanding, identical reference numerals have been used, where possible, to designate identical elements that are common to the figures.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

After considering the following description, those skilled in the art will clearly realize that the teachings of this invention can be readily utilized in the analysis of software programs or processes.

FIG. 24 is a block diagram of the Software Understanding Machine (SUM) analysis system 240 which comprises the present invention. Some of the elements contained within the analysis system are well known in the art as they are typically found in conventional personal computers. Well known components of the analysis system includes, keyboard 257, a CRT display 258, CPU 241, memory 242, and some form of mass storage facility 272 (e.g., a hard disc drive). These systems communicate via and are frequently capable of receiving a wide array of options via interface bus 256.

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

12 NOV 1997

- 9 -

It must be noted, that whether different components of the system reside on the mass storage or in the memory is often not important and is frequently controlled by the operating system.

A target software process 250 is defined by a target source program comprising one or more program statements. The target source is typically stored on mass storage device 272. Target process 250 is extracted from target source by SUM-Builder program 259. The SUM-Builder 259 examines the loaded target process 250 and constructs an SUM-Graph 262 of that target process 250.

An SUM-Repository 246 of the target process is built by the SUM-Builder 259 at the same time the SUM-Graph 262 is constructed. The SUM-Repository 246 so constructed is "control-less", in other words, it does not contain any process control statements which were present in the target process 250. Instead, process control information is preserved within the SUM-Model 243.

The SUM-Model 243 comprises one or more elements, each element corresponds to an executable statement of the target process 250. Multiple SUM-Models 243 are constructed when the target process 250 comprises multiple components, i.e., main, subroutines, functions, etc.

After the SUM-Model 243 and SUM-Repository 246 are built, SUM-Source Generator 249 constructs SUM-Source code 247 from the information contained within SUM-Model 243 and SUM-Repository 246. The SUM-Source 247 is representative of the original target process 250, since it is constructed from SUM-Model 243 and SUM-Repository 246 which in turn, were constructed from the target process 250.

The SUM-Source 247 is only created when SUM-Analyzer 240 is used in its traditional object code embodiment. The SUM-Source 247 permits the SUM-Analyzer 240 to operate with the traditional, object code representation of target process 250, which should be very familiar to those skilled in the art. The SUM-Source 247 so constructed contains "synchronization signals", i.e., CALL() statements or WRITE() statements which are inserted into SUM-Source 247 by SUM-Source Generator 249.

The synchronization CALL() statements reference the SUM-Interface library 263. The SUM-Source code 247 is compiled by compiler/linker 260 and the resulting code is linked with SUM-Interface library 263 thereby resolving the CALL() references. The output code produced by the

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

12 NOV 1997

- 10 -

compiler/linker 260 operating on SUM-Source code 247 and SUM-Interface library is then executed by CPU 241. During the execution of this code, the synchronization statements embedded into SUM-Source and now compiled/linked, are executed which, in turn invoke the execution of SUM-Analytical Engine 244.

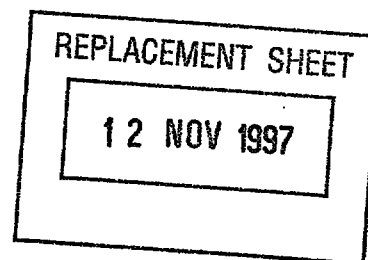
The creation of SUM-Source 247 and compiling/linking it with SUM-Interface library in order to send synchronization signals to the SUM-Model is not done in the embodiment utilizing the SUM-Object code or in the embodiment utilizing a language interpreter operating on SUM-Repository. In the two last mentioned embodiments the model controls the execution of the target process.

In a first embodiment utilizing traditional object code and upon receipt of a later synchronization signal from the SUM-Interface 263, SUM-Analytical Engine 244 traces the SUM-Model 243 from the point at which a previous synchronization signal was received up to the position within the SUM-Model 243 where the later synchronization signal was received. In this manner, the SUM-Analytical Engine 244 effectively "models" the target process by "executing" SUM-Model 243.

In the other two embodiments, i.e. SUM-Object Code 243 embodiment or SUM-Repository 246 interpretation by the Language Interpreter 268 embodiment, the SUM-Model 243 is not "executed" after the target process by the SUM-Analytical Engine 244, but instead, the SUM-Model is "executing" the target process by controlling the SUM-Driver 269, that passes corresponding elements of Executable Component of SUM-Object Code 248 for execution or corresponding statements of SUM-Repository 246 for interpretation. In these two embodiments the synchronization signals representing the state of the SUM-Model 243 are passed to SUM-Interface 263 by SUM-Driver 269.

In all of the mentioned embodiments the SUM-Analytical Engine 244 constructs the Analyzed Calculation Base (ACB) 273 by tracing effects to their cause(s). This process is called in the present invention a Knowledge Induction process. It is done in the direction opposite to the direction of the target process execution and only as far as defined by the Knowledge Propagation Terminals, that will be explained further. In this manner, the knowledge of the correctness or uncertainty of the effect of one statement execution event is propagated backwards to define the knowledge of the correctness/uncertainty of the effect of another statement execution event. The ACB 273 comprises Analyzed Calculation Words (ACW) which describe the knowledge of the correctness/uncertainty of target process elements (statements

This Sheet Has Been ReplacedTHIS SHEET HAS BEEN
REPLACED



and parameters within the statements) as a function of time within the process.

System Execution History, 253, System Input History 252 and System Result History 251 are all produced during the execution of the SUM-Model 243. These structures will be described in greater detail later in this description.

The system receives user input through keyboard 257 or other input device and informs the user of analysis results and/or progress via CRT 258.

FIG. 1 is a flow chart representing a sample target source process 250 that is a single module although our results can be easily extended to a process comprising a system of modules. The module could be either a main program or a single subprogram (i.e., subroutine, function, or paragraph) with one entry point. Although the example shown has only a single exit, the invention will work in an identical manner for modules with multiple exits. Those skilled in the art could readily extend our teachings to accommodate modules having multiple entry points.

During the initial examination of target source 250 by SUM-Builder 259, SUM-Model 243 may be built from two predecessor structures, SUM-Frame 261 and SUM-Graph 262. A representative SUM-Frame, so constructed from the target source 250 is shown in FIG. 2.

In the following description of the construction of the SUM-Frame, the below given designations and definitions will be used:

Table 1 SUM-Object Set

Designation	Object	Example
Process Entry	[	
End of process description	]	
Arithmetic or Assignment Statement	A	
Function Call other than Graphics Library	X	CALL, PERFORM
Logic Condition	L	IF
Loop Logic Condition	D	DO, WHILE, FOR
User Input Statement	E	READ
System Output Statement	W	WRITE(); CALL MOVETO(x,y,xy)
Null Statement	N	
Stop	=	STOP
Return	R	RETURN, EXIT

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

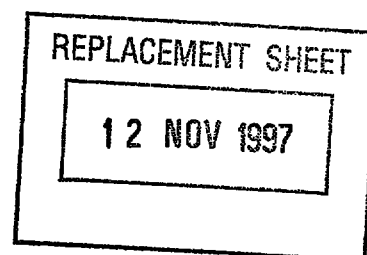


Table 1 (continued)

Unconditional Branch	*	GO TO
Loop Terminal Statement	C	CONTINUE; ENDDO
Assigned GO TO	+	
ASSIGN	S	
Entry of GO TO	/	
ENDIF	!	
Opening of Logic Condition	(	
Closing of Logic Condition	)	
Logic Condition Alternative	V	ELSE; OR

Definition F1: A Reduced Logic Condition (LC) is a process element with one control input and two control outputs. One of the two control outputs are assigned to Boolean 0 (False) and the other control output is assigned to Boolean 1 (True) which correspond to a solution to the logic condition.

Definition F1.1: The Low Potential Solution of a logic condition is the "0" Boolean solution (False) to the logic condition.

Definition F1.2: The High Potential Solution of a logic condition is the "1" Boolean solution (True) to the logic condition.

Definition F2: The Process Terminals are comprised of the following members of the SUM-Object set shown in Table 1: *, +, C, R, =.

Definition F3: The Main Branch of a process is the path traversed through a process from an origin of the main branch (process entry or high potential solution of logic condition) to a process terminal when all of the logic conditions which comprise that path are passed through their low potential solutions.

Referring to FIG. 9, the Main Branch is the path from entry 900, through logic conditions 904, 914 and 924, until terminal 930 is reached.

Definition F4: Exit Terminals result in an exit from a process and are comprised of the following members of the SUM-Object set shown in Table 1: R, =. As shown in FIG. 9, terminal 960 is an exit terminal.

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

12 NOV 1997

- 13 -

Definition F5: Rewind Terminals are terminals that rewind the process to an element that is in the same main branch or in a main branch to the left of the current main branch. An exception to this rule is when the rewind terminal is an exit of a loop. In such a case, the rewind terminal could reset the process to an element that is within a branch located to the right of the current branch. As shown in FIG. 9, terminals 908, 930, 944, 950, 952 and 958 are all rewind terminals.

Definition F6: A t-pass is a pass from the process entry to the process terminal. In any process there are $(n+1)$ t-passes, where n is the number of logic conditions in the process. Each t-pass is uniquely identified by its terminal.

In FIG. 9 there is shown a single entry 900, and seven terminals; 908, 930, 944, 952, 950, 960, and 958. Of those seven, 930, 944, 952, 950, 958, and 908 are all rewind terminals while terminal 960 is an exit terminal. Shown further in FIG. 9 are six logic conditions; 904, 914, 924, 938, 946 and 956. A single t-pass would be represented by entry at 900, progressing along line 902, taking the high potential path from logic condition 904 along line 906 until terminal 908 is reached. This t-pass would be uniquely identified as, for example, t-pass 908 where 908 identifies the terminal of the t-pass.

Definition F7: A branch of a logic condition is that part of a t-pass which begins at that logic condition. Each logic condition has at least one 0-branch and at least one 1-branch. Referring once again to FIG. 9, a high-potential or 1-branch of logic condition 904 is represented by line 906 and terminates at terminal 908. There are two 1-branches of logic condition 914 and they are ended by terminals 960 and 958.

Three of the four elements contained within dotted block 964, namely 954, 956, and 960, constitute a main branch of logic condition 914. The other element contained within dotted block 964, namely 958, constitutes a main branch of logic condition 956. The combination of all the main branches of a process necessarily and sufficiently represent all the elements which comprise that process. Further, there could exist many paths through a process element but that process element can belong to only one main branch.

Definition F8: The 0-subfield of a logic condition is a combination of all elements situated on the 0-branch(es) of that logic condition.

This Sheet Has Been Replaced

THIS SHEET HAS BEEN
REPLACED

REPLACEMENT SHEET

12 NOV 1997

- 14 -

Definition F9: The 1-subfield of a logic condition is a combination of all elements situated on the 1-branch(es) of that logic condition.

Definition F10: The field of a logic condition is a combination of the 0-subfield and 1-subfield of that logic condition.

Definition F11: The level of the logic condition field or logic level, is the numeric order in which that logic condition was encountered in the t-pass going through the logic condition.

Referring once again to FIG. 9, logic condition 904 is situated at logic level 1. Logic condition 914 is situated at logic level 2. Logic conditions 924 and 956 are both situated at logic level 3. Logic condition 938 is situated at logic level 4. Logic condition 946 is situated at logic level 5.

Definition F12: A process segment is that part of a main branch which is positioned between process entry, process terminals and logic conditions.

Process terminals and logic conditions terminate process segments. They belong to the process segments that they terminate.

An SUM-Frame is constructed in the present invention by representing the main branches of the target process in a special order along with representing the levels of the logic conditions and their corresponding fields.

Construction of the SUM-Frame

The first step performed by SUM-builder 259 is the construction of the SUM-Frame which is representative of the target process 250. This step is not needed if the SUM-Graph is constructed directly from the target process. The target process is processed by the SUM-Builder through the "Next Lowest Potential" rule or alternatively, through the complementary "Next Highest Potential" rule. FIG. 22 is a flow chart depicting the steps associated with the analysis of a software process by the next lowest potential rule. The SUM-Builder proceeds by beginning its analysis at start block 2200. Successive statements of the process are parsed by block 2202 until a logic condition or process terminal is met. If a logic condition is met as determined by block 2204, the low potential solution of this logic condition is taken as dictated by block 2206. When a rewind or exit terminal is encountered as determined by block

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

12 NOV 1997

- 15 -

2208 and untraversed high potential paths remain in the software process as determined by block 2210, the block 2214 will be executed. The SUM-Builder locates the logic condition, that was last passed through its low potential solution and which high potential path was not yet taken. SUM-Builder then takes its high potential path and proceed via block 2202, therefore building the Main Branch of that logic condition.

These steps repeat until no untraversed high potential paths are left in the target software process.

In order to provide the reader with additional clarity, FIG. 23 shows a PSEUDOCODE implementation of the steps involved in analyzing a target software process by the next lowest potential rule. Additionally, FIG. 23 shows at what points within the steps modifications are made to the SUM-Frame under construction and at what points in the process statements are stored in the SUM-Repository.

FIG. 2 shows an SUM-Frame 20 which was constructed from the flow chart of a target process shown in FIG. 1 by the next lowest potential rule. By inspection of FIG. 2, the reader can readily see that the SUM-Frame 20 is built in the form of an algebraic expression containing several levels of parenthesis. In SUM-Frame 20, the parenthesis are representative of the logic levels. The same general rule is applied to this SUM-Frame 20 that applies to any algebraic expression, that is, the lower (outer) level logic field cannot be closed until the higher (inner) level logic field is first closed.

Referring once again to FIG. 2, the different levels of SUM-Frame 20 become apparent. Dotted line 21 identifies that portion of SUM-Frame 20 which is the first level. Line 22 represents the second level. Dotted lines 23 show the third level. Dotted line 24 shows the fourth level and dotted line 25 shows the fifth level. Within the SUM-Frame, the fields of logic conditions are between the corresponding parenthesis and integers corresponding to the parenthesis represent the logic levels of those fields.

The labeling of the target process elements which are represented within the SUM-Frame 20 proceeds using the SUM-Object set members previously defined. Table 2 shows the SUM-Object set members of the resulted SUM-Frame and their respective process segments.

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

REPLACEMENT SHEET

12 NOV 1997

Table 2.

Process Segments SUM-Object Set Representation

1	L1
2	/7, A1, /1, L2
3	A2, L3
4	X1, *1
5	A3, /2, L4
6	A4, /3, X2, *2
7	/4, A5, /6, L5
8	A6, *3
9	W1, *4
10	A7, L6
11	R1
12	*6
13	E1, *7

Upon completion of the construction of the SUM-Frame 20, an SUM-Graph is built by the SUM-Builder 259. An SUM-Graph of target process 250 shown as a flow chart in FIG. 1, is shown in FIG. 5(A). The SUM-Graph is constructed along two perpendicular axis which are designated as k and b in the graph and are referenced by 501(A) and 500(A) respectively. The origin of the two perpendicular axis is at the top-left of the graph.

Main branches of the graph are directed vertically down and represent tracks by which the target process 250 execution progresses, unless the track is switched as the result of the high potential solution to a logic condition encountered in the track. As shown in FIG. 5(A), every track has a unique b coordinate. Every track of the graph is parallel to one another.

The graph is constructed and populated such that an SUM-Object set member representative of the target process 250 is inserted at appropriate k,b coordinate(s). For example, as shown in FIG. 5(A) at k,b coordinate 1,1 there is placed an SUM-Object set member '[' (left brace) which is indicative of a process entry as defined previously in Table 1.

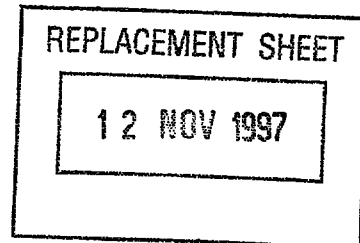
In the following description of the construction of the SUM-Graph, the below given designations and definitions will be used:

Definition G1: A Reduced Flow Chart is an SUM-Graph represented in the form of a flow chart. FIG. 4 shows a reduced flow chart for the target process represented by the flow chart of FIG. 1.

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED



A reduced flow chart is presented along two perpendicular axis, k and b which are shown as 401 and 400 in FIG. 4, respectively. The origin of the two axis is at the top-left-most portion of the chart. The k axis increases as one proceeds down the chart and the b axis increases as one proceeds to the right of the chart.

All possible changes in the direction of control is reduced to one - to the right as the chart is traversed. In this manner, all high potential solutions to a logic condition are represented as a shift to the right in the graph. Only backward (up and left) jumps of unconditional control statements are possible when a software process is represented by a reduced flow chart. An exception to this general rule is an exit from a loop. Such an exit may be made forward.

Definition G2: The negative state of a logic condition is the state of a logic condition which, when evaluated results in its 0-solution or low potential solution. With reference to FIG. 5(A), logic condition L1 is represented at k,b coordinate 2,1. Traversing the SUM-Graph shown in FIG. 5(A) through L1 with L1 in its negative state would result in relocating to k,b position 3,1 which is represented by SUM-Object set member /7 in the graph.

Definition G3: The positive state of a logic condition is the state of a logic condition which, when evaluated results in its 1-solution or high potential solution. With reference to FIG. 5(A), traversing the SUM-Graph shown in FIG. 5(A) through L1 with L1 in its positive state would result in relocating to k,b position 3,7 which is represented by SUM-Object set member E1 in the graph.

Definition G4: The binary address (BA) of an SUM-Graph element shows that elements position relative to process logic constructs and is represented by the integers 1 and 0 where the integers comprising the BA represent the solutions of the logic conditions in the t-pass up to that element with the process entry assigned the binary address of 1.

The binary address of those elements within the same process segment are the same. The binary address of a 0-segment of a logic condition is generated by shifting the address to the left by one bit position and placing 0 at the rightmost position of the binary address. The binary address of a 1-segment of a logic condition is generated by shifting the address to the left by one bit position and placing a 1 at the rightmost position of the binary address.

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

12 NOV 1997

- 18 -

Referring once again to FIG. 5(A), the binary address of k,b position 8,1 (L3) is 100. This is constructed from the process address (binary address 1), the negative solution (0) to the logic condition L1 located at k,b position 2,1 and a negative solution to logic condition L2 located at k,b position 6,1 in the graph. As a further example the binary address of A7 (located at k,b position 7,5 in FIG. 5(A)) is 101. This is constructed from the process address (binary address 1), the negative solution (0) to the logic condition L1 located at k,b position 2,1 in the graph and the positive solution (1) to the logic condition L2 located at k,b position 6,1 in the graph.

Construction of the SUM-Graph

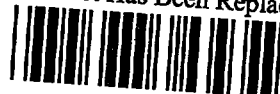
The SUM-Graph 262 which is representative of target process 250 is built by the same next lowest potential rule previously described. The SUM-Graph 262 may be built from a previously constructed SUM-Frame 261 or built directly from analyzing the target process 250.

FIG. 26 is a flow chart depicting the steps associated with the analysis of a software process by the next lowest potential rule and the construction of the resulting SUM-Graph. The analyzer proceeds by beginning its analysis at start block 2600. The position on the graph is assigned to k,b position of 1,1, 2601, and the entry binary address is assigned to 1 as indicated in block 2602. Successive statements of the process are examined by block 2603 with the SUM-Graph(k,b) position being populated with the SUM-Object set member representative of the examined statement as required by block 2604. The repository is updated in block 2605 and if the statement under examination presently is a logic condition 2606, then the low potential solution to the logic condition is taken 2609, the present k,b position is updated by $k = k+1$ as stated in block 2612 and the statement reexamination block 2603 is reentered.

If upon examination by block 2606 it is determined that the statement is not a logic condition, then the statement is tested in block 2607 to see if it is a terminal. If the statement so tested is not a terminal statement then the present k,b position is updated by $k = k+1$ (block 2608) and the statement reexamination is reentered.

If the statement examined by block 2606 is not a logic condition and it is a terminal (block 2607) then if it is a rewind terminal other than a loop terminal the rewind entry is located. If it does not exist, it is inserted in the SUM-Graph (block 2614). A determination is made at block 2616 as to whether any further untraversed high potential paths exist (i.e., a logic condition for which a main branch is not yet built). If there are no remaining untraversed high

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

12 NOV 1997

- 19 -

potential paths in the target process under examination, then the graph is built and the SUM-Builder terminates 2615.

If there were untraversed high potential paths as determined by block 2616, then the SUM-Graph position is updated to the last logic condition passed through its 0-solution and not yet passed through its 1-solution (block 2617) and a new main branch of that logic condition is begun at SUM-Graph position(nextk, nextb) where nextk is assigned to k(logic condition) + 1 and next b is assigned to the last used b+1 (block 2618). The last used b is updated and the process then continues with the examination statement 2603.

FIG. 5(A) shows an SUM-Graph constructed by the SUM-Builder 259 using the above stated method of the present invention. Once the target process 250 is represented as SUM-Graph 262 as constructed by the SUM-Builder 259, a number of useful characteristics of SUM-Graph 262 as shown in FIG. 5(A) emerge.

By examination of the SUM-Graph shown in FIG. 5(A) is should be apparent that every element represented within the SUM-Graph may be addressed by a unique coordinate pair (k,b). For example, logic condition L1 is located at k,b position 2,1 in the graph. Logic condition L6 is located at k,b position 8,5 in the graph. Table 3 shows the elements, their corresponding and unique k,b coordinates and their resultant binary addresses which are created during the analysis of target process 250

In the present preferred embodiment, the SUM-Graph is built in only two directions - down and to the right as one looks at the graph. When the SUM-Graph so constructed is traversed, the traversal proceeds down the graph and can only be interrupted by shifting to the right as a result of a high potential solution to an encountered logic condition.

Referring once again to FIG. 5(A) the graph traversal would begin at k,b position 1,1 (process entry) and proceed through logic condition L1 (k,b position 2,1), through label /7 (k,b position 3,1), through arithmetic A1 (k,b position 4,1) and so on until terminal *1 (k,b position 10,1) is reached. Such a traversal would proceed in a single direction, down the graph. If however, a high potential solution to a logic condition was passed through during traversal then a shift to the right would have taken place. For example, during the above traversal, if the logic condition L1 (k,b position 2,1) were passed through its high potential solution then a shift in flow would take place to E1 (k,b position 3,7).

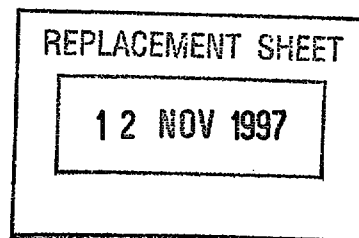
This Sheet Has Been Replaced

THIS SHEET HAS BEEN
REPLACED

Further, every segment is assigned a binary address that uniquely identifies the logic position of that segment within the target process 250. FIG. 5(B) shows the binary addresses which are assigned to the elements which are contained within SUM-Graph depicted in FIG. 5(A). For example, logic condition L1 (k,b position 2,1) is assigned to binary address 1, logic condition L2 (k,b position 6,1) is assigned binary address 10, output statement W1 (k,b position 16,4) is assigned binary address 100111. Table 3 shows the binary address which correspond to the elements which comprise target process 250.

Table 3

SUM-Object	Set Member	(k,b) Position	Binary Address
[	1,1	1	
L1	2,1	1	
/7	3,1	10	
A1	4,1	10	
/1	5,1	10	
L2	6,1	10	
A2	7,1	100	
L3	8,1	100	
X1	9,1	1000	
*1	10,1	1000	
A3	9,2	1001	
/2	10,2	1001	
L4	11,2	1001	
A4	12,2	10010	
/3	13,2	10010	
X2	14,2	10010	
*2	15,2	10010	
/4	12,3	10011	
A5	13,3	10011	
/6	14,3	10011	
L5	15,3	10011	
A6	16,3	100110	
*3	17,3	100110	
W1	16,4	100111	
*4	17,4	100111	
A7	7,5	101	
L6	8,5	101	



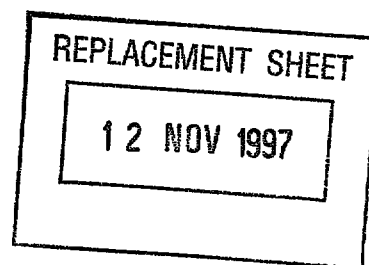
This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

Table 3 (continued)

R1	9,5	1010
*6	9,6	1011
E1	3,7	11
*7	4,7	11



Lastly, once SUM-Graph 262 construction is completed by SUM-Builder 259, the accessibility between two elements of target process 250 may be effectively determined. Accessibility between two elements s1, and s2, is defined as the presence of a path from s1 to s2. That is, s2 is accessible from s1 if there is some possible input data combination that results in s2 being executed some time after the execution of s1. Accessibility between two process elements is required if one element is capable of effecting the result of the second element.

Referring once again to FIG. 5(A), it becomes obvious that statement A7 (k,b position 7,5) is not accessible from statement W1 (k,b position 16,4). That is because there is no possible path from statement W1 that would proceed through statement A7. Statement A7 is however, accessible from statement X1 (k,b position 9,1). That is because statement *1 (k,b position 10,1) repositions the path taken to /1 (k,b position 5,1) which immediately precedes statement L2 (k,b position 6,1). The high potential solution of L2 leads directly to statement A7. Therefore, the present invention is able to determine whether statement(s) or elements of target process 250 could be influenced by the correctness of a particular statement or statements within that process. When the target process 250 is represented in SUM-Graph form as shown in FIG. 5(A) and is used subsequently for the construction of SUM-Model 243, the invention is capable of evaluating the possibility for a statement s1 of target process 250 to be responsible for the misbehavior of statement s2 of target process 250 by defining accessibility between s1 and s2 as a necessary condition for existence of cause-effect relationship between those two elements. Alternatively, the analyzer is capable of evaluating the potential for statement s2 to be effected by the modification of statement s1.

The accessibility between statements that comprise the SUM-Graph may be defined with respect to three potentials: 1) the free fall potential within a main branch; 2) the shift potential of a logic condition; and 3) the rewind potential of a terminal.

THIS SHEET HAS BEEN
REPLACED

This Sheet Has Been Replaced



12 NOV 1997

- 22 -

Free fall potential is determined by one process element becoming active after another unless the direction of execution is changed by a logic condition in a high potential state. Quite simply and with reference to FIG. 5(A), a free fall potential exists between arithmetic statement A1 (k,b position 4,1) and arithmetic statement A2 (k,b position 7,1) because A2 follows the logic condition if target process 250 fell through the logic condition L2 (k,b position 6,1) in its low potential state until terminal *1 (k,b position 10,1) is reached.

The shift potential is the potential of a logic condition to increment the SUM-Graph b coordinate as a result of a positive solution of the logic condition. With reference to FIG. 5(A), logic conditions L3 (k,b position 8,1), L4 (k,b position 11,2), L5 (k,b position 15,3) and L6 (k,b position 8,5) all have a shift potential of 1 because a positive solution to any of them results in a right shift by one of the b coordinate. Similarly, logic condition L2 (k,b position 6,1) has a shift potential of 4 and logic condition L1 (k,b position 2,1) has a shift potential of 6.

In order to appreciate the significance of the SUM-Graph structure and the information which the analyzer derives from its inventive structure, some additional definitions are required to facilitate understanding.

Definition G6: *bn will refer to the terminal with b-coordinate equal n. With reference to FIG. 5 *b3 refers to the GO TO terminal on the b=3 and *b5 refers to "exit" terminal on b=5.

Definition G7: Mbn will refer to a Main Branch built on the b-coordinate equal n. By way of example and with reference to FIG. 5, MB3 refers to the Main Branch built on coordinate b=3.

Definition G8: The shift potential of LC is the potential of LC to increment b-coordinate as a result of a positive solution of the LC.

Definition G9: The rewind potential of a terminal is the potential of that terminal change the next active Segment Address (SA).

Definition G10: The segment address (SA) is defined as a pair (BA,k), where BA is the binary address of the segment and k is a k coordinate within the segment. For example with reference to FIG. 5(A), the segment address of SUM-Object set member /3 is 10010,13. Further, segment addresses can be compared by the present invention in a manner similar to the comparison of binary addresses.

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

REPLACEMENT SHEET

12 NOV 1997

- 23 -

To compare binary addresses, the software analyzer left justifies the binary addresses, then corresponding bit position contents are compared from left to right until a mismatch occurs. When comparing binary address bit position contents, 1 > 0 > empty. By examining FIG. 5(A), 5(B) and Table 3, the comparison of binary addresses becomes apparent. The binary address of logic condition L2 (k,b coordinate 6,1 ; binary address 10) is greater than the binary address of logic condition L1 (k,b coordinate 2,1; binary address 1). The binary address of E1 (k,b coordinate 3,7; binary address 11) is greater than the binary address of L2 (binary address 10).

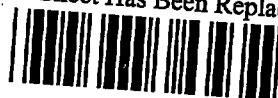
To compare Segment Addresses (SA), the binary address (BA) component of the SA is given greater priority, i.e., a segment address of 11,7 is greater than a segment address of 10,10 and segment address 11,7 is greater than a segment address of 11,6. Therefore, the segment address of L5 (FIG. 5) is greater than the segment address of A5 because their binary addresses are equal and the k value of L5 is greater than the k value of A5.

Definition G11: The vision of an element is the set of all elements which lie within any t-pass which passes through that element. An element s2 is within the vision of element s1 (or element s2 is seen from element s1) if the segment address (SA) of s2 is a descendant of the SA of s1. A statement s2 is within the vision of statement s1 if there is a path from s1 to s2 that is part of a t-pass.

Definition G12: A segment address of a statement is a descendant of the segment address of another statement if the binary address of the second statement is built from the binary address of the first statement by concatenating binary digits at the right, or if the BA(s1) is equal to the BA(s2) and the k coordinate of s2 is greater than the k coordinate of s1.

With reference to FIG. 4., A6 424 is within the vision of A3 412 since the segment address of 424 (100110) is a descendant of the segment address of 412 (1001). By way of an additional example, the binary address 1001 can be mapped onto 10011, therefore any element within segment 10011 is within the vision of an element within segment 1001. Further, element X2 418 (binary address 10010) lies within the vision of element A4 415 (binary address 10010) since they have the same binary address and the k coordinate of X2 (14) is greater than the k coordinate of A4 (12) when mapped to the SUM-Graph.

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

REPLACEMENT SHEET

12 NOV 1997

- 24 -

Definition G13: The vision of a rewind terminal is defined by the vision of its rewind entry. The vision of the exit terminal is empty. As an example, refer to FIG. 9. The vision of rewind terminal 1 930 is the vision of its rewind entry 912. The vision of exit terminal 960 is empty.

Definition G15: The access field of a process element is the combination of all the visions that the element successively owns. When an element sees a terminal, a vision of that terminal, which is the vision of its rewind entry may see some terminals which were not originally seen. And so on...therefore expanding the access field of the examined process element.

Definition G16: The rewind binary address (RBA) is the binary address of the rewind terminal entry. The rewind binary address of the exit terminal is 0. Referring to FIG. 5(A), the RBA of terminal *1 (k,b coordinate 10,1) is defined by its terminal entry /1 (k,b coordinate 5,1). Therefore the RBA of terminal *1 is the binary address of /1, which is 10. The position of rewind terminal entry is identified by a pair {BA,rk}, where rk is the k coordinate value of the rewind terminal entry. The rk value of the exit terminal is zero.

Definition G19: An open terminal is an exit terminal or rewind terminal that has access to an exit terminal.

Definition G18: A closed terminal is a rewind terminal that does not have access to an exit terminal.

Definition G17: The exit access indicator (EAI) is an indicator which is set to 0 for closed terminals and set to 1 for open terminals.

Definition G20: A closed region is a combination of adjacent main branches which are identified by closed terminals. Once a target process 250 enters a close region, that process cannot terminate. Referring to FIG. 9, terminals 944, 952 and 950 identify a closed region.

Definition G21: The propagated vision point (PVP) of a process terminal is defined by the vision of all the terminals that the process terminal sees. As previously defined, the vision of a terminal is the vision of its rewind entry. If the rewind entry of a first process terminal can see a second terminal whose rewind entry can see the rewind entry of the first process terminal, then the vision of the second rewind entry becomes the propagated vision point for both the first and second terminal.

THIS SHEET HAS BEEN
REPLACED

This Sheet Has Been Replaced



12 NOV 1997

- 25 -

By way of example and with reference to FIG. 10, if rewind entry /n 1001 of process terminal *n 1004 can see another terminal *i 1002 whose rewind entry /i 1000 can see the rewind entry /n 1001, then the vision of *n 1004 is propagated by the vision of *i 1002 and the vision of *i 1002 becomes the propagated vision point of both terminals *i 1002 and *n 1004. Therefore the propagated vision point of terminals *i 1002 and *n 1004, is the rewind entry of /i 1000.

Further, the terminal *k 1005 has a rewind entry /k 1003 which sees another terminal *n 1004. The propagated vision point of terminal *k 1005 is therefore propagated to the vision of terminal *n 1004, that had been already established as propagated to /i 1000.

The propagated vision point of a terminal is represented by a segment address of a specific rewind entry. If a terminal vision is not propagated by the visions of other terminals, then the propagated vision point of that terminal is its own vision, i.e., the vision of its rewind entry.

When all of the terminals within the vision of a rewind entry have that entry as their propagated vision point, the result is a closed region (Data Independent Endless Loop). This is because none of the terminals within the vision of that rewind entry have had an access to an exit terminal since the vision of an exit terminal would have been zero.

Once the SUM-Graph 262 is constructed, the exit access indicator (EAI) table is constructed. FIG. 9 shows a sample EAI table constructed from the terminal binary address, the rewind binary address and rk of each process terminal, all of which are known once the SUM-Graph is constructed.

FIG. 9 shows an EAI table which is constructed according to the following method. First, the original value of the propagated vision point of a terminal is built from that terminal rewind entry, that is, the rk and RBA of the terminal. Second, the EAI of all rewind terminals are set to 0 (set closed) and the EAI of all exit terminals are set to 1 (set open). Lastly, and proceeding from $b = 1$ to the maximum b of the SUM-Graph, each terminal PVP is compared with the Binary Address and PVP of every other terminal in an attempt to propagate (decrement) the PVP of the current terminal under examination and attempt to open that terminal if closed. If the PVP of a closed terminal can see any open terminal, then the examined terminal is defined as open by setting its EAI to 1.

This Sheet Has Been Replaced

THIS SHEET HAS BEEN
REPLACED

REPLACEMENT SHEET

12 NOV 1997

- 26 -

By way of further explanation and with reference to the EAI table shown in FIG. 9, the construction of the EAI table proceeds as follows:

1) Set the PVP of terminals to the segment address of their rewind entries:

BA of PVP of terminal *b1 (PVP*b1) is set to 10;
BA of PVP*b2 is set to 1001;
BA of PVP*b3 is set to 10010;
BA of PVP*b4 is set to 10011;
BA of PVP*b5 is set to 0, since *b5 is an exit terminal;
BA of PVP*b6 is set to 10011;
BA of PVP*b7 is set to 10;

2) Propagate the PVP and Set the EAI. If the PVP of a closed terminal can see any open terminal, then the examined terminal is defined as open by setting its EAI to 1.

a) PVP*b1 {10,rk1} sees *b2,*b3,*b4,*b5, but neither of those terminals PVPs can see PVP*b1. Therefore
PVP*b1 is not changed; PVP*b1 {10,rk1} sees open

*b5 (1010);

open *b1: set EAI*b1 to 1.

b) PVP*b2 {1001,rk2} sees *b3,*b4 whose binary addresses are 100110 and 100111, but neither of these terminals can see the PVP*b2. Therefore,

PVP of *b2 is not changed;

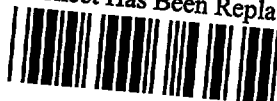
PVP*b2 {1001,rk2} does not see any open terminals;
EAI*b2 is kept 0;

c) PVP*b3 {10010,rk3} sees *b2 (10010) and PVP*b2 {1001,rk2} sees PVP*b3, that is, {10010, rk3}. Therefore,
set PVP*b3 to PVP*b2: {1001,rk2};
propagated PVP*b3 still ~~does~~ not see any opened

terminals;

EAI*b3 is kept 0;

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

REPLACEMENT SHEET

12 NOV 1997

- 27 -

d) PVP*b4 {10011,rk4} sees *b3 (100110), and PVP*b3 {1001,rk2} can see PVP*b4 {10011,rk4};
set PVP*b4 to {1001,rk2};
propagated PVP*b4 still does not see any open terminals;
EAI*b4 is kept 0;

e) PVP*b5 is not changed, since it is an exit terminal;

f) PVP*b6 {10011,rk6} sees *b3 and *b4, whose current PVPs {1001,rk2} can see PVP*b6;
set PVP*b6 to {1001,rk2};
propagated PVP*b6 still does not see any opened terminals;
EAI*b6 is kept 0;

g) PVP*b7 {10,rk7} can see terminals from *b1 to *b6, but neither of them can see PVP*b7;

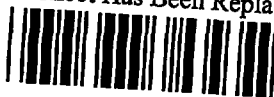
PVP*b7 is not changed;
PVP*b7 {10,rk7} can see open terminals (1000 and 1010);
open *b7: set EAI*b7 to 1;

In some target processes and with reference to FIG. 10(B), terminal *i 1006, that is propagated by the vision of *n 1007, is examined before the PVP of *n 1007 is finally set, since the PVP of *n 1007 may in turn be upgraded by the vision of the other terminals. Therefore, the only way to make sure, that all the possible upgrades of PVP and EAI were performed, is to repeat the step (2) of examining all the terminals in the order from *b1 to *bn once again. If no upgrades were done within the last pass, the process of upgrading the PVPs and EAIs is finished.

In the worst case, the number of required passes through the terminals, i.e. executions of step (2) is equal to the number of Main Branches, NB, since each pass will finally set the PVP and the EAI of at least one terminal. In the above example all upgrades were performed in one pass, and the second pass through *b1 to *b7 will not change any PVP or EAI.

Once the exit access indicator table is constructed, Propagated Vision Points built into this table allow one to define the accessibility between any two elements of the graph. As a consequence, all data independent endless loop constructs are located, as those constructs do not have accessibility to an exit terminal.

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

12 NOV 1997

- 28 -

It is commonly accepted that the question of whether or not a program will terminate is proven to be unsolvable. We can divide this question of process "terminability", i.e. ability to terminate, into two categories: 1) data independent endless loops and 2) data dependent endless loops.

A data independent endless loop consists of process elements that do not have an access to "exit" terminals. A data dependent endless loop is such a construction that after entering it the control will never reach an exit (STOP or RETURN) by the reason other than the absence of control passes to these exit terminals. The reason in this case is either: a) the absence of the manipulation of program variables, that could potentially change the control pass, on the specific control passes, or b) change of these program variables in the direction that takes the values of LCs further from changing their states between Low Potential State and High Potential State.

In both cases a and b above, the control will fall into an endless loop, going through the same terminals in the same sequence, without ever being able to reach an "exit" terminal.

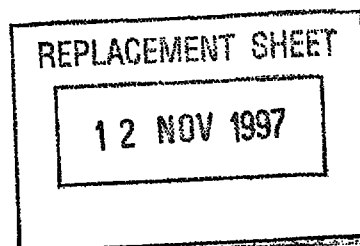
As an example of data dependent endless loop and with reference to FIG. 9, if there are no elements on the pass between /1 912 and *1 930 that can redefine the values, being examined by L2 914 and L3 924, then terminal *b1 930 will produce a data dependent endless loop as soon as L2 914 and L3 924 will be resolved by their Low Potential solutions the first time (since the state of L2 914 and L3 924 will never change during the forward execution of the target process.)

As an example of a data independent endless loop and with reference to FIG. 9, a data independent endless loop construction is presented by Main Branches MB2, MB3, MB4. If L3 924 is ever solved positively, the process will go in the infinite loop, no matter, what data manipulation statements are coded within those branches.

While process constructions that could allow for data dependent endless loops present a potential for process "non-terminability", they are not necessarily faulty constructions, since it could be so, that an input data can never be such, that would set LCs in the state of infinite loop.

This Sheet Has Been Replaced

THIS SHEET HAS BEEN
REPLACED



Entry /q 922 in FIG. 9 does not correspond to any terminal and is shown only as an example of the fact, that the position of L3 924 in the negative subfield of L2 914 with rewind terminal *b1 rewinding the process control directly in front of L2 914 is not necessarily a fault, even if L2 914 and L3 924 are not dependent on the data manipulations that are coded on the path /1 912 to *1 930. This is, because that if L3 924 had received control through the entry /q 922, returning the control to the entry /1 912 will not necessarily result in L2 914 being solved negatively thus in creating an endless loop.

On the other hand the programs that contain Data Independent Endless Loops constructs are definite faults in the target process construction.

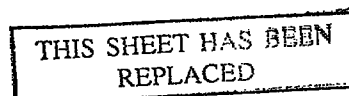
The detection of data independent endless loops is done by evaluating an accessibility between rewind terminals and exit terminals as it was explained on the previous example. The same process would allow evaluation of accessibility between any two elements of SUM-Graph (for example between A2 and A7 in FIG. 9).

As a result we see: EAls of terminals *b2, *b3, *b4 and *b6 are 0, these terminals are closed. Terminals *b2,*b3,*b4 constitute a closed region (data independent endless loop). Terminal *b6 is also closed and its rewind entry is within the mentioned closed region [*b2 - *b4]. Dotted line 966 represents the vision field of an entry /1 {10,rk1}. This entry is PVP of *b1. This means, that any element of Main Branch b=1 can access any element within the vision of {10,rk1}.

An access field of A2 is represented by the dotted line 966. Dotted line 967 represents the vision field of an entry /2 {1001,rk2}. This entry is PVP of *b2, *b3, *b4 and *b6. This means, that any element of Main Branch MB2, MB3, MB4 and MB6 can access any element within the vision of {1001,rk2}.

By definition, an access field of a process element is the combination of all the visions it successively owns. Therefore, an access field of A7 is combined by the fields represented by dotted line 964 and dotted line 967, as dotted line 964 is the vision of A7 and dotted line 967 is the vision of *b6 that is seen from A7.

From FIG. 9 we can see, that A2 can access A7, and A7 cannot access A2.



12 NOV 1997

- 30 -

As an example from FIG. 9, examining the attributes (BA, RBA and PVP) that are built for every terminal, we can determine the following: a) A2 can not see A7, as 101 is not seen from 100. However, A2 can access A7, since one of the terminals, seen from A2, *b1, can see A7 through its PVP {10,rk1}, as 101 is a descendent of 10; b) A7 can not see A2, as 100 is not a descendent of 101; the terminals that are seen from A7 are ones with BAs 1010 and 1011; Terminal 1010 has no RBA or PVP, being an exit terminal; terminal 1011 has PVP {1001,rk1}, which can not see 100 either. Therefore A7 can not access A2.

Data dependent endless loops could be determined using our method of propagated vision points in the following manner. Once the sequence of terminals executed repeats many times it could become useful to locate the propagated vision point of those terminals and then to see whether or not there are any data manipulation statements on the t-passes from this PVP to those terminals which data manipulation statements affect the variables checked by the logic conditions located on the t-passes from this PVP to those terminals. If no such statements exist, we have a data dependent endless loop. If such statements exist but are constructed in a way such that the variables change in the direction that would take the value calculated by the logic condition further from the value that would change the status of the logic condition. In this case we also have a data dependent endless loop.

Determining Process Correctness

As a result of the structures which are constructed by the SUM-Analyzer it is possible to evaluate the correctness of a target process. In order for those skilled in the art to readily appreciate how the SUM-Analyzer operates, the following definitions are necessary at this point.

Definition P1: The symbol (s), represents a process element or individual statement of a program.

Definition P2: The Process Address, (PA) {k,b} of an element is the position of the element within the reduced flow chart of the process or the position of the element within SUM-Graph of that process. With reference to FIG. 5, the PA of an element is defined by the k,b coordinate pair which describes the position of the element within the reduced flow chart. With reference to FIG. 5, element A2 has a PA of 7,1. Each statement that comprises a process has a unique PA.

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

12 NOV 1997

- 31 -

Definition P3: The local time $\{p\}$ parameter is an integer value which represents the present occurrence of a PA within a process execution. Local time is started with the initial invocation of the process. The local time can only be measured at a PA. The SUM-Analyzer does not utilize time as it is normally used, i.e., a measurement of duration. Instead, time as recorded by the SUM-Analyzer is a measurement of the number of occurrences of a particular statement. Therefore each event or occurrence of a statement execution is identified by its own time parameter.

Definition P4: A process event is an event of a process statement execution. A process event is defined by $\{k,b,p\}$, where k,b are the coordinate pair which identify the process address and p is the local time within this process address.

Definition P5: The Process time is represented by process events $\{k,b,p\}$. As an example and with reference to FIG. 5(A), when statement A5 is executed the third time, that moment of execution is referred to as the $\{13,3,3\}$ event.

Definition P6: A process event $\{k,b,p\}$, other than an output event, is correct if that event takes part in a chain of events that effect a correct output event. An output event that is an event of execution of system output statement is correct if it produces expected output result.

Definition P7: A Correct Process Address (CPA) contains a process element (s) that had been executed as a correct event.

Definition P8: An Uncertain Process Address (UPA) contains a process element (s) that has not yet been executed as a correct event.

A state of PA as UPA or CPA, is relative to some other time in the process. This state could be changed at some process time $\{k_e,b_e,p_e\}$ which is an event of execution of some other process address $\{k_e,b_e\}$ which event could define (s) event execution as correct or could make it uncertain.

As an example, consider a process defined by the chain of events $e_1, e_2, e_3, e_4, e_5, e_6$ and e_7 where e_7 is a correct output event, e_1 and e_3 are input events and e_1, e_2, e_3, e_4 and e_6 were involved in producing the correct output event e_7 . Therefore, events e_1 through e_7 except e_5 are correct. Event e_5 may be defined as correct by some other correct output event chain in which e_5 took place, but until such a time its correctness is uncertain.

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

12 NOV 1997

- 32 -

Definition P9: The algorithm space of a process is the space defined by the combination of all process addresses $\{k_i, b_i\}$. With reference to FIG. 5, the algorithm space of the target process described by the reduced flow chart is bounded by the maximum k and maximum b shown.

Definition P10: The correct algorithm space of a process is the space of all process addresses which are presently defined as correct process addresses.

Definition P11: The uncertain algorithm space of a process is the space of all process addresses which are presently defined as uncertain process addresses.

In both of the above definitions the correctness or uncertainty of a space is relative to the present time within the process. This is due to the fact that the SUM-Analyzer has the ability to later redefine a correct event as uncertain (losing the knowledge). The process address may become uncertain if it does not any longer have a history of being executed as a correct event.

Definition P12: The event definition space of an event (e) consists of process addresses of the event's ancestor events.

Definition P13: An ancestor event is a prior event which took part in producing a later event by either: 1) producing a value of the later event (value ancestor event) or 2) producing a parameter for a conditional control event, i.e. an event of executing a conditional control statement (L or D of the SUM-Object Set) in a path leading to the later event (control ancestor event). If either of these conditions are present, then the later event is a descendent of the prior event.

By way of example and with reference to FIG. 14, $We(a)$ 1400 is a descendent of the events shown below $We(a)$ and within its branches. Further, $e(d)$ is a descendent of $e(i)$ 1410, $\#e(j)$ 1413, $e(k)$ 1414, $*e(l)$ 1415, $*e(m)$ 1418 and the other events that are the ancestors of $\#e(j)$ 1413, which are $\#e$ 1417, $\#e$ 1419, $\#e$ 1420, $\#e$ 1421.

The SUM-Analyzer operates by preserving the event definition space of correct events which, in turn, preserves the elementary knowledge of process element execution, i.e., the correctness or uncertainty of this event. The SUM-Analyzer continuously evaluates the event definition space of new output events and expands the correct algorithm space of a target process 250, when possible. On the other hand, when evaluating the event definition space of

This Sheet Has Been Replaced

THIS SHEET HAS BEEN
REPLACED

12 NOV 1997

- 33 -

output events found to be incorrect, it may lose previously built knowledge of correct algorithm space by losing the status of some CPA to UPA.

The relationship between A,L,D,E,X,*,+,C,!,/,R,=,S and W elements of the SUM-Object Set are such, that an approved event of a W element execution activates a chain of secondary processes. These secondary processes thereby approve the execution events of those SUM-Graph elements that were involved in producing the approved event.

As a consequence of the principles we are using, the fault(s) responsible for the failure of an output result will be located within that part of the output result event definition space that lies within the area of uncertain algorithm space. Likewise, modifications made to a program as a result of new system requirements must be located within that part of the output result event definition space that lies within the area of uncertain algorithm space. If there is no uncertainty within the event definition space of the results that must be modified, then the new branches should be built in order to preserve correct algorithm space.

Definition K1: The approval of an event is the process of validating that events correctness by the SUM-Analytical Engine.

Definition K2: A control event is any event which may potentially change the direction of a process from falling down the main branch.

Definition K3: An unconditional control event is an execution of rewind or exit terminals as defined in the SUM-Object set by {*,+,C,R,=}.

Definition K4: A conditional control event is an execution of a logic condition (L or D elements of the SUM-Object Set).

A positive solution of a conditional control event results in a control shift of a process to the right as viewed within an SUM-Graph or reduced flow chart. A negative solution to a conditional control event results in the direction of a process to proceed downward as viewed within an SUM-Graph or reduced flow chart.

Definition K5: A data manipulation event is an execution of an (A) or (E) or the arithmetic part of a (D) element of the SUM-Object Set.

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

12 NOV 1997

- 34 -

Definition K6: An initial data manipulation event" is one of the following: E-event, A-event of data initialization or D-event of the loop index initialization.

Definition K7: A Secondary Data Manipulation event is any data manipulation event other than an initial data manipulation event.

Definition K8: An elementary knowledge of a process element located within $PA\{k,b\}$ is obtained, when one of its execution events is approved. Accumulating knowledge about a process is performed by accumulating information about the correctness or uncertainty of the process element's execution events.

Definition K9: Elementary knowledge propagation is a process by which a elementary knowledge of an event is applied to different process addresses. Elementary knowledge propagation is performed by locating and analyzing all of the ancestor events up to a knowledge propagation terminal events.

Definition K10: A knowledge propagation terminal (KP terminal) is defined as an event that either has no ancestors within the target process (i.e., an initial data manipulation event) or is a previously approved event.

Terminating knowledge propagation at previously approved events stabilizes the process of knowledge propagation and prevents the knowledge propagation process from becoming increasingly complex with the accumulation of ancestor events during the forward execution of a target process. When a knowledge propagation process encounters a previously approved event, all of the ancestor events of the approved event were previously approved as well.

FIG. 14 shows events $We(a)$ 1400 and $We(n)$ 1401 above a tree of their ancestors. The process direction is indicated within FIG. 14 by arrow 1422, therefore event $We(a)$ 1400 occurs at some time before event $We(n)$ 1401. In this example, $*e$ represents a KP terminal event, that is an initial data manipulation event and $\#*e$ represents a KP terminal that is a previously approved event.

In FIG. 14, the value of $We(a)$ 1400 was calculated from three parameters whose three values were produced by the three events $e(b)$ 1402, $\#*e(c)$ 1403 and $e(d)$ 1404, respectively. Further, a value associated with $e(b)$ 1402 was calculated from two parameters whose two values were produced by the two events $*e(f)$ 1407 and $\#*e(g)$ 1408 respectively. Event $*e(f)$ 1407 does

This Sheet Has Been Replaced

THIS SHEET HAS BEEN
REPLACED

12 NOV 1997

- 35 -

not have any ancestors, since it is an initial data manipulation event and is therefore a KP terminal. Event #*e(g) 1408 was previously approved (as indicated by #) and therefore it is also a KP terminal.

There is no attempt by the SUM-Analyzer to locate the ancestors of a KP terminal. This is because such ancestors either: 1) are not present (as is the case with *e(f) 1407) or 2) have been previously approved (#*e(g) 1408).

With reference to FIG. 17, event e(0) 1700 is shown as occurring at some process time (k0,b0,p0). Further, if event e(0) 1700 were the first approved event of s0 1701, then the PA of s0 1701 is upgraded from UPA to CPA as indicated by arrow 1710. This example shown in FIG. 17 indicates that a value of variable y is a function of three other parameters, x1 1702, x2 1703 and x3 1704.

The immediate ancestors of s0 1701 are the events which produced the value of the parameters involved in the s0 1701 calculation. The value of the parameters of s0 1701, namely x1 1702, x2 1703 and x3 1704 are determined by ancestor events #*e1 1705, #*e2 1706 and e3 1707. If events #*e1 1705 and #*e2 1706 were previously approved as shown, they are treated as KP terminals and we do not examine their ancestors.

Event e3 1707, which calculates parameter x3 1704 was not previously approved. The CPA status 1708 which was a state of the process address of statement s3 1709 during the process time (k0,b0,p0), means that some other event(s) of this PA execution was previously approved. Therefore, it is necessary to examine all of the ancestors of the event e3(k3,b3,p3) 1707 until a KP terminals are met during such examination on each direction.

Definition K11: The immediate owner of an event is another event which approved the event.

Definition K12: A W-owner of an event is a descendent of the event which originated the chain of elementary knowledge propagation that approved the event.

Each event has only one W-owner event and only one Immediate Owner event. In some situations, the W-owner of an event and the immediate owner of an event are one and the same.

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

12 NOV 1997

- 36 -

As an example of event approval and with reference to FIG. 16, a process address (k,b) 1600 is shown during five different process times t1 1605, t2 1606, t3 1607, t4 1608 and t5 1609, respectively. These process times are defined by the execution of events at some address or addresses that could be other than (k,b). At time t1 1605, the PA 1600 had no occurrences of its execution. At time t2 1606, there has been two occurrences of PA 1600, e1 1610 and e2 1611, neither of which has been approved. At time t3 1607, there has been three occurrences of PA 1600, e1 1610, #e2 1611 and e3 1612. Further, event #e2 1611 has been approved during the interval which occurred between t2 1606 and t3 1607. The status of Process address 1600 therefore, is known to be CPA at time t3 1607. The W-owner attribute of CPA helps to maintain the current knowledge of the target process 250. When a previously approved W-event is disapproved, the target process knowledge is dropped to the level determined by the position of the disapproved W-owner event in the target process time.

The process of approving an event generates a secondary process which approves the immediate ancestors of the event unless the immediate ancestors are already approved or do not exist. Therefore, the propagation of elementary knowledge of the process continues in the direction of all ancestors until a knowledge propagation terminal encountered in each direction.

Building the SUM-Model of the Target Process

After the SUM-Builder 259 finishes construction of the SUM-Graph previously described, an SUM-Model 243 of target process 250 is built. The invention constructs a different SUM-Model for each distinct software process analyzed where the process is procedure i.e., program, subroutine, function, paragraph. Although the preferred embodiment of SUM-Model 243 discussed in this detailed description is a three dimensional model, those skilled in the art could readily adapt the model of the present invention to additional dimensions. Two of the three dimensions on which the SUM-Model is constructed have already been described, namely the k,b coordinates of the SUM-Graph 262.

The third dimension of the constructed SUM-Model comprises a structure, that is populated with attributes of process addresses. The process address of an element is the position (k,b) of that element within the reduced flow chart, or alternatively within the SUM-Graph 262 of the target process 250. Table 4 is an example of the implementation of the structure constituting the third dimension of SUM-Model for the elements of the SUM-Object set *, C or +.

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

12 NOV 1997

- 37 -

Table 4

Attributes

* C +	ind	re-wind ID	loc.time	BA	tar.rec#	rep.rec#
SUM-Srec#	RBA					

Different attributes are assigned to different process addresses, depending upon the type of process element (i.e., A,L,D,W,E) that is positioned at this process address. Referring to Table 4, one member of the structure is populated with the type of process element. Another member of the structure contains ind, which is the index of the corresponding process element in the SUM-Frame and SUM-Graph.

Each element of the SUM-Model corresponds to an executable statement and contains three distinct attributes which establish a relationship between the target process 250, the SUM-Model 243, the SUM-Repository 246 and SUM-Source 247. Those attributes providing this relationship are: 1) a record number of the corresponding statement within the target process; 2) a record number of the corresponding statement within the SUM-Repository; and 3) a record number of the corresponding statement within the SUM-Source. Referring once again to Table 5, these three attributes are present in the structure members.

For each executable process element, the SUM-Model keeps an attribute of local time. Local time is an integer value of the occurrence of a statement within a target process. For example, the first time that a statement is executed during a program execution, the local time would be 1. The second time that the same statement is executed during a program execution, the local time would be defined as 2. The local time attribute is present as a structure member of the sample SUM-Model structure shown in table 4.

The binary address attribute is assigned to each process address of the SUM-Model. In Table 4, the binary address is shown as a structure member.

Structures representing rewind terminals *,+,C also possess an attributes of Rewind ID and Rewind Binary Address that represent the Process Address and BA of the position in the SUM-Graph, that will be active after that statement execution.

This Sheet Has Been Replaced

THIS SHEET HAS BEEN
REPLACED

12 NOV 1997

- 38 -

The ID value is an alternative representation of (k,b) coordinate pair. Each process element represented by a (k,b) coordinate pair in the SUM-Graph could be alternatively represented by one integer value ID, where the integer is the numerical order of the element as it is encountered during a traversal of the two dimensional table on which the SUM-Graph is built in order of traversing columns, left to right. The ID of any particular element with process coordinates {k,b} could be defined by the following formula:

$$ID = ((b-1) * k_{max}) + k$$

where k_{max} is the maximum value of the k dimension represented on the SUM-Graph.

From the other side knowing the ID of element s {k,b} and knowing the k_{max} of the model, one could define {k,b} by the following formulas:

$$b = (ID - 1) / k_{max} + 1$$

$$k = ID - (b-1) * k_{max}$$

Referring to SUM-Graph shown in FIG. 5(A), graph element *1 (k,b coordinate 10,1) has an ID of 10. Element L6 (k,b coordinate 8,5) has an ID of 76.

Table 5 shows the representative SUM-Model structure for a process element represented by the following SUM-Object set members, R and =

Table 5
Attributes

R =	ind	loc.time	BA	tar.rec#	eXS.rec#
-----	-----	----------	----	----------	----------

Exit terminals, when represented in the SUM-Model, and in Table 5 do not contain the following attributes of rewind terminals; Rep.rec#. Rewind ID and Rewind BA.

Process addresses containing process elements represented by the SUM-Objects A,L,D,E,W,X and S each have an ACB Pointer attribute. The ACB Pointer attribute is a pointer to a corresponding Analyzed Calculation Base construction that contains information on the history of approved and uncertain execution events of the current process elements in relation to the knowledge of

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

12 NOV 1997

- 39 -

their effects correctness.

In addition, structures representing L and D elements have an additional attribute which represents the increment in the b coordinate when traversed through their positive, or high potential solution. This corresponds to the shift potential previously described with respect to the SUM-Graph. For example, element L1 (k,b position 2,1) shown in FIG. 5(A) would have a shift potential of 6 recorded in its shift potential attribute field.

The Analyzed Calculation Base (ACB) 273 is a collection of structures, each corresponding to a different elements (statements) of the target process 250. The information recorded within the ACB indicates the knowledge of correctness or its uncertainty of the effects of the different process address execution events at different times within the process.

FIG. 19 shows a number of ACB structures with each ACB construction corresponding to an executable process address of the target process 250. Every member or element of an individual ACB structure represents an event of a target process element execution.

In a present embodiment, the ACB pointer element of the SUM-Model previously described points to a binary file that is divided into two parts. The first part of the ACB structure is a control area that contains two distinct fields; Next Parameter Word (NPW) and Uncertainty Displacement Word (UDW). The NPW is only used for elements that have parameters, i.e., E and X. Referring once again to FIG. 19, an ACB structure for element A1 is shown as 1900. This ACB structure contains three fields, 1901, 1902 and 1903. The NPW is shown as 1901 and the UDW is shown as 1902.

The second part of the ACB structure, shown in FIG. 19 as 1903, is the information area which stores knowledge state bits. Each execution event of the element that is represented by this ACB structure is represented by a bit in the information area 1903. The position of a particular bit within the information area corresponds to the local time of the event. That is to say, the 20th bit of the information area 1903 corresponds to a local time of 20, or the 20th time that particular element was executed. The state of the binary digits (0 or 1) comprising the information area correspond to the current knowledge of the events effects correctness, where 1 represents a correct effect of an event and zero corresponds to an uncertain effect of an event. When an event is represented by zero in the ACB structure existing at some process time, it means that at that process time, the effect of the examined event is not known to be correct.

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

12 NOV 1997

- 40 -

Those skilled in the art can readily appreciate that the monitoring of a target process 250 can generate enormous volumes of data even when recorded in a structure as simple as the ACB. While many forms of compression could be used, the present implementation UDW 1902 stores the relative displacement of the first bit that is in a zero state. For example, UDW 1902 contains the value 2000. This represents that the first bit position stored in information area 1903 is for a local time value of 2001 and that all 2000 previous events of that element execution were approved or disregarded as a blank events, i.e., events which did not produce any results which were used subsequently.

Each parameter of an input statement (E) is represented by a ACB structure. Referring once again to FIG. 19, two parameters of an input statement are shown represented by two ACB structures 1905, and 1906. The NPW of 1905 points to structure 1906.

Similarly, each parameter of a call statement (X) is represented by an ACB structure. FIG. 19 shows three ACB structures 1907, 1908 and 1909 which correspond to three parameters of the call statement. The NPW of 1907 points to structure 1908. The NPW of 1908 points to 1909. The NPW of 1909 is empty since there is no next parameter in this CALL() statement.

Building the SUM-Repository

The SUM-Repository 246 of target process 250 is built at the same time as the SUM-Graph 262 or the SUM-Frame 261. Each element of the main branch of the target process that corresponds to an executable statement, i.e., represented by SUM-Object set members A, L, D, E, W, X and S, is stored in the SUM-Repository 246.

The size of the SUM-Repository 246 is less than the corresponding target process source code. This is because the SUM-Repository 246 does not contain any process control statements, therefore the SUM-Repository does not contain any unconditional branches, stop, return, then, else, endif, etc. The control structures are instead, preserved by the SUM-Builder 259 in the SUM-Model 243. To demonstrate how particular statements of target process 250 are represented in the SUM-Repository, consider the following examples.

If target process source 250 contained the statement:
if (d < 10) go to label

THIS SHEET HAS BEEN
REPLACED

This Sheet Has Been Replaced



REPLACEMENT SHEET

12 NOV 1997

- 41 -

The SUM-Repository 246 would contain the statement:

```
if (d < 10)
```

Alternatively, if target process source 250 contained the following statements:

```
if (a .gt. b) then
  c = d
else
  e = f(g)
endif
return
```

The SUM-Repository 246 would contain:

```
if (a .gt. b)
  c = d
  e = f(g)
```

Each statement of the SUM-Repository is referenced by the repository record number (rep rec.#) attribute of the corresponding process address within the SUM-Model. For example if the SUM-Repository is kept as a Relative Record Data Set (RRDS) then the repository record attribute of the process address within the SUM-Model must contain the relative record number of the corresponding statement within SUM-Repository 246.

Building the SUM-Source Code

Subsequent to the construction of the SUM-Graph 262, the SUM-Model 243 and the SUM-Repository 246, SUM-Source 247 is generated by the SUM-Source generator 249. This step is only done if the SUM-Analyzer is implemented in an embodiment which uses traditional object code.

The SUM-Source 247 provides a synchronization mechanism between the target process 250 and the SUM-Analytical Engine 244. This synchronization mechanism is implemented through the use of synchronization signals, which are inserted within the SUM-Source 247, by the SUM-Source Generator 249. In addition to the inserted synchronization signals, the SUM-Source 247 may differ from the target process 250 in that the SUM-Source 247 will not contain any unreachable or "dead" code.

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

12 NOV 1997

- 42 -

The SUM-Source code 247 is built by the SUM-Source Generator 249 from two predecessor structures, the SUM-Repository 246 and the SUM-Model 243. Since the SUM-Source is built from the SUM-Model and the SUM-Repository which were built from the target process 250, there exists a functional equivalence between the target process 250, the SUM-Model 243 and the SUM-Source 247.

During the creation of the SUM-Source 247 by SUM-Source generator 249, the following steps are performed by the SUM-Source generator: 1) the contents of each executable statement which comprises the SUM-Source is taken from the SUM-Repository 246; 2) the control statements contained within the target process 250 which are now represented within the SUM-Model 243 are extracted from the SUM-Model; and 3) the target process 250 - SUM-Model 243 synchronization signals are inserted into the SUM-Source 247.

Those skilled in the art can readily appreciate that the synchronization signals can be implemented through a variety of means. The synchronization signals inserted into the SUM-Source may consist of CALL() or WRITE() statements, which should be immediately familiar to those skilled in the art. The inserted synchronization signals provides the SUM-Interface 263 with three parameters. The three parameters are defined as (k,b,m) where k,b are the k,b coordinates, that define the process address, and m defines the model of the target process.

Alternatively, this SUM-Interface 260 could be implemented with only two parameters. For example, instead of the (k,b,m) parameters one could pass only (ID,m) where ID is the unique identification of the element's process address which is calculated from the k,b coordinate as previously described.

The synchronization signal is inserted into the SUM-Source 247 by the SUM-Source Generator 249 during the construction of the SUM-Source for each statement representing a process terminal. As an example, consider the reduced flow chart shown in FIG. 4 which represents target process 250. The flow chart shows a terminal 411 at k,b coordinate 10,1 which represents a FORTRAN GOTO statement contained within target process 250.

When the SUM-Source Generator 249 constructs the SUM-Source for terminal 411 the following program statements are inserted within the program code:

```
call SUMI(10,1,1)
GOTO 'LABEL a'
```

This Sheet Has Been Replaced

THIS SHEET HAS BEEN
REPLACED

12 NOV 1997

- 43 -

The call SUMI() statement is made with three parameters as identified above. By way of an additional example, consider terminal 431 shown in FIG. 4. When the SUM-Source Generator constructs SUM-Source for terminal 431, the following program statements are inserted within the program code:

```
call SUMI(9,6,1)
GOTO 'LABEL e'
```

When CPU 241 executes either of the above synchronization signals, the three parameters are passed to the SUM-Interface 263. When the SUM-Interface receives the signal, it records the process address or ID of the calling terminal and the model identifier (m) within the system execution history 253.

While the present embodiment passes the three above parameters with the synchronization signal, an alternative embodiment would pass only two parameters with the synchronization signal. Since the element ID characteristic previously described uniquely identifies a process element and the ID is derived from the k,b coordinates of the process element, the process ID could be passed as a parameter to the synchronization signal instead of the k,b coordinate pair.

When implemented in this manner, the sequence of executed process terminals is sufficient information to allow the analyzer to recreate the target process 250 behavior for any given model of a target process.

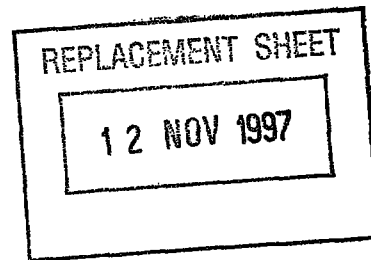
This is because, within the model, there is only one pass between the START and a specific terminal or between an entry of a Rewind ID of a Rewind ID of the previous terminal and the next terminal. The Rewind ID has been stored as an attribute of each terminal represented within the SUM-Model 243.

The terminal which sent the synchronization signal is then located by the SUM-Interface in the SUM-Model through its k,b address.

When SUM-Interface 263 working in the traditional object code embodiment of Software Understanding Machine is signaled and the parameters associated with the signal are sent, the SUM-Interface invokes SUM-Analytical Engine 244 which retraces on the SUM-Model a part of the pass executed within the target process 250. The part retraced is from the Rewind ID of the previously executed terminal (or from the START, if the currently executed terminal is the first since the process was entered) to the current terminal. Each

This Sheet Has Been Replaced

THIS SHEET HAS BEEN
REPLACED



pass through a process address represents an execution event for that process address and the local time of that address is incremented by one.

By way of example, consider the execution where the following three CALL() instructions were made to the SUM-Interface during the course of an analysis.

```
CALL SUMI(10,1,1)
CALL SUMI(10,1,1)
CALL SUMI(9,5,1)
```

With reference to FIG. 5(A), these signals correspond to the execution of terminals mb1, mb1, and mb5 respectively. From this information, the analyzer determines that the target process 250 was executed through the path identified by the following elements in this order Entry, L1, /7, A1, /1, L2, A2, L3, X1, *1, L2, A2, L3, X1, *1, L2, A7, L6, R1 (Exit).

To represent repetition of the same terminal, as is often the situation with control loops structures comprising DO, FOR, While, etc., a negative number notation is used. For example, a notation 27 -5 42 would mean that a loop was passed through 6 times (reached the terminal with IDT=27) before exiting from the loop.

As may be appreciated by those skilled in the art, the synchronization statements could have been invoked after every element or after every exit of a logic condition. Our preferred embodiment minimizes the number of signals utilized to effectively represent the target process.

These above examples demonstrate that recording the target process execution history by recording the sequence of the executed process terminals is sufficient and the absolute minimum recording required to characterize the target process 250 dynamics.

In addition to the synchronization signal previously described that synchronize the execution (retracing) of SUM-Model necessary in this embodiment with the execution of the target process, two other types of synchronization signals are inserted into the SUM-Source by the SUM-Builder. Those other two signals allow the construction of the incremented system output process synchronized map and optionally additional synchronization between the target process and SUM-Model execution processes. Those two types signals are inserted immediately following system output statements (W) and immediately before system input statements (E).

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

12 NOV 1997

- 45 -

Synchronization of the system output does not invoke the SUM-Analytical Engine. Process synchronization at the system input is optional. Process synchronization at the system input, when performed, invokes the SUM-Analytical Engine and is performed in our preferred embodiment only because it is a convenient place to catch up with the target process by the SUM-Analytical Engine since the system input interrupts the target process anyway by the operator or user "think time".

With reference to FIG. 21A and FIG. 21B the entry for the Incremental System Output Process Synchronized (ISOPS) map corresponding to the system output 2151 will be performed by the call to the SUM-Interface SUMI 2152. Process synchronization at the system input 2154 is performed by 2153, process synchronization at the terminal 2161 is performed by 2160, at the terminal 2159 by 2158 and at the terminal 2157 by 2156. Entry for the System Input Process Synchronized (SIPS) map corresponding to the system input 2154 will be recorded by 2155.

Incremented System Output Process Synchronized Map

The incremented system output process synchronized map (ISOPS) provides a mechanism to associate an output of a target process with the process events which produced that output. The ISOPS map is constructed out of two elements, Incremental Image Monitor (IIM) 267 and ISOPS parameters 264 that in turn in its present embodiment consists of five fields, each containing an entry which defines a characteristic of the map.

In reference to FIG. 27 entry 1 defines a W-event as a set of $\{k,b,p,m\}$ process parameters. $\{k,b\}$ are the synchronization parameters associated with the signal sent to SUM-Interface 263. Parameter $\{p\}$ is derived from the SUM-Model 243 of the target process 250. It is the current time attribute of the process element which corresponds to the $\{k,b\}$ coordinate pair. The time attribute is obtained from the local time attribute kept by the SUM-Model incremented by one since this W event was not yet executed (retraced) in the SUM-Model in the traditional object code embodiment. In the SUM-Object code embodiment, where the SUM-Model drives the execution, the local time attribute will already be up to date and the retracing is not performed. Parameter $\{m\}$, that is also present within the synchronization signal, is SUM-Model identifier.

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

12 NOV 1997

- 46 -

Entry 2 defines the screen line and column position $\{sl,sc\}$ at the beginning of each system output signal. The $\{sl,sc\}$ entry is $\{1,1\}$ for the very first W-event entry. When SUM-Interface 263 receives a synchronization signal sent by a W-event, SUM-Interface obtains the current screen position that is in effect after the execution of the W-event. The screen position so obtained is stored and later used as $\{sl,sc\}$ entry for a next W-event.

Entry 3 defines the smallest rectangular area of the system output screen 258 $\{l1,c1,l2,c2\}$ which represents the area of the screen changed by the W-event execution. This rectangular area is determined by comparing the previous screen image 255 that is saved at the process time after the previous W-event, and the current system output screen buffer 265.

Entry 5 contains the contents of the sysout screen buffer 254 that corresponds to the area defined in Entry 3. Entry 5 is contained in a variable length field and is stored in the present embodiment as a separate file, the Incremental Image Monitor File 267.

Entry 4 is an integer which defines the displacement of Entry 5 from the beginning of IIM file 267.

• Building System Input Process Synchronized Map

A system input process synchronized map (SIPS) map is built by the SUM-Interface during the course of an execution of the target process to be later used for analysis of a specific input to the system made at some process time. The SIPS map is stored in system input history file 252. The entry made to this file consists of two records. The first record is the set of parameters $\{k,b,p,m\}$ which describe the process time $\{k,b,p\}$ and the model $\{m\}$ corresponding to the current input string. The second record is the actual input string.

The SUM-Interface 263 creates the entry made to the system input history file 252 from the synchronization parameters (k,b,p,m) associated with a signal made to SUM-Interface 263 and the system input buffer .

Alternatively, from taking the system input values for the SIPS map from the system input buffer, these values could be taken from some other buffer where they were temporarily stored. They could be placed there by the statement inserted immediately after the system input statement and constructed from this input statement by simply changing READ to WRITE directed to the buffer. The remainder of the statement, i.e., its parameters and format, remain

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

12 NOV 1997

- 47 -

unchanged. As shown in FIG. 21(B), the READ(*,002)CH is followed by a WRITE(M,002)CH.

FIG. 21(B) shows an example of {ID,m} process synchronization call 2153 that is made before an input from the terminal. The ID of this synchronization signal (21) is translated by SUM-Interface to k,b of 5,3 which corresponds to the process address containing the element E1 (2154 in FIG. 20(A)). If done as shown in the example, the system input is recorded by this means of statement 2155 to the input monitor buffer file 254. There could be several system input strings recorded in the system input monitor buffer by several E elements before the next terminal is met. During the next process synchronization signal performed at the process terminal (which in our example could be performed at = 2157 or C1 2159 in FIG. 20(A)), the system input string entries are taken from the system input monitor buffer 254 and recorded in the system input history file 252. After that time, the system input monitor buffer could be reused.

Alternatively, if the keyboard buffer is used to create an entry in the system input history file (ISOPS file) the presence of the system input monitor buffer is not needed as well as inserting statements 2155 (FIG. 21(B)) within the SUM-Source. Instead, the signals which would read the keyboard buffer and make an entry to the SIPS map would be needed after each system input statement 2154 in FIG. 21(B).

Operational Examples of Software Understanding Machine (SUM) Analyzer

Delayed Sequential execution of the SUM-Model is implemented by one or more CPU(s) which execute the target process and the SUM-Analytical Engine. Synchronization between the execution of the target process and the SUM-Analytical engine takes place through synchronization signals that are placed within the target process by the SUM-Source generator. When the CPU executes such a synchronization signal, program control is transferred to the SUM-Analytical Engine. The target process execution is suspended for the duration of the signal. Target process execution resumes upon return from the signal.

Delayed parallel execution of SUM-Model is implemented through the use of multiple CPUs, where one of the multiple CPUs executes the target process and another of the multiple CPUs executes the SUM-Analytical Engine at the same time. Synchronization between the multiple CPUs takes place through any one of a variety of methods known in the art. The present invention only

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

12 NOV 1997

- 48 -

requires communication of the parameters $\{k,b,m\}$ between CPU(s) working on the target process and CPU(s) working on the SUM-Analytical Engine analyzing the target process.

Both, methods of execution of the SUM-Model are implemented through the use synchronization signals from process terminals $\{idt,m\}$ and optionally, system input $\{ide,m\}$. The synchronization signals $\{idt,m\}$ and optionally $\{ide,m\}$, bring the SUM-Model "up to date" with the target process. The synchronization signal $\{idw,m\}$ is designed to bring ISOPS map "up to date" with the target process and Process Time within SUM-Model. The SIPS map is created either: 1) at the terminal synchronization points if the system input monitor buffer is used; or 2) by the synchronization signals performed after each system input statement.

When SUM-Interface 263 receives a synchronization signal that is implemented in the form $\{id,m\}$, the SUM-Interface translates the $\{id\}$ parameter of the corresponding model $\{m\}$ into $\{k,b\}$. The process address $\{k,b\}$ will point to one of the following: a process terminal $\{idt\}$; a system output signal $\{idw\}$; or an input from the terminal $\{ide\}$.

If the synchronization signal received by SUM-Interface 263 originated from system output element $\{idw,m\}$, SUM-Interface 263 initiates the process previously described which updates ISOPS map.

If SUM-Interface 263 receives an $\{idt\}$ synchronization signal, the $\{idt\}$ parameters are written into system execution history file 253.

In the traditional object code embodiment, if the process synchronization signal received by SUM-interface 263 is $\{idt,m\}$ or in the alternative embodiment $\{ide,m\}$, SUM-Interface 263 then initiates SUM-Analytical Engine 244. Once initiated, SUM-Analytical Engine 244 will retrace, i.e. execute, the SUM-Model 243 from the point of the model at which a previous synchronization signal $\{k,b\}$ was sent or from the entry point of the SUM-Model to the $\{k,b\}$ of the current synchronization signal. As each element of the SUM-Model is traced, the time attribute $\{p\}$ of that element is incremented by one, thereby keeping the SUM-Model 243 time current.

When during the model execution in any of the embodiments, traditional object code embodiment, SUM-Object Code embodiment or embodiment using interpreting of the SUM-Repository, a process element is met that is a system output element (W), SUM-Analytical engine 244 initiates a knowledge induction process. Alternatively the same Knowledge Induction

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

REPLACEMENT SHEET

12 NOV 1997

- 49 -

Process could be initiated directly from the ISOPS map synchronization signal.

The SUM-Repository entry corresponding to the current (W) element is analyzed and each parameter of the W element initiates a cycle of knowledge induction chains. Each of these chains traces back events to their ancestors, up to knowledge propagation terminals.

Tracing backwards is analogous to, but in a different direction from the earlier explained principle of tracing the target process forward. It is done from the present position of the synchronization signal within the target process SUM-Model to the previous position of the synchronization signal within the target process SUM-Model. It is possible within the present SUM-Model construction which knows the rewind entry of each terminal.

The construction of terminated event definition space of a W element that is terminated by knowledge propagation terminals is called knowledge induction.

The terminated event definition space is built with reference to FIG. 25 or alternatively with the reference to FIG. 17. When the W element statement stored in the SUM-Repository is examined all of its variable parameters are identified. In the example shown in FIG. 25, there are four parameters, 2511, 2512, 2513, and 2514, that have defined the value of some $w\{k,b,p\}$ event, which value was saved on the ISOPS map. The branch 2501 represents these four parameters. Starting from the last one, 2514, the branch 2501 is retraced backwards, towards 2511. During the retracing, the SUM-Model is traced against the direction of execution between two events. Originally the first event is the event $\{k,b,p\}$. The second event is the event that defined the value of one of its parameters. In this example, 2514. Further with respect to FIG. 25, the parameter 2514 was produced by the knowledge propagation event that is either a previously approved event or an event that had no ancestors, i.e., system input event or initial data manipulation event. Therefore, this parameter 2514 did not produce a new branch.

Retracing the current branch backwards, parameter 2513 is encountered. The process event which produced the value of this parameter is identified by tracing the SUM-Model against the direction of execution from the event $\{k,b,p\}$. With reference to FIG. 25, this value was produced prior to $\{k,b,p\}$ event by some statement s2 executed at the process time $\{k2,b2,p2\}$. The examination of the SUM-Repository entry of this statement identified the presence of two variables, 2521 and 2522.

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

REPLACEMENT SHEET

12 NOV 1997

- 50 -

Tracing back the model from $\{k2, b2, p2\}$ to the events that defined the values of both parameters 2522 and 2521, the SUM-Analytical Engine finds that in the example of FIG. 25, the value of both of the parameters 2522 and 2521 prior to $\{k2, b2, p2\}$ event, were defined by knowledge propagation terminals previously described. Therefore, new branches were not built for those parameters 2522 and 2521 and therefore the knowledge propagation processes were not activated in those directions.

The branch 2502 therefore was the only branch representing the event definition space of the event $\{k2, b2, p2\}$. The SUM-Analytical Engine is brought back to the branch 2501 to continue the examination backwards against the direction in which the branch was built. The next parameters examined would then be 2512. In our example the value of this parameter 2512 was determined to be produced by some knowledge propagation terminal event (KPT event) and therefore the branch that would represent its event definition space was not built.

SUM-Analytical Engine then retraces the current branch back to the parameter, 2511. In our example its value was also defined by KPT event.

The analysis of parameter 2511 concludes the analysis of the value of W-event $\{k, b, p\}$ event definition space which in the example shown in FIG. 25, is defined by events that define the value of parameters located on the two branches 2501 and 2502. If, during the construction of event definition space we pass an event which was previously uncertain, (in our example it was the event $\{k2, b2, p2\}$) its status is upgraded within the corresponding ACB structure.

Every time that an execution event status is upgraded from uncertain to correct by the knowledge induction process, a corresponding bit in the Analyzed Calculation Base structure 273 is updated from a value of 0 to a value of 1, thus increasing the knowledge of the ACB structure and therefore the knowledge of target process SUM-Model.

A user of the SUM-Analyzer views the system output that is recreated from the system result history by the Incremental Image Extractor/Constructor. Operationally, the user positions a cursor, mouse or other pointing device at the system result. The system result so identified, corresponds to $\{k, b, p, m\}$ parameters of the ISOPS map which defines the event of W element execution. By analyzing the pointing device position within the area defined by entry 3 of the ISOPS map, the corresponding parameter of that W element (system output statement) is defined.

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

REPLACEMENT SHEET

12 NOV 1997

- 51 -

If the so identified system output parameter result is defined by the user as unexpected and therefore incorrect, the SUM-Analytical Engine will start the knowledge deduction process.

Knowledge Deduction Process

Knowledge deduction is performed by a process identical to the knowledge induction process which builds the event definition space for the output events. The output events are traced back to their ancestors. This tracing is terminated in each direction by reaching knowledge deduction terminals which are the same as knowledge induction terminals, i.e., previously approved events or events that have no ancestors.

Unlike knowledge induction, there is no upgrading to the ACB structure knowledge during the knowledge deduction process. Instead, when a previously approved W-event is disapproved, the target process knowledge is dropped to the level, determined by the position of the disapproved W-owner event in the target process time as previously described.

There are two effects of the knowledge deduction process. First, there could be a loss of knowledge which is represented by changing the status of some process addresses from CPA to UPA. With reference to FIG. 16, when W owner of element k,b was defined, its position on the ISOPS map was recorded as one of the attributes of that k,b address. When some other position within the ISOPS map is identified as a failure, SUM-Analyzer will change the status of all addresses previously defined as CPA to UPA if their attributes representing W owner position within ISOPS map point to a

place within the ISOPS map representing a later or equal time as compared with the reference failure. This works because the ISOPS map is built in chronological order in one direction and the process time of the W event chronology is preserved in the ISOPS map.

Second, the position of possible faults or position of required modifications is determined as an area of uncertainty within the definition space of the system output result. This area of uncertainty is defined as those process addresses of this results event definition space that lie outside the correct algorithm space.

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

12 NOV 1997

- 52 -

As can be fully appreciated by those skilled in the art, a target process is fully represented by its SUM-Graph and SUM-Model. In that regard, once the SUM-Graph, SUM-Model and SUM-Repository of a target process are generated by the SUM-Analyzer, there is no longer a need for the SUM-Source Generator, SUM-Source, the Compiler/Linker. The Software Understanding Machine can execute and analyze the target process without these above-mentioned components and therefore without the necessity for synchronization signals.

There are two embodiments which could be done. One embodiment uses an interpreter 268 that interprets statements contained within SUM-Repository 246 (shown in FIG. 18(C)). The order of the interpretation is controlled by SUM-Driver which is tracing, i.e. executing, the SUM-Model 243 and therefore is being controlled by the SUM-Model.

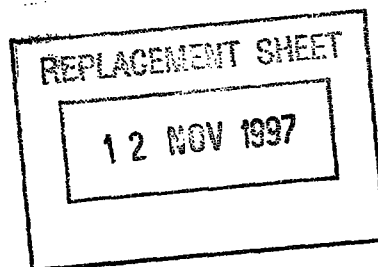
Another embodiment uses SUM-Object code which comprises the translated to machine code elements of the SUM-Repository 246. The resulting code, therefore does not contain any control elements such as logic conditions, loop conditions, conditional or unconditional GOTO, RETURN, STOP, or loop terminals. The conditional statements are represented in SUM-Object code only by the expressions to be evaluated. In this embodiment as well as the embodiment which uses an interpreter, the control of the target process is performed by the SUM-Driver tracing (executing) the SUM-Model. In the SUM-Object code embodiment, the SUM-Model becomes a part of the object code as it is traditionally understood since the SUM-Model performs the control functions.

Since the SUM-Model is controlling the process execution the SUM-Model does not need to be informed of the active event in the target process. Therefore, the process synchronization signals are not needed. The only function of the SUM-Interface in these two embodiments is to create the ISOPS and SIPS maps.

With reference to the SUM-Object code embodiment shown in FIG. 18(A) and FIG. 18(B), two components of the SUM-Object code are shown. Shown in FIG. 18(B) are a control component 1800 and an executable component 1801. The control component 1800 has a pointer 1802 to a segment 1804 within the executable component. Each segment within the executable component represents a statement of the SUM-Repository that has been translated into object code.

This Sheet Has Been Replaced

THIS SHEET HAS BEEN
REPLACED



- 53 -

The Executable Component Address (ECA) attribute 1802 is added to each executable element PA of the SUM-Model. Each segment i.e., 1804, is followed by the Statement Separator Byte (SSB) 1803. SUM-Driver 269 reads the value of ECA from SUM-Model 243, reads the corresponding segment of object code 1803 and passes it to the cpu 241 which is located within 1806, that runs the executable component of SUM-Object Code.

As one of different methods for monitoring system input, the following is implemented: when module 1807 translates a statement representing input from the system terminal into Executable Component of SUM-Object Code, it adds a next statement, that reverses the action of an input from a system terminal to output to the input monitor buffer file 254. For example and with reference to FIG. 21(B), READ (* ,002)CH 2154 is followed by WRITE(M,002)CH 2155.

SUM-Driver then has an option to activate or deactivate this second statement in order to control monitoring of the system input while selecting the next code fragment to be passed to the module 1806.

Since the SUM-Object code architecture consists of two components, control and control-less, this architecture allows for easier modification of the target process. When the change needed is a control change, the control component is changed. When the change needed is a non-control component, a new entry is made in the executable component file and the ECA reference in the control component is updated.

The resolution of the link between the control and non-control components through the ECA is done at a run time. Therefore compiling, recompiling or interpreting is needed only for newly created or modified statements of the non-control component of SUM-Object Code. It is even possible for the interpreter to be invoked only for the newly created or modified statements. This includes those situations where the newly created or modified statement is just a single statement.

The control component of SUM-Object Code can be rearranged by insertion, deletion or modification to the SUM-Model element corresponding to the newly created SUM-Repository element. In some situations, when only the control structure needs to be rearranged, a modification to SUM-Object Code can be made without any changes being made to the non-control component. This is done by rearranging the order of the elements within SUM-Model.



12 NOV 1987

- 54 -

Modification to the SUM-Object code can be done in some situations by deactivating the control component and, therefore, bypassing the corresponding element of the non-control component. Further, if the resolution of the links between the Control and non-control SUM-Object code components is performed at run time, linking or re-linking of a group of two or more target processes implemented by SUM-Object Code is done by linking only the control components of the SUM-Object Code.

Result Oriented Programming

The SUM-Analyzer can address any event of any statement execution through the reference to the system results by using a system pointing device or cursor. Therefore, while building the event definition spaces of the system results, there exists a way to automatically modify the target process code.

The event definition space of system result events or of the event definition space of an intermediate value calculation event could be reused to create new system results or conditions for new system results. The system results could be rearranged in the process time that corresponds to the system results.

This serves as the basis for the method of Result Oriented Programming (ROP) that is, a very high level language for forward development of a software process. In fact the level of this language grows with the forward development of the target process, since there are more and more events to analyze and more event definition spaces to reuse.

Program Stethoscope

Audio analysis, combined with traditional video analysis allows for better dynamic understanding of programs or system of programs. Execution of any target process can be seen as it is traditionally done by tracing the graph dynamically, but it also can be heard, if one generates sound pitches which correspond to statement positions within the SUM-Graph. Generated this way, "music" would be defined by the process algorithm and is a function of the input data combination.

During the cognitive process of program understanding we are faced with a task of "playing computer". This is by definition a dynamic process. On the other hand a graph (picture) is by definition a static object. The difference between a graphical and sound representation of dynamic process may

This Sheet Has Been Replaced

THIS SHEET HAS BEEN
REPLACED

REPLACEMENT SHEET

12 NOV 1997

- 55 -

be understood by comparing two ways of recreating music: 1) by reading the music score (graphic representation); and 2) by actually playing the music, or hearing it when looking at the score.

Audio effect during a program analysis can be achieved as follows. During parallel or post-life program emulation by applying next recorded terminal address $\{k,b\}$ to the SUM-Model of the program we can trace the program execution on the SUM-Graph of the program algorithm.

The audio effect is added by calling the hardware internal function that allows the generation of an audio frequency and send it to the speaker. A head phone will be a more appropriate addition to the computer when analyzing a system by the described method, and will not affect other programmers.

The frequency can be calculated as a function of two parameters, (k,b) , where a single increment in $\{k\}$ increments the frequency by one discrete step, and a single increment of $\{b\}$ increments a frequency by a larger step, for example by an octave. A branch back will accordingly decrement the frequency.

The duration of beep, corresponding to the tracing of one individual element of the SUM-GRAPH can be set as such that it would be recognized by the human ear. The effect of such audio execution emulation is such, that each process of program execution has its own "music", depending on that process algorithm structure and input data combination. In the case of emulating a loop having of large number of iterations, a code signal can be set to indicate such a loop in order to eliminate a necessity of tracing all the iterations. This code can be for example represented by longer duration signal, followed by different duration signals, each representing let say 10 iterations of the loop. In addition, a graph depicting the process may be made and as each element is executed, the corresponding portion of the graph is highlighted and its characteristic tone is played.

When a particular output result is not being observed for a long time, this analysis can be activated to hear: 1) whether the system is dead; 2) whether the program in loop; 3) or is it waiting for an input. When the last executed element was an input statement a special audio tone can be generated

Most stressful places of the program's algorithm can be heard as a most common frequency and therefore easily identified and the difference in program behavior with different input data combinations can be heard.

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

12 NOV 1997

- 56 -

A special tone can indicate a call to a child function and a special tone can indicate a return from the call. The audio analysis can be limited to only those signals originally and expanded optionally to indicate process execution within one specific function, if functionality within the function is to be studied.

The Audio analysis can be recorded during the process emulation (alive or post-life) and then replayed without the actual mode emulation in both directions forward and backward.

It can be replayed synthesized with the replay of the system result and thus give a filling of the intensity of the processing necessary for those results achievement;

It can be recorded and replayed for different input data combinations for the same process in order to analyze the difference in intensity of calculations for those different input situations. Given that the potential for the program fault is a function of the number of the elementary processes involved, this analysis will aid in selection simpler (input-process-output) combination during system evolution to a new specifications.

Although various embodiments which incorporate the teachings of the present invention have been shown and described in detail herein, those skilled in the art can readily devise many other varied embodiments that still incorporate these teachings.

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

12 NOV 1997

- 57 -

I claim:

1. An apparatus for automatic analysis of a target process, wherein the apparatus determines the location of faults contained within the target process or location of modifications within the target process required due to a changed expectation of the result, wherein the apparatus generates a secondary process of accumulating knowledge of the target process functionality through knowledge induction, comprising:

an analyzed calculation word attribute for each executable element of the target process;

means for executing the target process and said secondary process,

means for said secondary process to build and propagate an elementary knowledge of the target process functionality through knowledge induction that propagates said elementary knowledge through a place and time within the target process,

wherein said elementary knowledge indicates a knowledge about or an uncertainty about the correctness of an effect of the target process element execution event and,

wherein said elementary knowledge is recorded within said analyzed calculation word and,

wherein the location of fault or location of modification within the target process required due to the changed expectation of the result is obtained through knowledge deduction.

2. The apparatus of the claim 1, wherein the target process is implemented by execution of code in traditional object code form, while synchronization signals are sent from the target process to said secondary process.

3. The apparatus of the claim 1, wherein the target process is implemented by a driver tracing a model of the target process and presenting corresponding executable elements of the target process represented in their source code form to a language interpreter, while said secondary process is activated by said driver.

4. The apparatus of the claim 1, wherein the target process is implemented by a driver tracing a model of the target process and presenting corresponding executable components of SUM object code to the computer operating system, while said secondary process is activated by said driver.

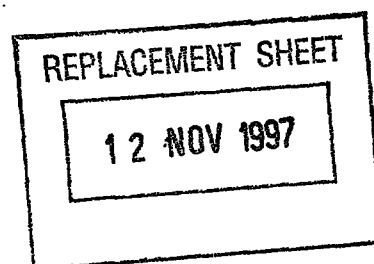


This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

5. The apparatus of the claim 1, further comprising means for constructing the definitions of modifications within the target process, required due to the changed expectation of the result, wherein said definitions are obtained by reusing event definition spaces of the prior results of the target process.



This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

METHOD AND APPARATUS FOR THE AUTOMATIC ANALYSIS OF COMPUTER SOFTWARE

Abstract of the Disclosure

A method and apparatus for the analysis of a computer process is disclosed. The method consists of the steps of analyzing a target process and constructing a model and repository which are representative of the control and non-control components contained within the original target process respectively. The resultant model and repository are executed, and their execution is representative of the original target process. Structures created during the execution of the model and repository assist a user to identify failures which occur during the execution, and then locate the fault(s) responsible for the failures as well as places within the target source where modifications due to new requirements are needed. An alternative form of object code is proposed which consists of two components, control and non-control components. Previously described model representing the control component constantly increases the intelligence of the resulting object code in respect to the target process being executed through the processes of propagation of elementary knowledge about the correctness or uncertainty of the target process elements execution.

REPLACEMENT SHEET

12 NOV 1997

THIS SHEET HAS BEEN
REPLACED

This Sheet Has Been Replaced



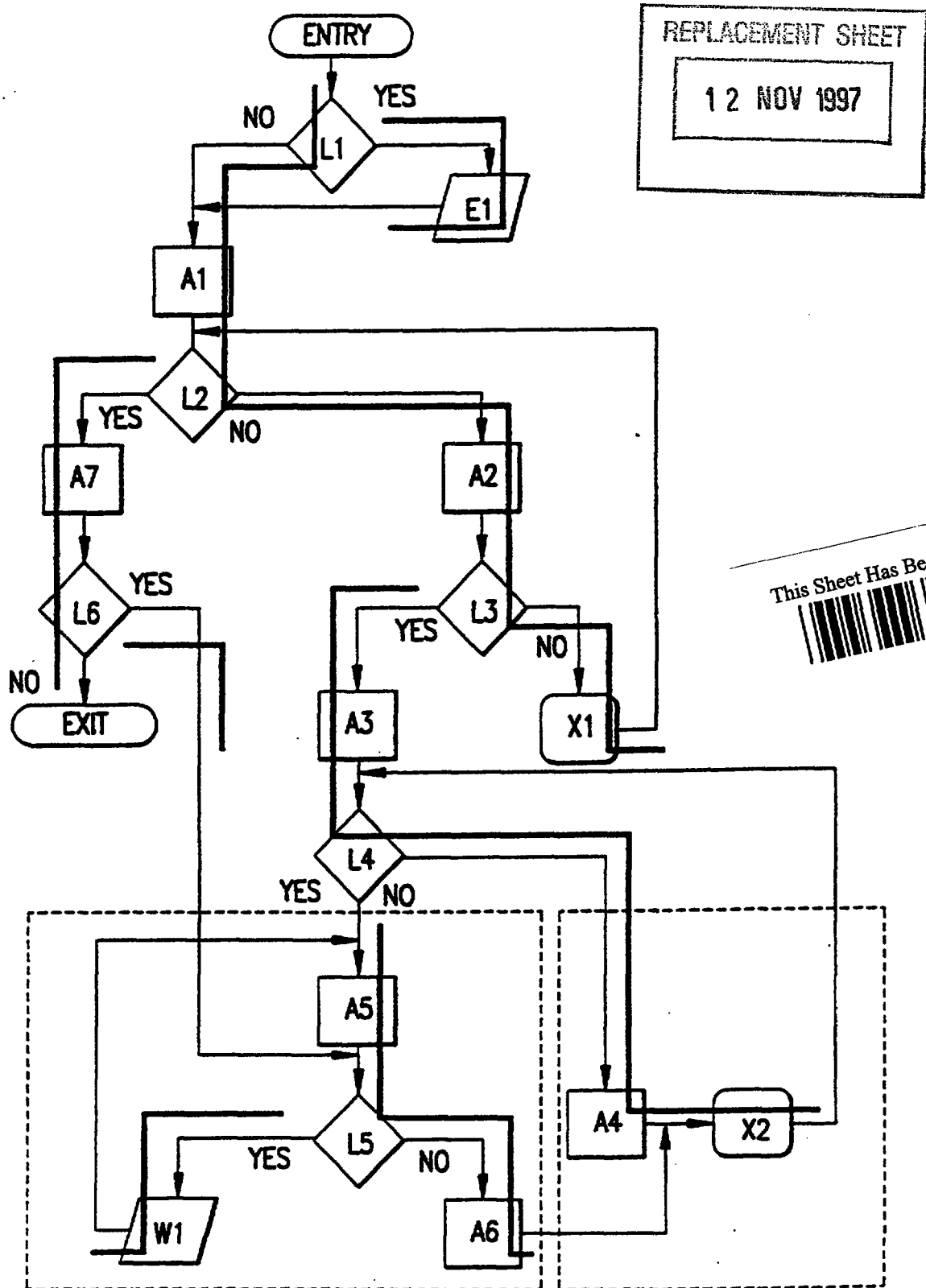


FIG. 1

THIS SHEET HAS BEEN
REPLACED

This Sheet Has Been Replaced

THIS SHEET HAS BEEN
REPLACED

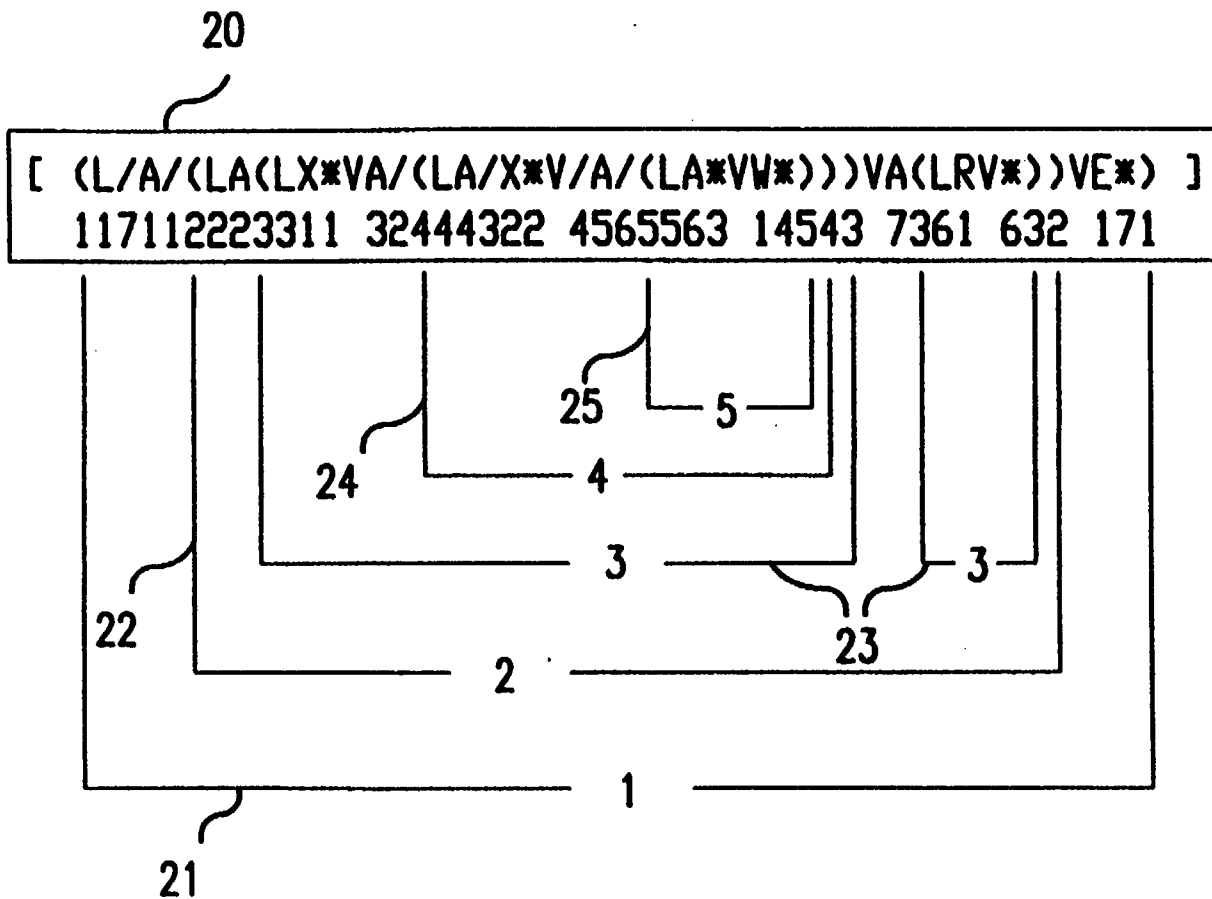
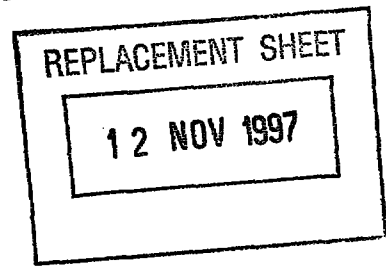


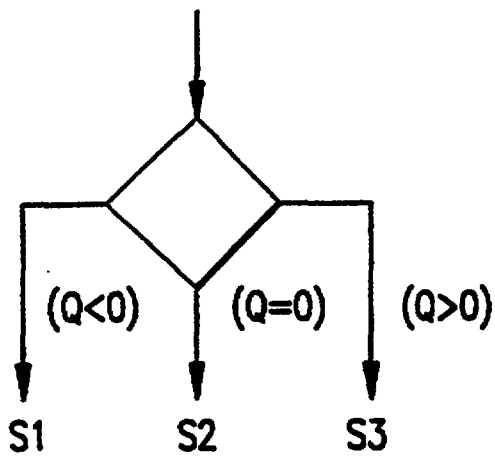
FIG.2

REPLACEMENT SHEET
12 NOV 1997



EXAMPLE:
IF (Q) S1, S2, S3

THAT IS:



CAN BE REDUCED TO:
IF (Q.LT.0) GO TO S1
IF (Q.EQ.0) GO TO S2
GO TO S3

IS REDUCED TO:

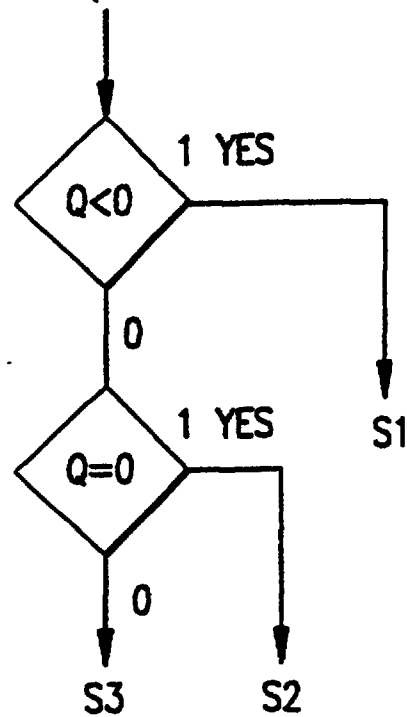
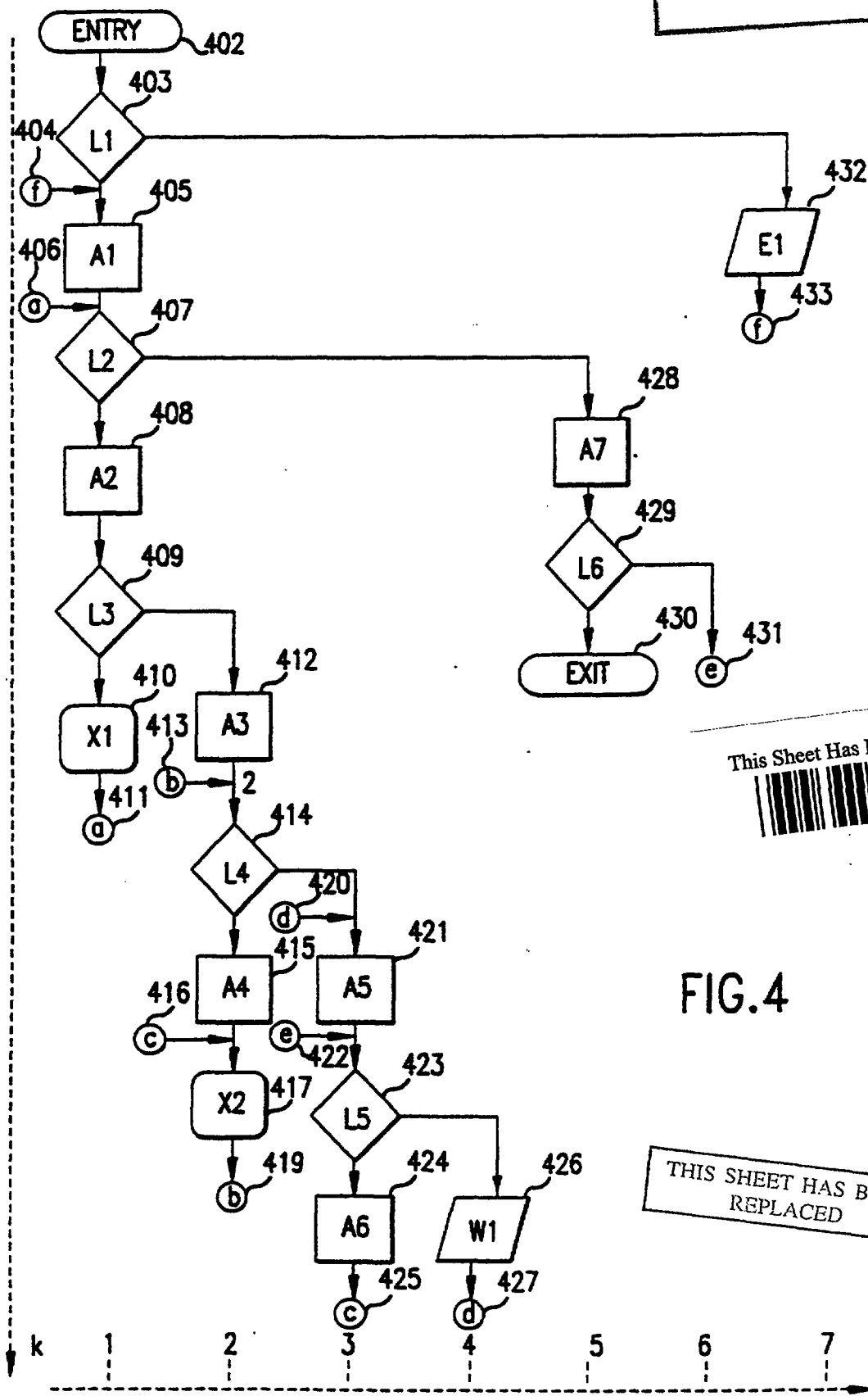
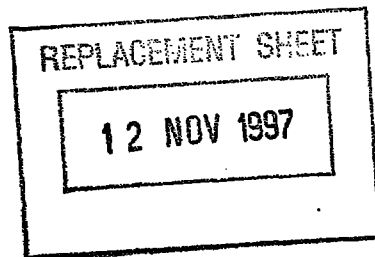


FIG.3



THIS SHEET HAS BEEN
REPLACED



This Sheet Has Been Replaced



FIG.4

THIS SHEET HAS BEEN REPLACED

Barcode

12 NOV 1997

- 5 -

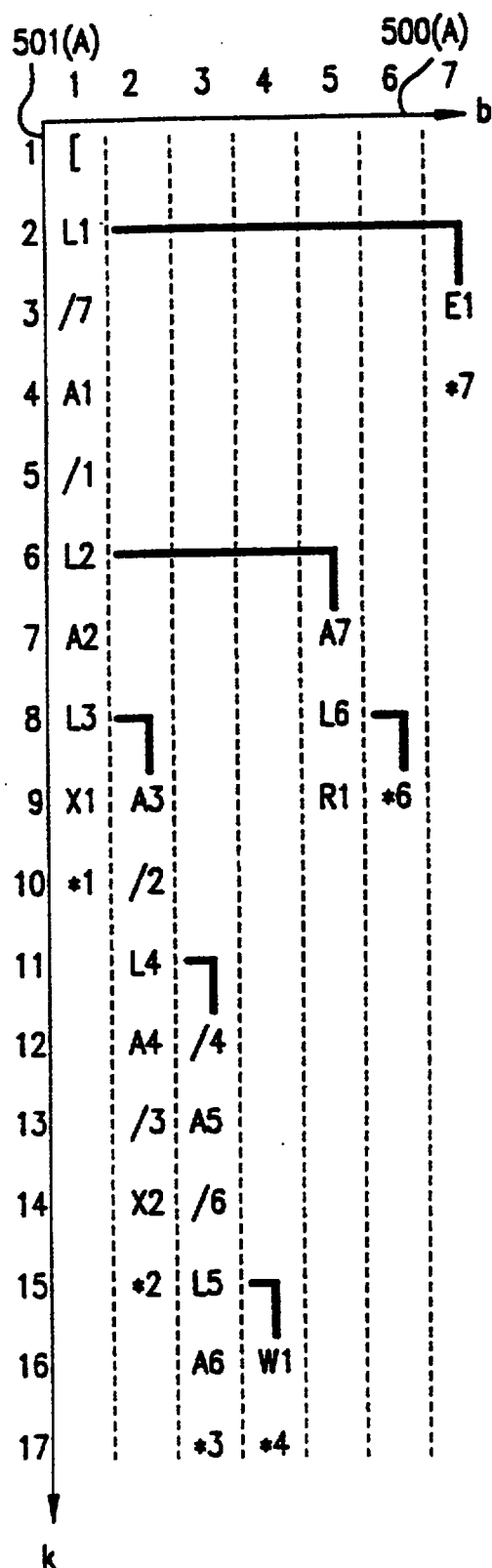


FIG. 5A

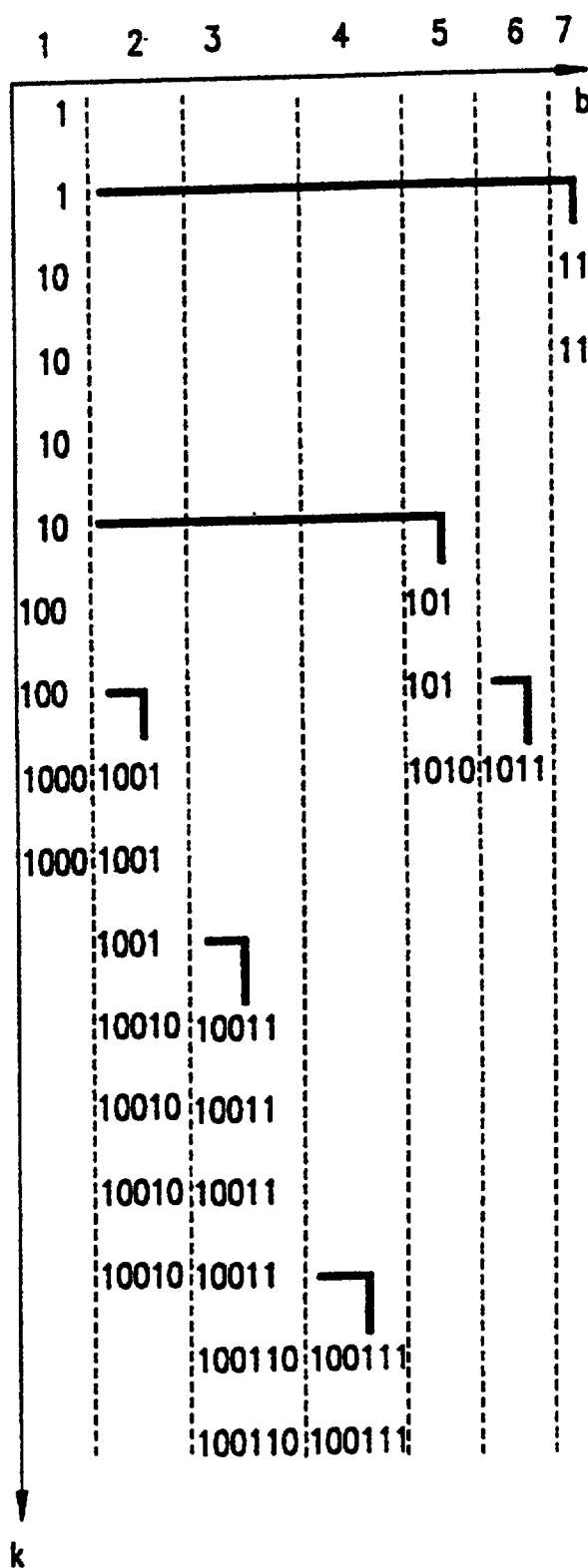
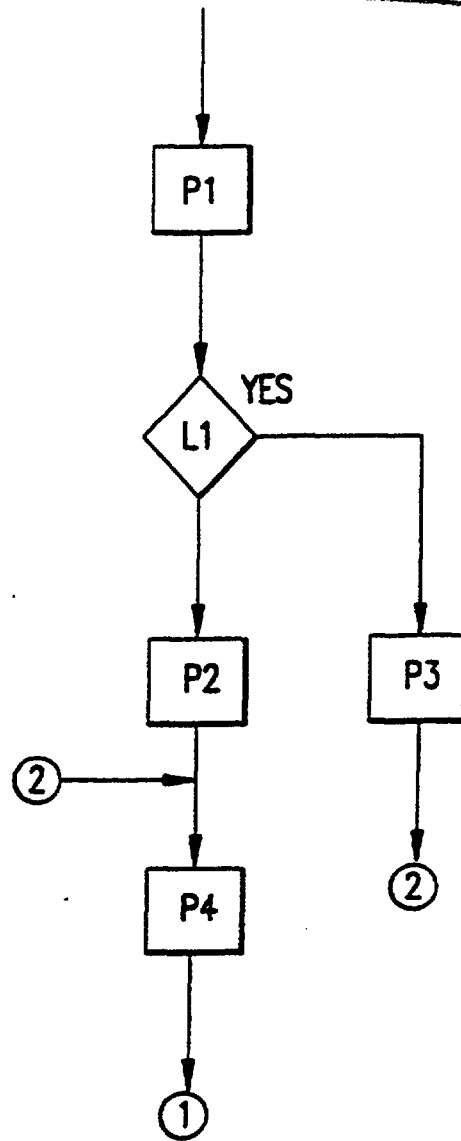
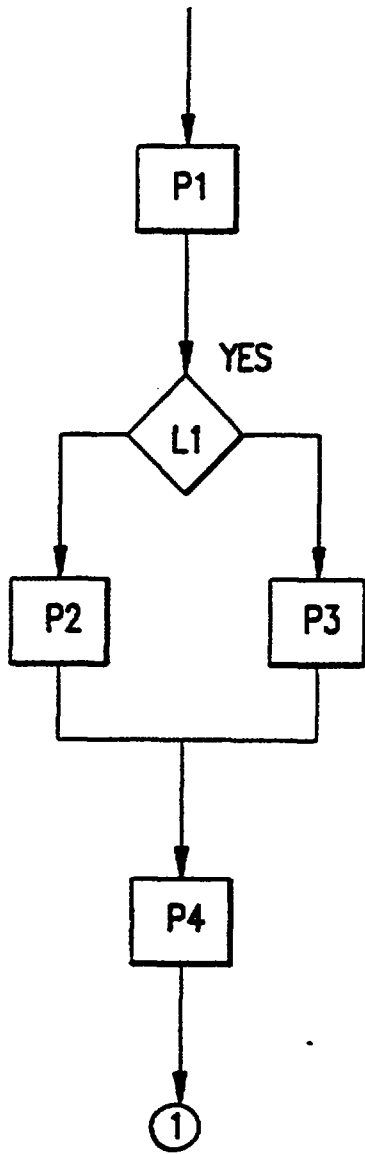
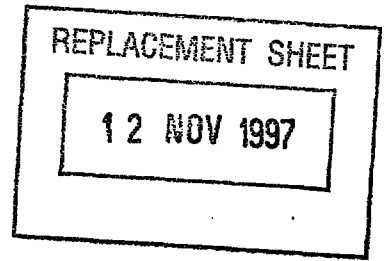


FIG. 5B

THIS SHEET HAS BEEN
REPLACED



P1 (L1 P2 !2 P4 *1 V P3 *2)
i i

P1
IF (L1) THEN
P3
ELSE
P2
END IF
P4



FIG.6

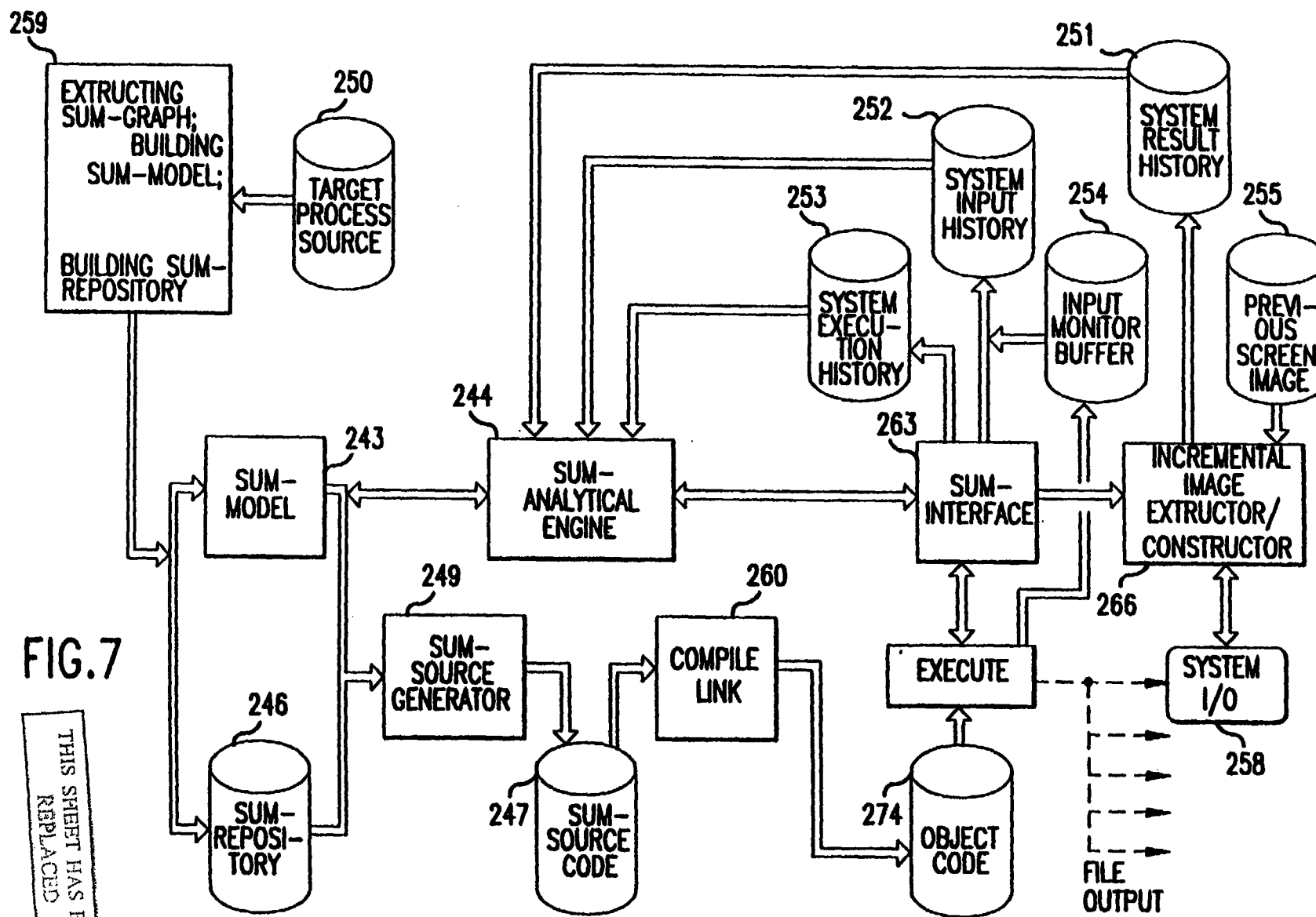


FIG. 7

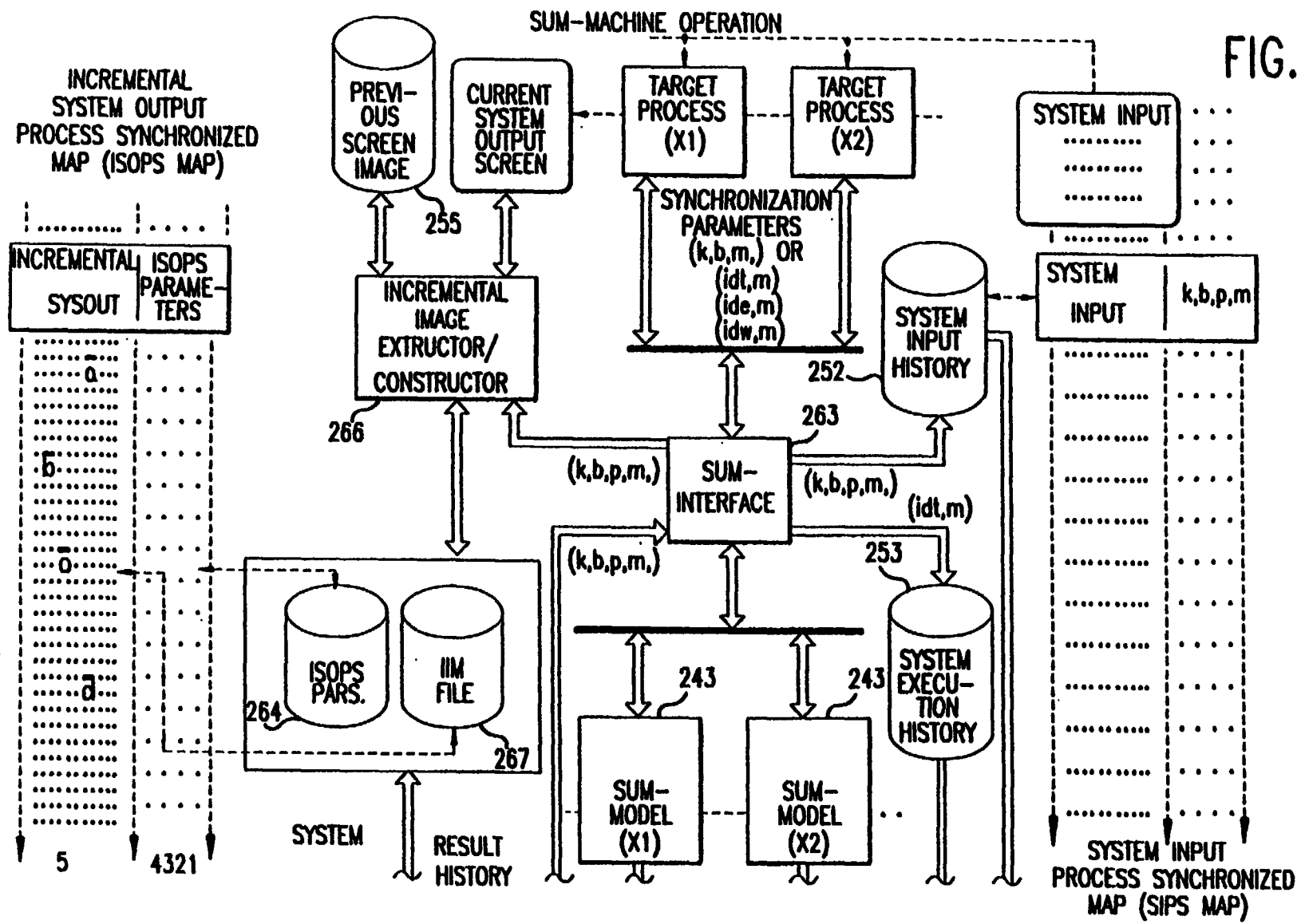
THIS SHEET HAS BEEN REPLACED

REPLACEMENT SHEET
12 NOV 1997



12 NOV 1997

FIG.8A



THIS SHEET HAS BEEN REPLACED

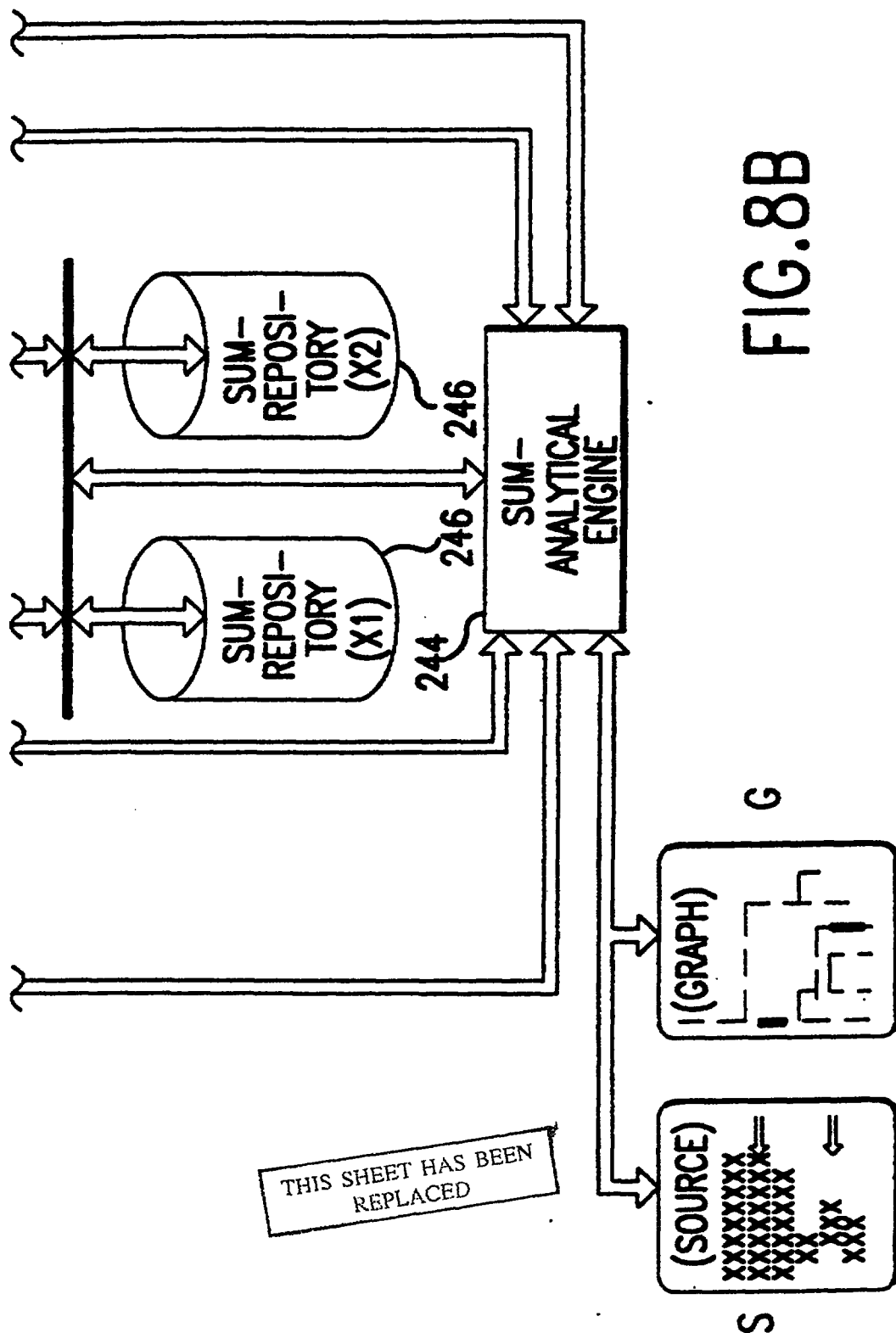
This Sheet Has Been Replaced



- 9 -

REPLACEMENT SHEET

12 NOV 1997



G

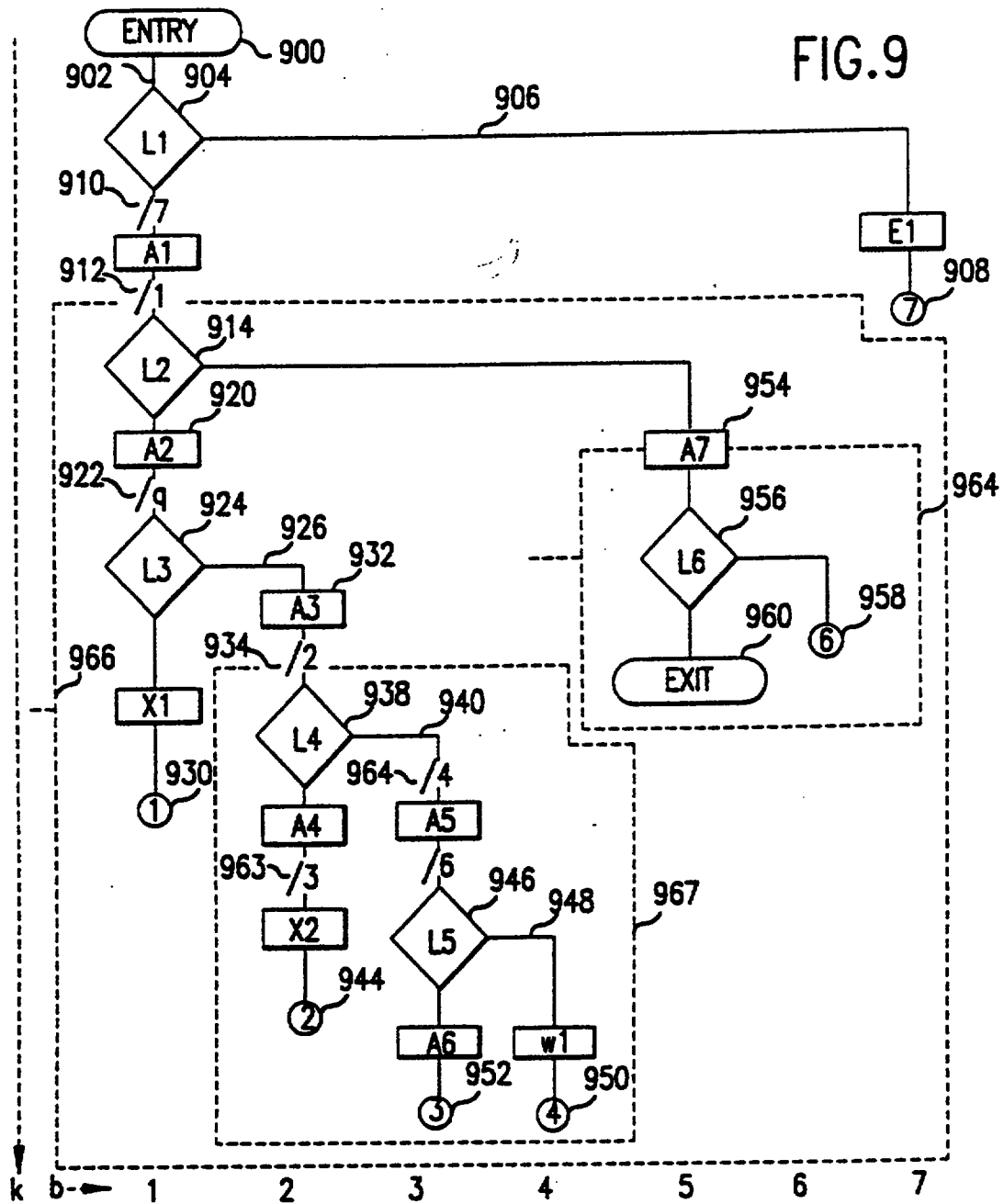
S

THIS SHEET HAS BEEN
REPLACED



12 NOV 1997

FIG.9



EXIT	ORIG	0	0	0	0	1	0	0
ACC	END	1	0	0	0	1	0	1
IND.	ORIG	rk1	rk2	rk3	rk4	0	rk6	rk7
	END	rk1	rk2	rk2	rk2	0	rk2	rk7
PVP	ORIG	10	1001	10010	10011	0	10011	10
	END	10	1001	1001	1001	0	1001	10
RBA	ORIG	10	1001	10010	10011	0	10011	10
TBA	END	1000	10010	100110	100111	1010	1011	11

THIS SHEET HAS BEEN REPLACED

This Sheet Has Been Replaced



- 11 -

REPLACEMENT SHEET

12 NOV 1997

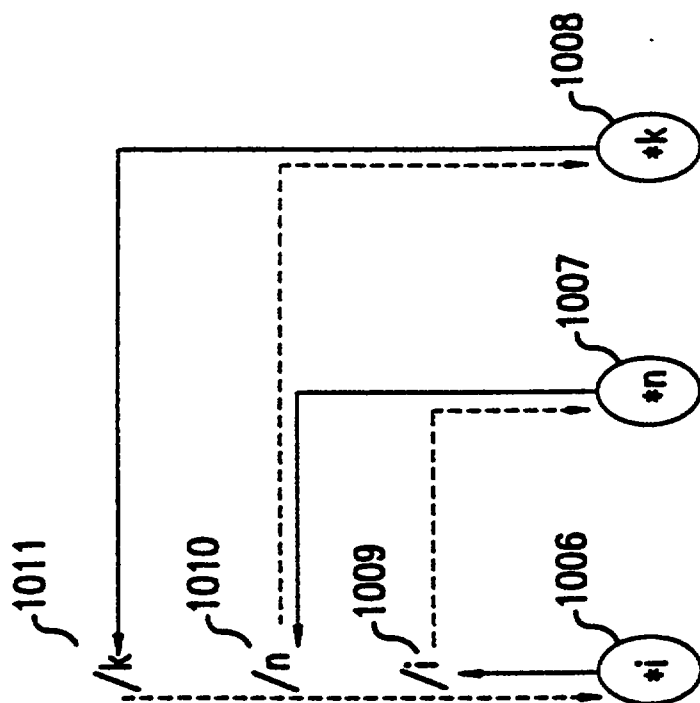


FIG. 10B

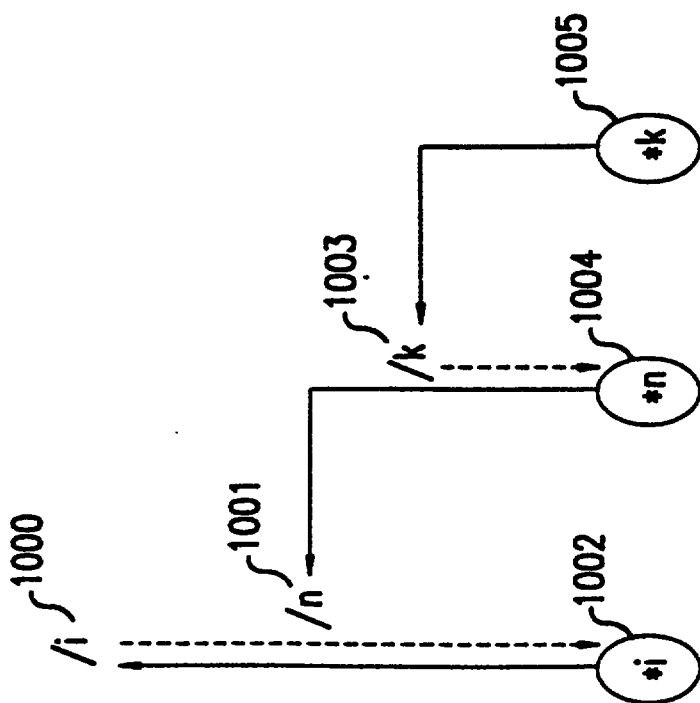


FIG. 10A

THIS SHEET HAS BEEN
REPLACED

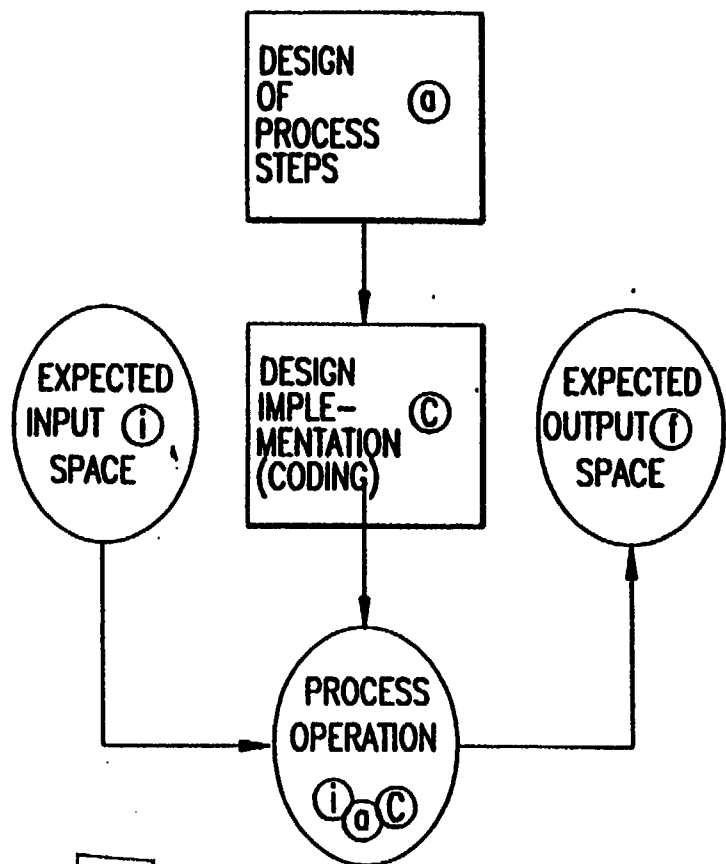


FIG.11

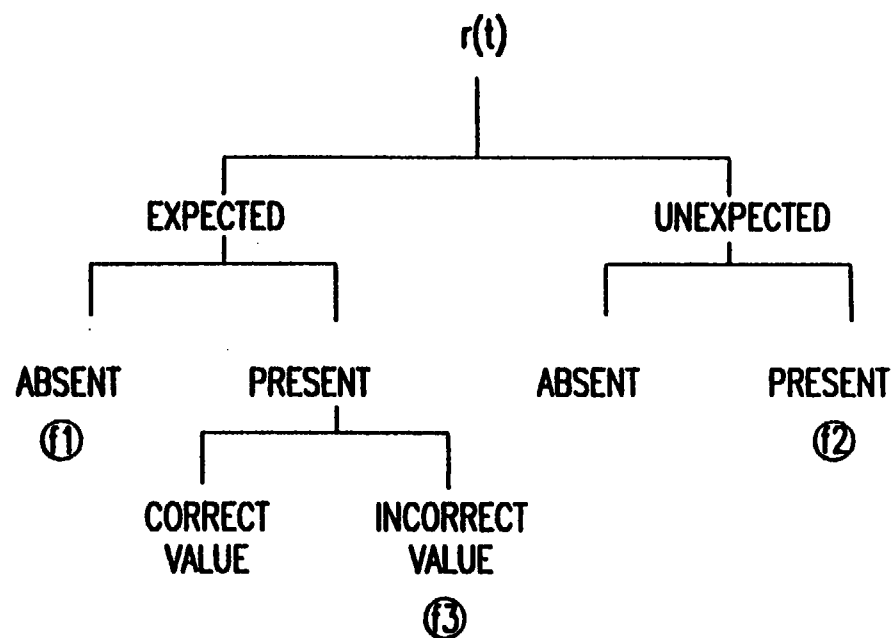


FIG.12



This Sheet Has Been Replaced

- 12 -

THIS SHEET HAS BEEN
REPLACED

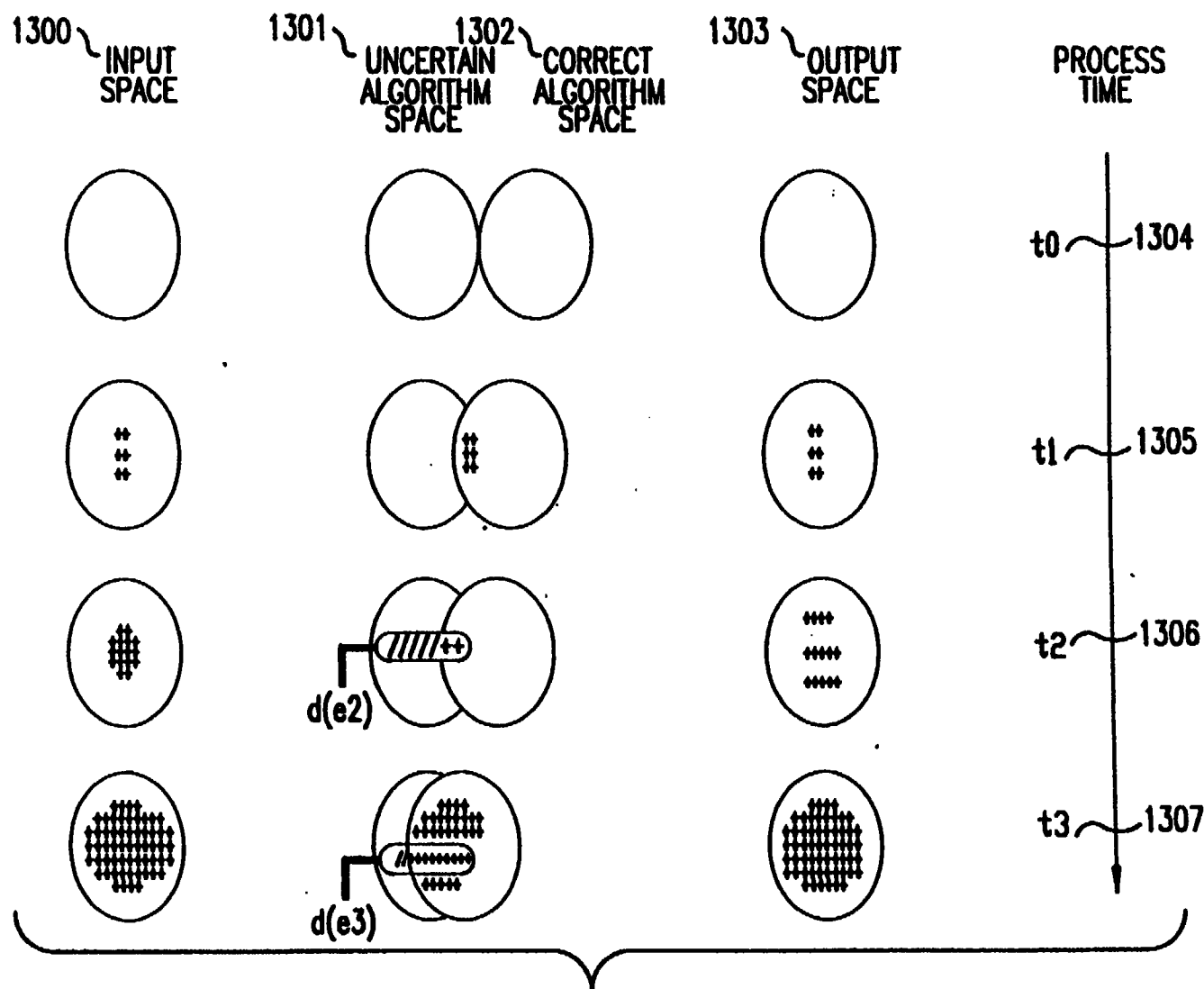
12 NOV 1997

REPLACEMENT SHEET

This Sheet Has Been Replaced

- 13 -

REPLACEMENT SHEET
12 NOV 1997



THIS SHEET HAS BEEN
REPLACED

FIG.13

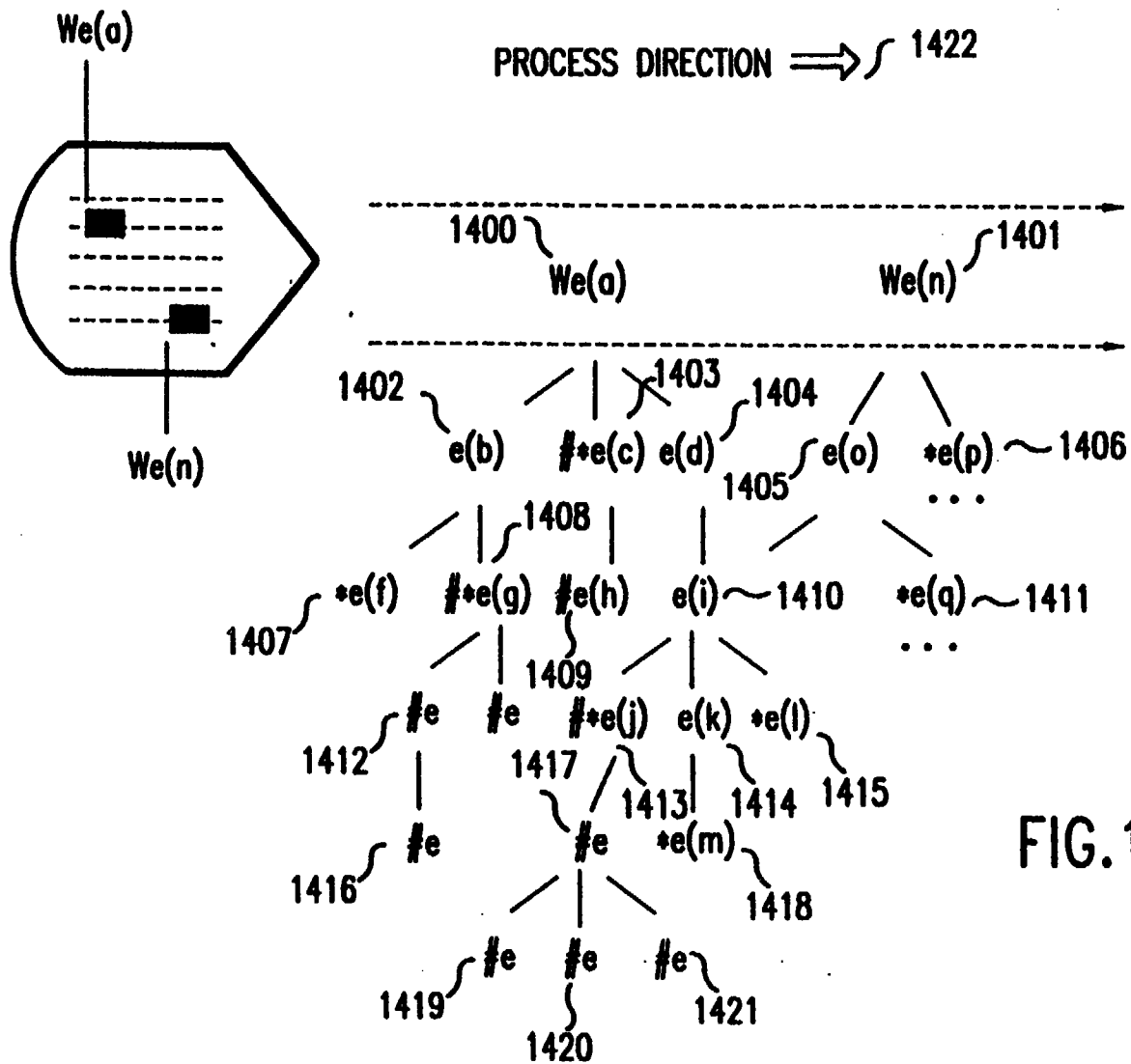


FIG.14

This Sheet Has Been Replaced

- 14 -

THIS SHEET HAS BEEN
REPLACED

REPLACEMENT SHEET
12 NOV 1997

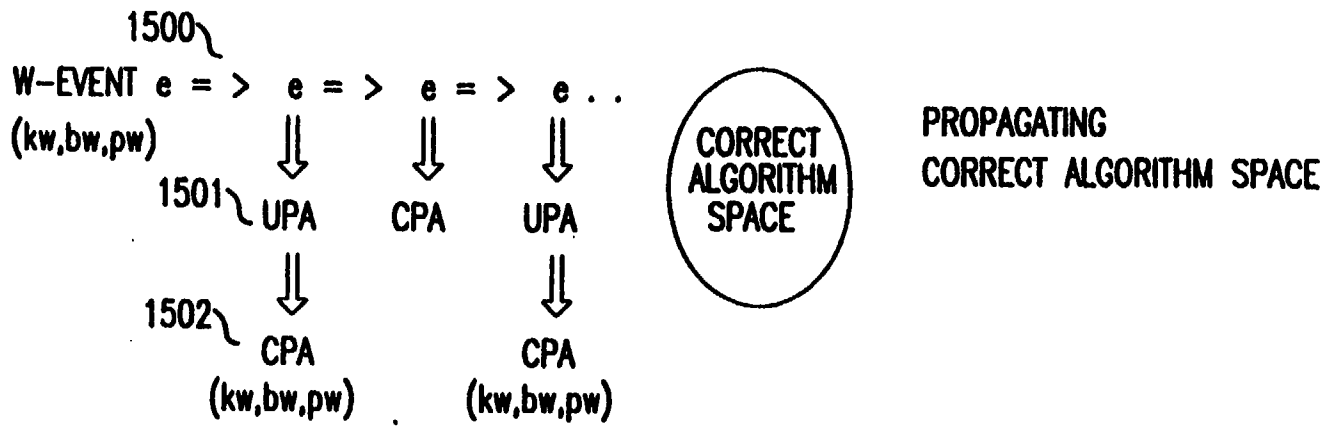


FIG.15

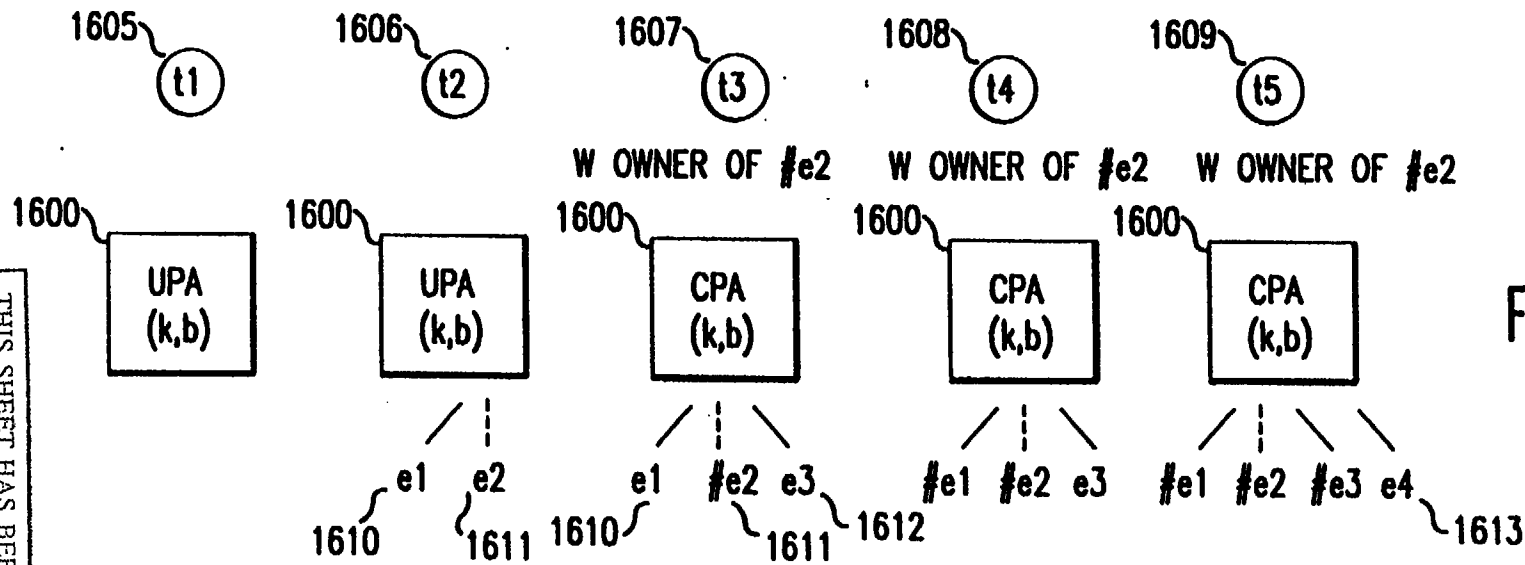


FIG.16

THIS SHEET HAS BEEN
REPLACED

REPLACEMENT SHEET
12 NOV 1997

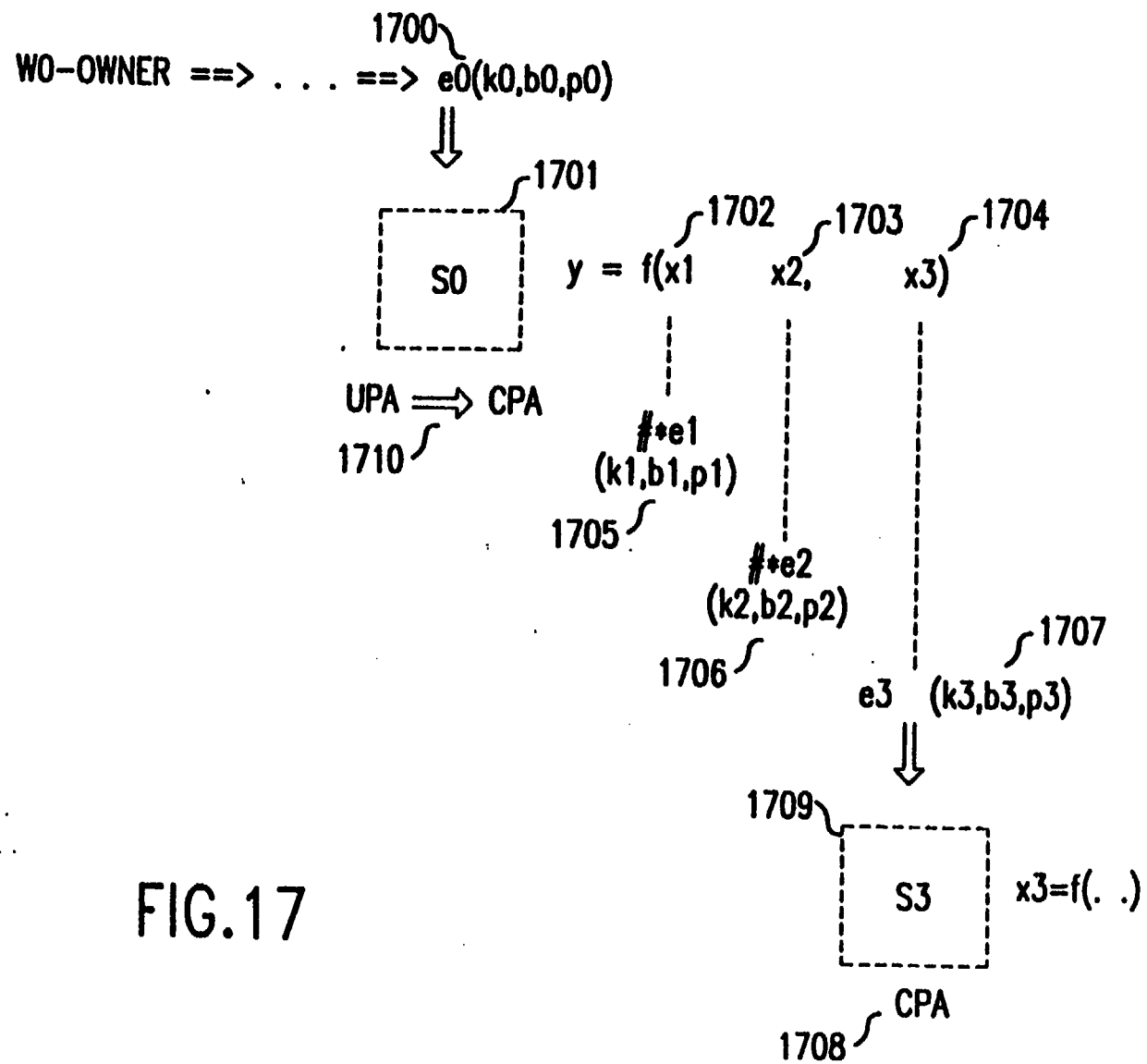


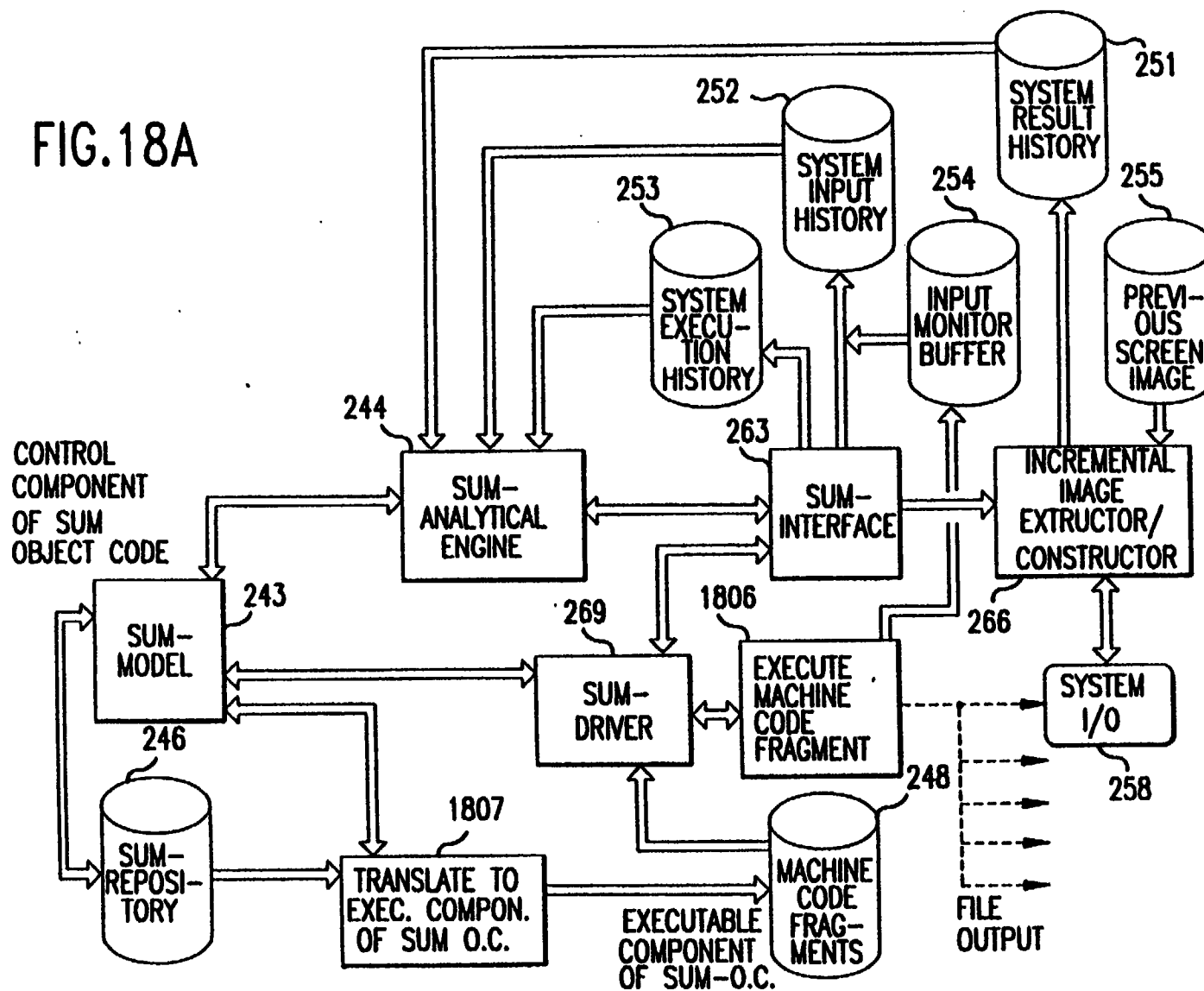
FIG.17

This Sheet Has Been Replaced

REPLACEMENT SHEET
12 NOV 1997

THIS SHEET HAS BEEN
REPLACED

FIG.18A



This Sheet Has Been Replaced

THIS SHEET HAS BEEN REPLACED

REPLACEMENT SHEET
12 NOV 1997

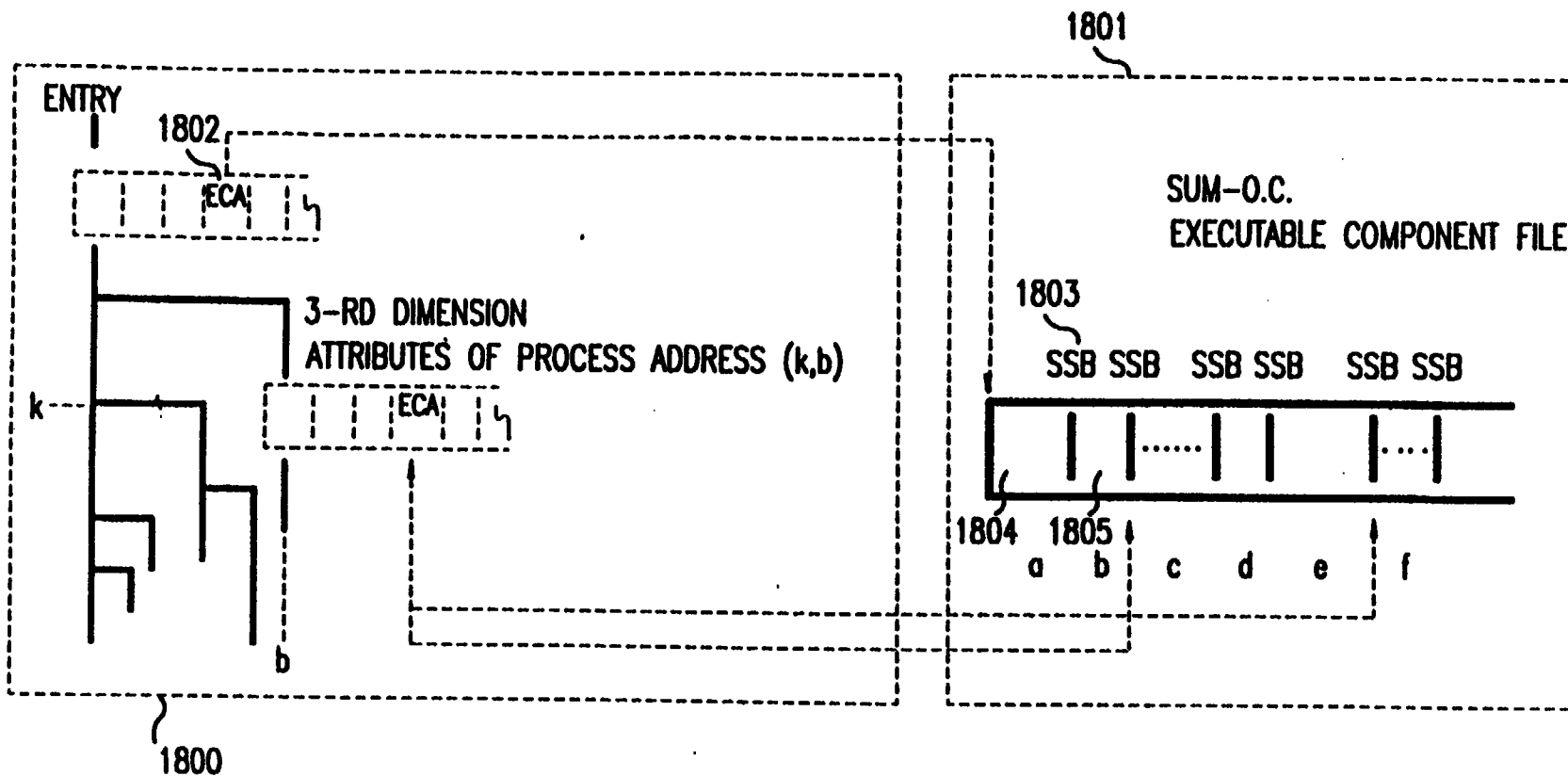


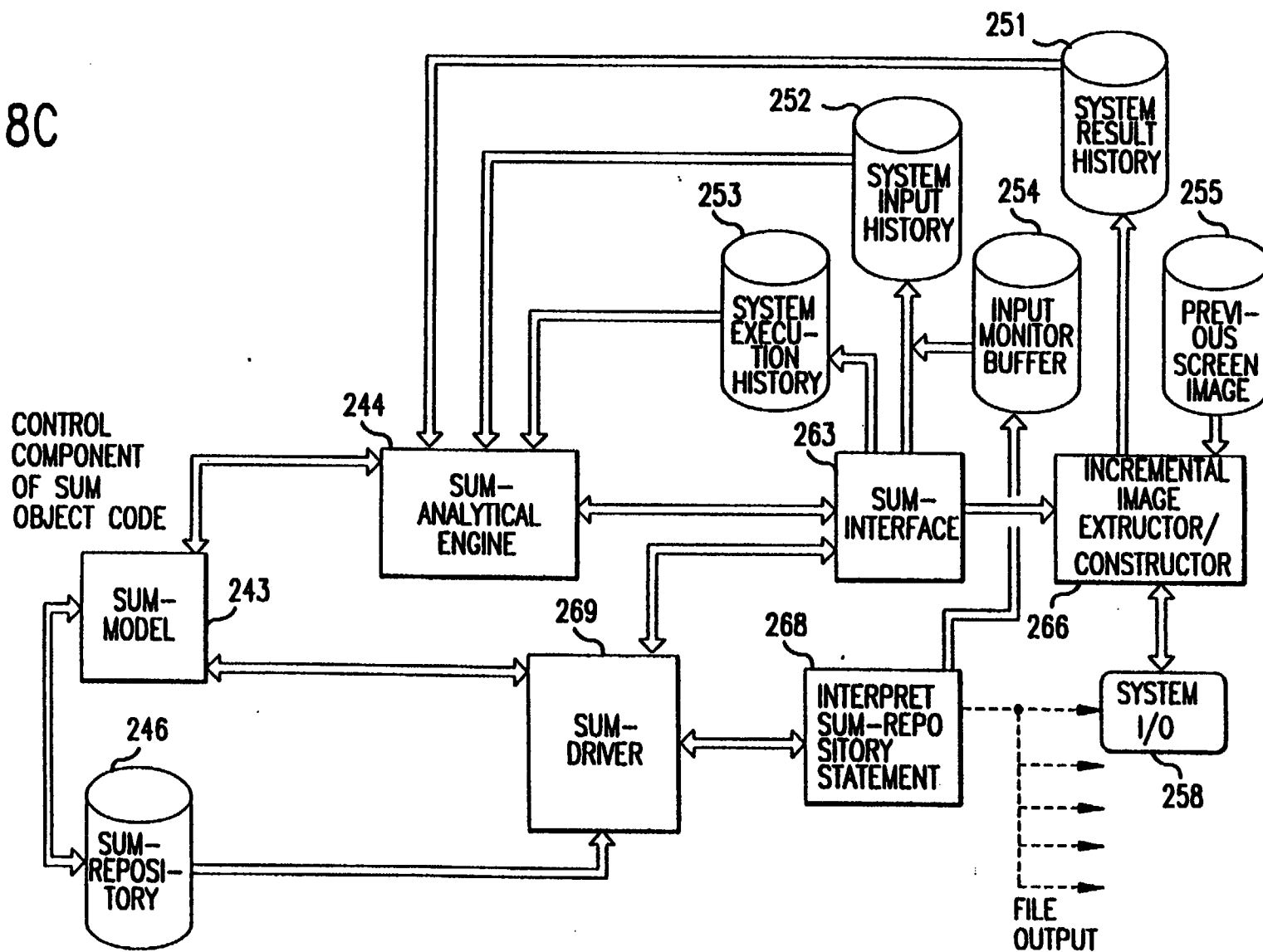
FIG.18B

This Sheet Has Been Replaced

THIS SHEET HAS BEEN
REPLACED

REPLACEMENT SHEET
12 NOV 1997

FIG.18C



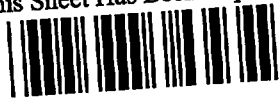
This Sheet Has Been Replaced

THIS SHEET HAS BEEN REPLACED

REPLACEMENT SHEET

12 NOV 1997

This Sheet Has Been Replaced



- 20 -

REPLACEMENT SHEET

12 NOV 1997

		CONTROL AREA		INFORMATION AREA
		NPW	UDW	
1900 ~ {	1901 { A1		2000 ₁₀	00011010010001101111
	A2		127 ₁₀	00011010010001101111
	D1		0 ₁₀	01101110010111101111
1905 ~ {	E1-1	*E1-2	804 ₁₀	00111001100010010001
1906 ~ {	E1-2		0 ₁₀	01111100011110010001
1907 ~ {	X1-1	*X1-2	97 ₁₀	01001010110111001111
1908 ~ {	X1-2	*X1-3	104 ₁₀	01111111111110010001
1909 ~ {	X1-3		0 ₁₀	01000000110111001111

THIS SHEET HAS BEEN REPLACED

FIG.19

12 NOV 1997

- 21 -

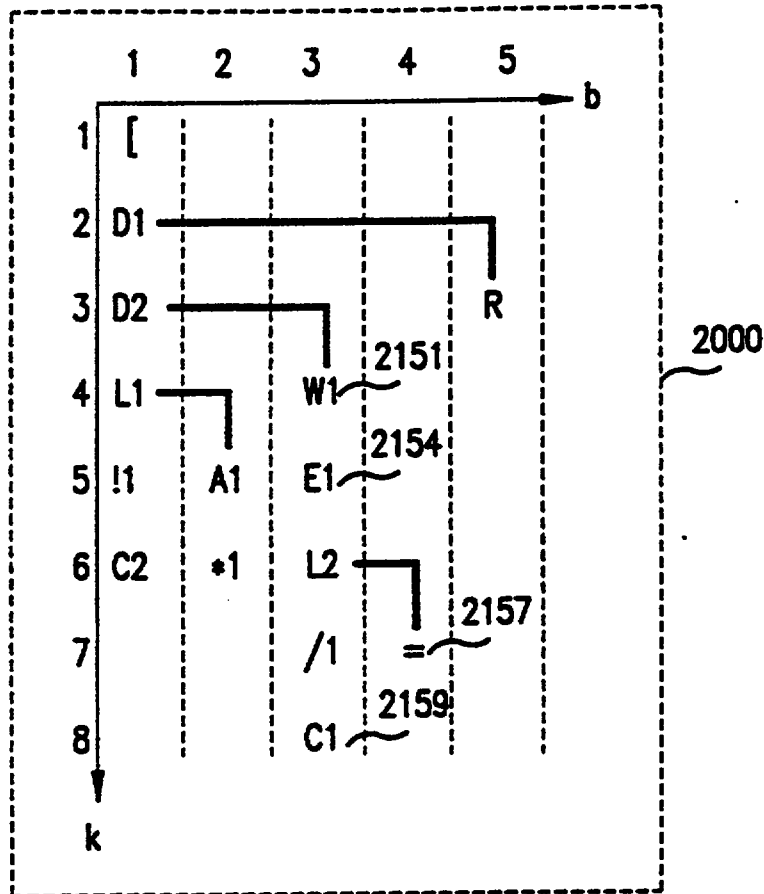


FIG. 20A

```

      INTEGER IN(3)
      CHARACTER IA(3), IALF(10), CH
      DATA IALF/'0','1','2','3','4','5','6','7','8','9'/
001  FORMAT(3X,'IN(I)=' ,I3,' DOES NOT MATCH INTEGER FROM 0 TO 9',
      C' TYPE *Y* TO CONTINUE')
002  FORMAT(A)
D1:   I=1,3
D2:   J=1,10
L1:   IF(IN(I).EQ.J-1)
A1:   IA(I)=IALF(J)
W1:   WRITE(*,001)IN(I)
E1:   READ(*,002)CH
L2:   IF(CH.NE.'Y'.AND.CH.NE.'y')

```

THIS SHEET HAS BEEN
REPLACED

FIG. 20B

This Sheet Has Been Replaced



This Sheet Has Been Replaced

THIS SHEET HAS BEEN REPLACED

2100

```

SUBROUTINE I2A(IN,IA)
INTEGER IN(3)
CHARACTER IA(3),IALF(10)
DATA IALF/'0','1','2','3','4','5','6','7','8','9'/
DO 10 I=1,3
  DO 55 J=1,10
    IF(IN(I).EQ.J-1)THEN
      IA(I)=IALF(J)
      GO TO 10
    ENDIF
55  CONTINUE
  WRITE(*,100)IN(I) ~ 2151
100  FORMAT(3X,'IN(I)=' ,I3,' DOES NOT MATCH INTEGER',
C    ' FROM 0 TO 9 TYPE "Y" TO CONTINUE')
  READ(*,20)CH ~ 2154
20  FORMAT(A)
  IF(CH.NE.'Y'.AND.CH.NE.'y') [STOP] ~ 2157
10  CONTINUE ~ 2159
  RETURN ~ 2161
END
  
```

FIG.21A

2150

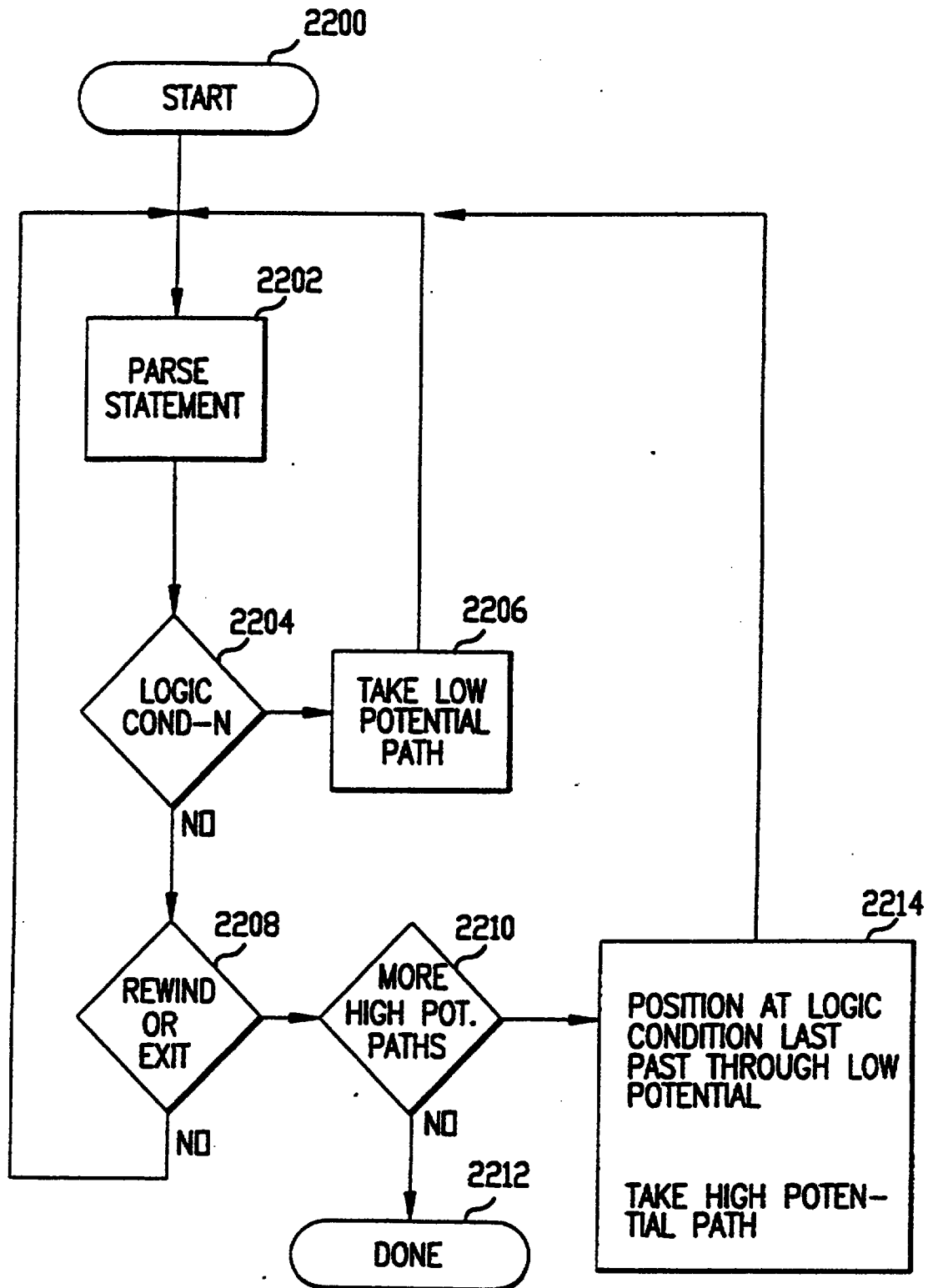
```

SUBROUTINE I2AXXX(IN,IA)
INTEGER IN (3)
CHARACTER IA(3),IALF(10)
DATA IALF/'0','1','2','3','4','5','6','7','8','9'/
001  FORMAT(3X,'IN(I)=' ,I3,' DOES NOT MATCH INTEGER',
C    ' FROM 0 TO 9 TYPE "Y" TO CONTINUE')
002  FORMAT(A)
  DO 3001 I=1,3
    DO 3002 J=1,10
      IF(IN(I).EQ.J-1)THEN
        IA(I)=IALF(J)
        0  CALL SUMI(14,2)
        GO TO 23001
      ENDIF
    0  CALL SUMI (6,2)
3002  CONTINUE
    WRITE(*,001)IN(I) ~ 2151
    0  CALL SUMI(20,2) ~ 2152
    0  CALL SUMI(21,2) ~ 2153
    READ(*,002)CH ~ 2154
    0  WRITE(M,002)CH ~ 2155
    IF(CH.NE.'Y'.AND.CH.NE.'y')THEN
      0  CALL SUMI(31,2) ~ 2156
      STOP ~ 2157
    ENDIF
23001 0  CALL SUMI(24,2) ~ 2158
3001  CONTINUE ~ 2159
    0CALL SUMI(35,2)
    RETURN
    END
  
```

FIG.21B

12 NOV 1997

- 23 -



This Sheet Has Been Replaced

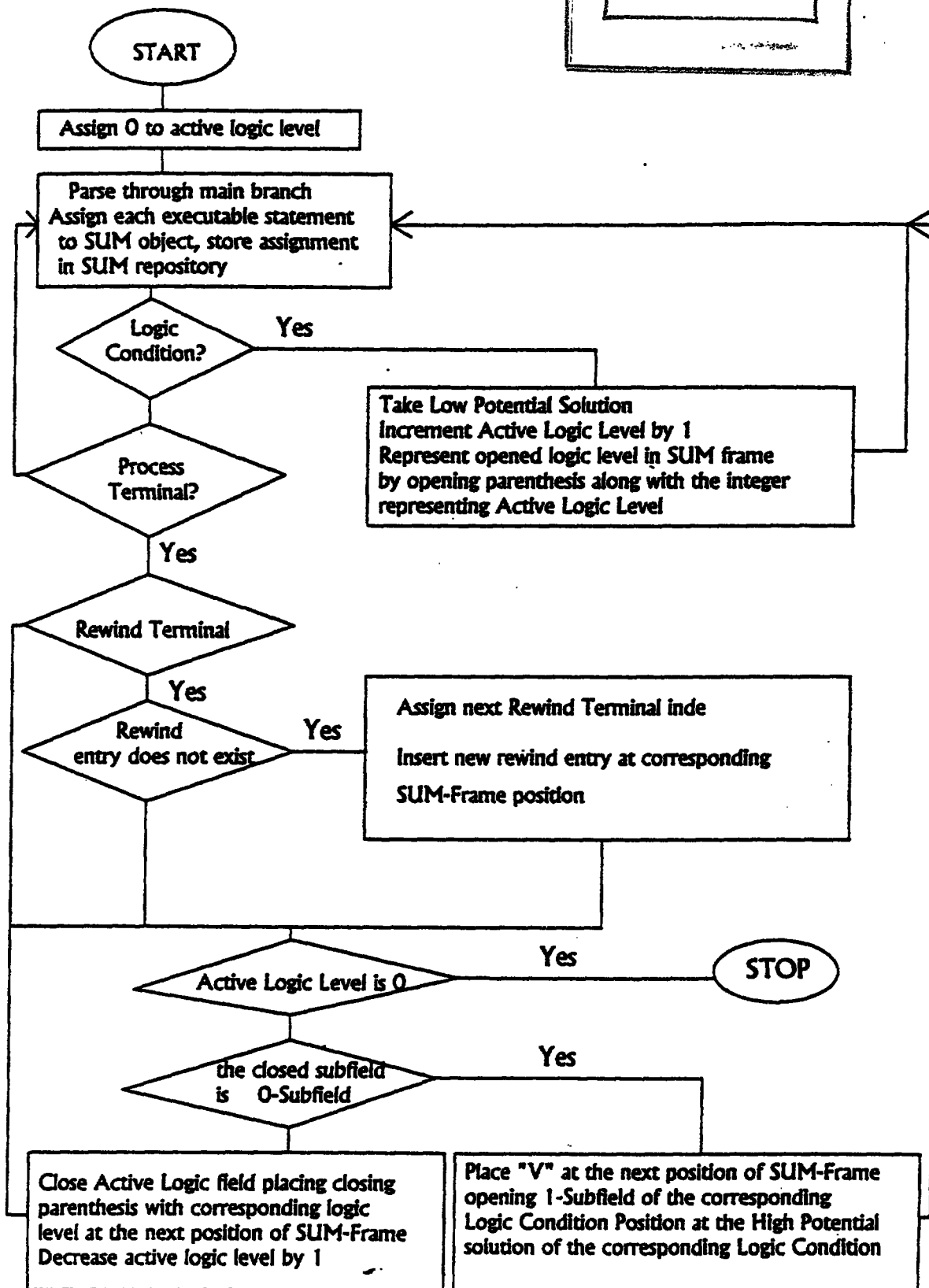


FIG. 22

THIS SHEET HAS BEEN
REPLACED

REPLACEMENT SHEET

12 NOV 1997



This Sheet Has Been Replaced



FIG 23

THIS SHEET HAS BEEN
REPLACED

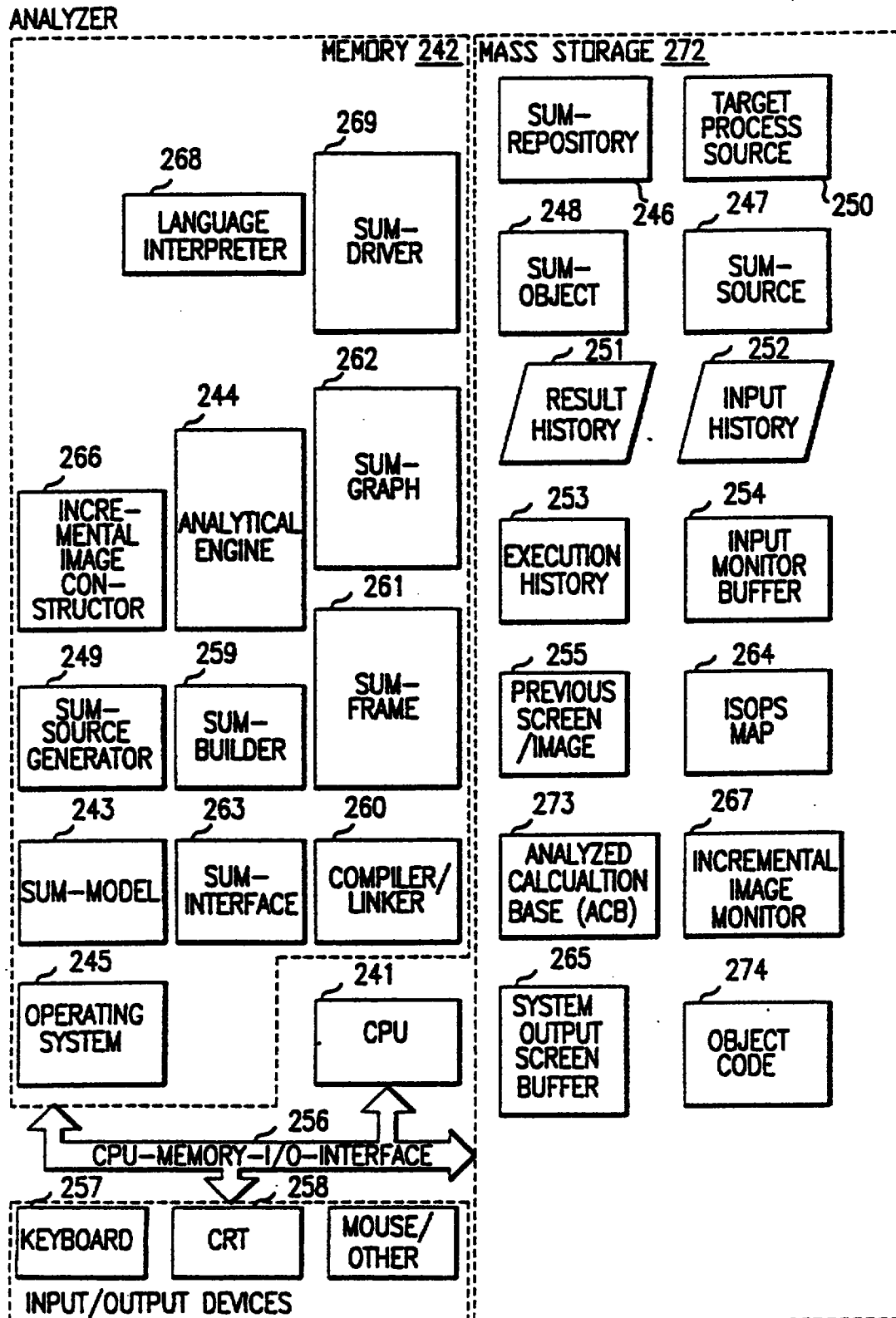
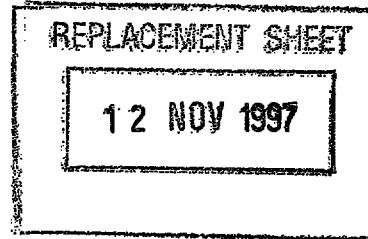


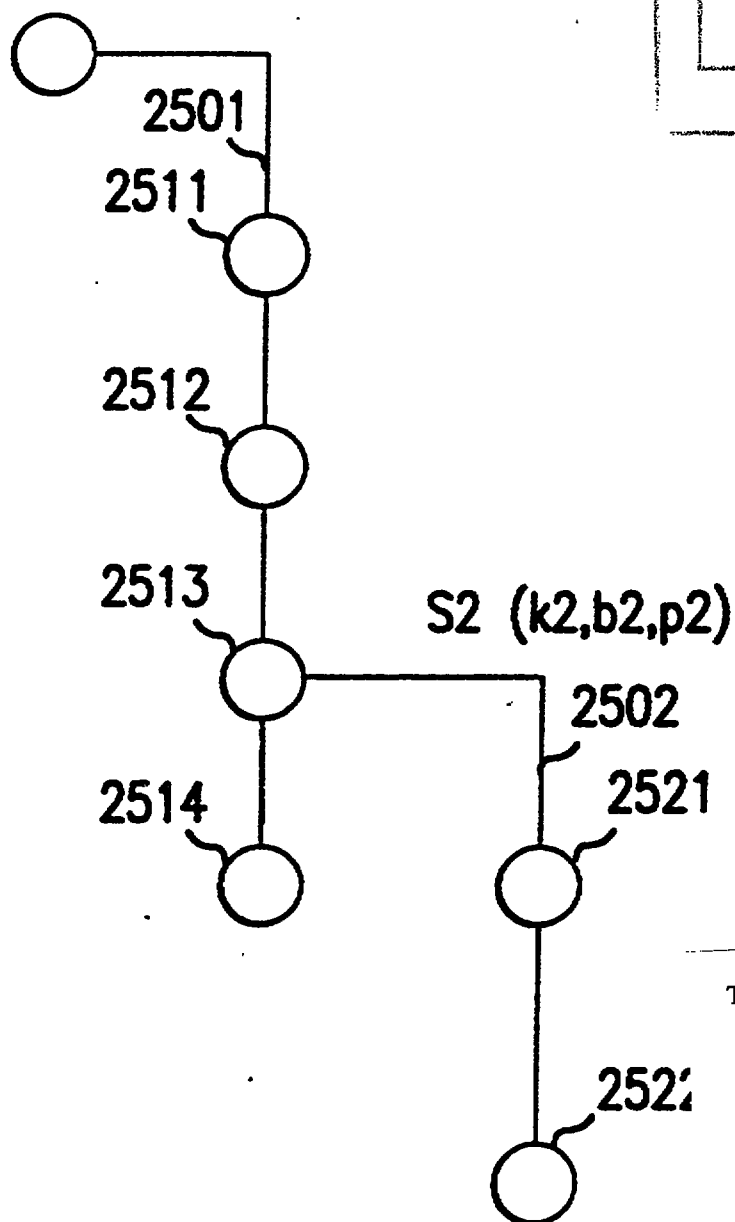
FIG.24

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

W (k,b,p)



REPLACEMENT SHEET

12 NOV 1997

This Sheet Has Been Replaced



2522

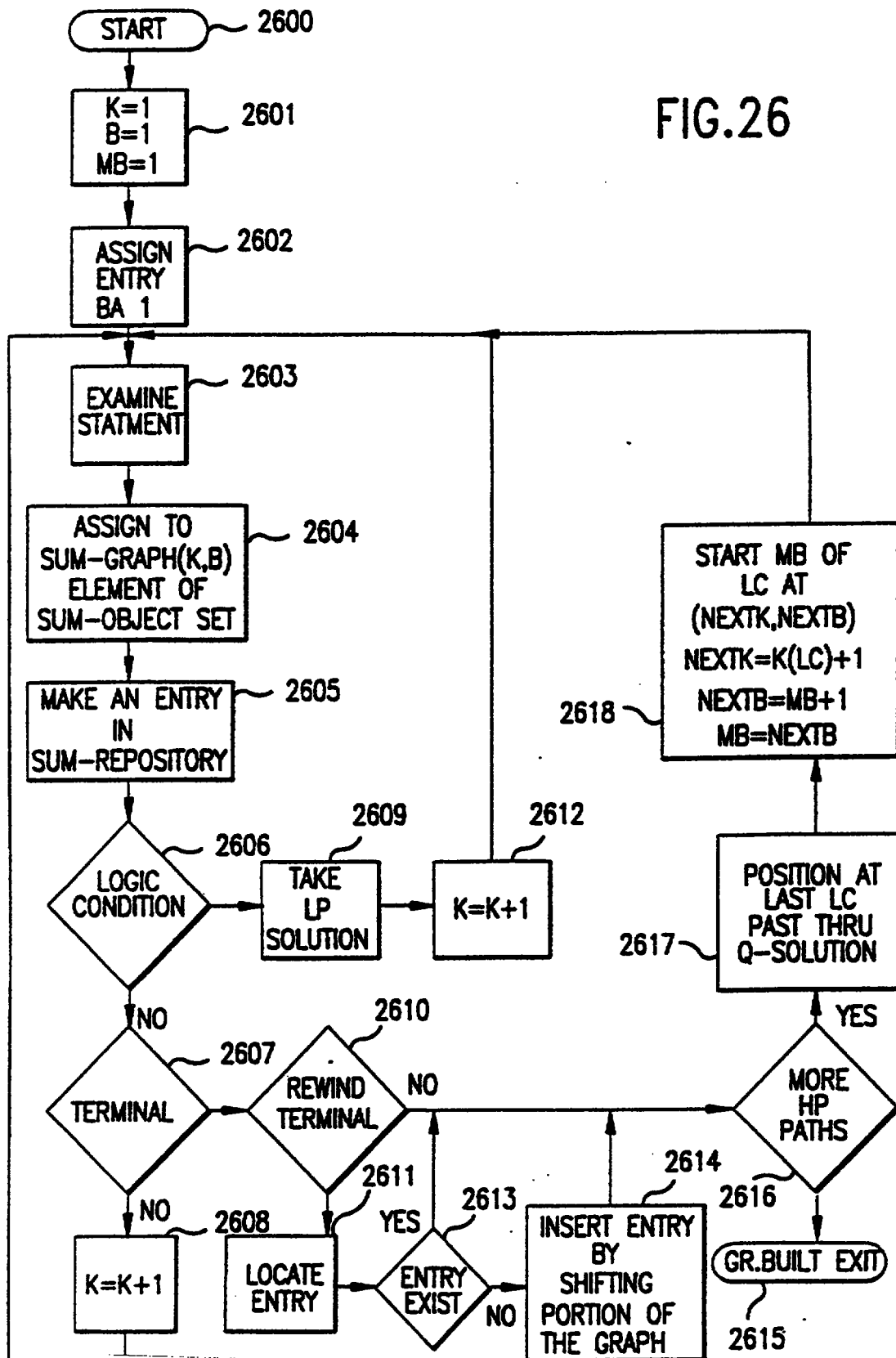
FIG.25

THIS SHEET HAS BEEN
REPLACED

12 NOV 1997

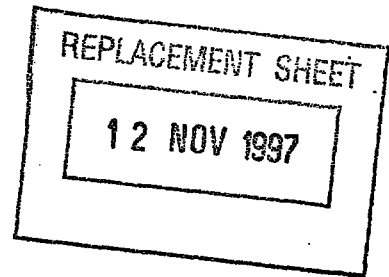
- 27 -

FIG.26



This Sheet Has Been Replaced

THIS SHEET HAS BEEN
REPLACED



ISOPS map sample format:

1.	2.	3.	4.	5.
W-	screen	screen	displ.	incremental
event -	position	area	in image	image
		changed	monitor	monitor
			file	file

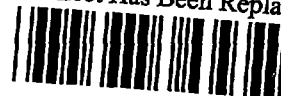
\ _____ ISOPS parameters file _____ / \ _____ IIM file _____ /

{k,b,p,m}	{sl,sc}	{l1,c1,l2,c2}	DDDDDD	'oooooooooooooooo'
{k,b,p,m}	{sl,sc}	{l1,c1,l2,c2}	DDDDDD	'ooooooo'
{k,b,p,m}	{sl,sc}	{l1,c1,l2,c2}	DDDDDD	'oooooooooooooooo'
{k,b,p,m}	{sl,sc}	{l1,c1,l2,c2}	DDDDDD	'ooooooo'
{k,b,p,m}	{sl,sc}	{l1,c1,l2,c2}	DDDDDD	'oooooooooooo'

**FIGURE
27**

THIS SHEET HAS BEEN
REPLACED

This Sheet Has Been Replaced



25 FEB 1998

METHOD AND APPARATUS FOR THE AUTOMATIC ANALYSIS OF COMPUTER SOFTWARE

BACKGROUND OF THE DISCLOSURE

1. Field of the Invention

The invention relates generally to the field of fault identification and analysis in computer software. More particularly, the invention relates to a method and apparatus used for the automatic analysis of computer software.

2. Background of the Invention

Since the day the first computer program was written and to this day, errors or faults in computer software are an expected and commonplace occurrence. Even though such faults are expected, fault identification and analysis is a problem which has forever plagued programmers. This due in part, to the fact that the fault rate for a carefully constructed program can be about 5 faults per 1,000 lines of source code so that a one million-line program may contain 5,000 faults! Compound this with the fact that computer software faults are different from computer hardware faults in that the software faults do not obey the same simplified rules of proximity of cause and effect. That is to say, a computer software fault does not necessarily have to appear close to its failure, but instead can be located almost anywhere within a much larger software system. Therefore, even when a fault is known to exist, determining the origin and location of that fault is extremely difficult at best.

3. Description of the Prior Art

As a result of the above mentioned characteristics of computer software, the prior art has developed a number of methods and apparatus to assist a programmer in the development and analysis of software. The methods and apparatus of the prior art encompass a wide array of systems and techniques, each generally applicable to a different period of the overall computer software development process. Among these prior art methods and apparatus are: 1) structured design techniques; 2) language-sensitive development tools; 3) highly-efficient and informative language interpreters and compilers; and 4) an assortment of program profilers and debuggers and emulators which assist in identifying and locating faults in software during the debugging and testing of the software. Despite all these tools, techniques and related efforts, a large amount of programmer time is spent in trying to understand how a program works and many failures are tolerated out of fear of further "breaking" a software system.

25 FEB 1998

- 2 -

There are many problems associated with existing tools and techniques for identifying and locating faults in computer software. Traditionally, a programmer had to debug or analyze a program through the use of some kind of "hook" or output statement which was part of the programming language used for the software program. For example, in one particular language a "PRINT" statement would be inserted by a programmer into one or more locations of a program being analyzed so that upon encountering the PRINT statement, the program being analyzed would output to an appropriate output device the contents of some registers, variables, etc. In this manner, the programmer would examine the output of the executed PRINT statement searching for clues as to why the program failed. Needless to say, this method of program analysis and debugging is very tedious for the programmer and error prone as well.

Another method of program debugging and analysis involves the use of a special program, known as a program debugger which monitors the execution of a software program and oversees its execution. Such debuggers typically offer enhanced flexibility and more information to the programmer than possible with the simple PRINT method, particularly with respect to a user interface presented to the programmer. Utilizing a debugger, a programmer will normally specify one or more lines of the software program as "breakpoints", or points within a software program which when encountered suspend the execution of the program so that the programmer may examine processor registers or program variables. Breakpoints are not limited to specific lines of code, breakpoints may be used to interrupt execution when an expression changes value, or when an expression reaches a value. Despite apparent advantages over the PRINT method, program debugging and analysis using known tools suffer from many of the same infirmities i.e., they do not automate the understanding of cause-effect relationship between faults and failures (manifestation of fault).

The first problem associated with these above mentioned debugging and analysis methods and tools is where to begin. This is largely a hit or miss proposition for the programmer. Therefore, the programmer will typically insert a number of breakpoints or PRINT statements which are scattered throughout the program under test. As the programmer narrows down his understanding of the nature and location of program fault(s) responsible for a failure, unnecessary breakpoints or PRINT statements are normally removed or deactivated. This technique of randomly inserting breakpoints or PRINT statements throughout a program is highly inefficient, time-consuming, and error-prone.

25 FEB 1998

- 3 -

Second, even if the programmer has some idea where within a program to place a breakpoint, the programmer must determine what to examine at that location. That is, a particular section of a program may use a large number of variables, registers or memory locations, thereby making the programmer's job of identifying the fault(s) more difficult. While debuggers and analyzers have certainly facilitated the examination of variables, registers or memory locations, they have by no means eliminated the problem of making a decision as to what variables, memory, or registers to examine and when to examine them. Further more, debuggers do not help to decide what is the meaning of the value of a variable and what other variables or what other statements need to be examined until the fault statement is found. As can be appreciated, the traditional techniques and tools used for debugging or analyzing a software program is not highly automated.

In an attempt to overcome these stated deficiencies in the art of program debugging and analysis, the prior art has developed additional tools and techniques.

United States Patent No. 4,853,851, "System For Determining The Code Coverage Of A Tested Program Based Upon Static And Dynamic Analysis Recordings" issued to Borsch on August 1, 1989 teaches a system for analyzing programs by measuring the degree of code coverage of a program being tested during specific test phases. The system disclosed utilizes a static and dynamic instruction flow indicator which permits a determination of the test cover results by correlating the data of the static and dynamic instruction flow. In this manner, the number of untested statements can be determined during a test phase of a computer program.

Although the more statements covered during the test of a program the better, the fact that a statement was executed does not provide any information on the statement validity. In particular, the statement may have been constructed wrong; the statement may be executed at the wrong place at the wrong time; or the statement may not contribute to the observed result.

United States Patent No. 4,802,116, "Programmed Controller" issued to Ward, et al on January 31, 1989 teaches programmed controller which controls a machine process and facilitates its debugging. The application programs disclosed are specifically designed to simulate specific control schemes for sections of the machine by being organized into one or more separate loops. A built-in control mechanism records the states of these loops along with the conditions responsible for the transitions between those states. This teaching is limited in application to only those processes that are specially coded by very

REPLACEMENT SHEET

25 FEB 1998

- 4 -

specific syntax rules as consisting of independent loops, where the means of tracing the states of these loops are pre-coded in the programs controlling the process.

A theoretical approach to the general problem of program analysis was disclosed by Thomas J. macabre in the article "A Complexity Measure" published in IEEE Transactions on Software Engineering, Vol. Se-2, No. 4, December 1976, pp. 308-320. Cyclomatic Complexity was proposed in this article as a measure of program complexity. This measurement is nothing more than the number of independent passes within a subroutine. This complexity measurement is useful to promote the design of less sophisticated software routines which are also easier to test. However, counting the number of independent passes does not eliminate their necessity

Despite these and other attempts made by the prior art, software debugging and analysis remains a difficult, time-consuming and error prone process. A critical shortcoming in the prior art is that there are no teachings for general methods or apparatus which are capable of automatically determining the location of logic faults in software programs even within a broad area of a program. This is due, in large part, to a general belief which exists in the art that such automation is not possible because software does not conform to the rules of proximate cause and effect and that a software program has no knowledge of what results are expected of it unless those expected results are pre-coded within the program. Such pre-coding is itself programming and therefore is also error-prone.

Therefore, a continuing need exists in the art for a method and apparatus which more highly automates the chore of software program debugging and analysis.

SUMMARY OF THE INVENTION

It is therefore the object of the present invention to provide a method and apparatus which automates the process of program debugging analysis and maintenance by allowing for automatic identification of an area of a program, which area contains a fault(s) responsible for specific failure or contains the statements to be modified in order to comply with the new requirements to the program. The mentioned area is automatically and continuously minimized while the invented system automatically learns about the target software program.

REPLACEMENT SHEET

25 FEB 1998

- 5 -

The invention presents, what will be referred as "Software Understanding Machine", that is an apparatus and method that build and propagate the knowledge about the internal processes within the target software program.

The invention thus presents the advantage that the method and device for debugging and analyzing software programs does so with precision that grows with the accumulation of knowledge about the target or analyzed program behavior. The invention successively accumulates this knowledge, and utilizes it to determine correct or uncertain behavior of the analyzed software program. The invention further shows the ability to partially "unlearn" previously acquired knowledge, thereby recognizing that correct results are a function of a user's expectations of the results which may change over time.

The invention operates by analyzing the results or effects produced by a software process and automatically traces back from these effects to their cause(s). In this manner failures, which are unexpected results, are automatically traced back to the fault(s) responsible. This is in contrast to traditional methods, i.e., stepping forward through the software process to precipitate a failure. The analysis is accomplished by the invention of the present application by first building a model of the software process under analysis on the basis of a static analysis performed under a set of strict rules. The model is constructed such that it serves as a framework for the further accumulation of knowledge which is acquired through a dynamic analysis of the software process.

The invention constructs the model of the target software process in such a manner that allows for the automatic determination of accessibility, or presence of a path, between any two elements of the target software process. Accessibility is required for dependency, that is, if one statement within a software process is accessible from another statement within the process, then, and only then, the correctness of the other statement may affect the behavior of the one statement under examination. If the second statement is not accessible from the first statement, then correctness/incorrectness of the first statement cannot affect the behavior of the second statement and therefore the second statement is not dependent upon the first statement.

During the construction of the model of the target software process, the invention extracts the control component of the target software process into a form which represents a control structure of that target process. Furthermore, the invention accumulates information about the behavior of the target software process thereby permitting the invention to determine the correctness of a software process element relative to the time at which the

25 FEB 1998

- 6 -

element is executed. Still further, this accumulated correctness information is propagated between target process elements (statements) thus automatically increasing the intelligence of the target process model. Through this accumulated knowledge that the invention acquires about the correctness of the software process behavior, the invention is able to automatically determine the location of fault(s) responsible for a failure.

These resources maintained by the inventive software analyzer, or "Software Understanding Machine", are presented to the user in an easy-to-use implementation. The preferred embodiment makes extensive use of an intuitive user interface as well as multimedia capabilities known in the art to further facilitate software debugging and analysis.

It is still another object of the present invention to introduce an original form of the target process object code, SUM-Object code, and a method for its execution. The SUM-Object code is more suitable for program analysis, both static and dynamic and for program maintenance. It eases and sometimes eliminates the traditional recompiling and re-linking of the target process code which are traditionally necessary in order to implement modifications to the target process. The SUM-Object code comprises the model extracted from the target process. Therefore, increasing the intelligence of the model performed by the "Software Understanding Machine" thus increases the intelligence of the resulting SUM-Object code.

BRIEF DESCRIPTION OF THE DRAWINGS

The teachings of the present invention can be readily understood by considering the following detailed description in conjunction with the accompanying drawings, in which:

FIG. 1 shows a flow chart of a sample target process.

FIG. 2 shows an SUM-Frame of the sample target process of FIG. 1.

FIG. 3 shows how complex logic conditions are reduced into two or more reduced logic conditions.

FIG. 4 shows a reduced flow chart of the sample target process shown in FIG. 1.

FIG. 5. shows SUM-Graph of the sample target process shown in FIG. 1 and the SUM-Binary addresses assigned to the corresponding elements of the SUM-Graph..

25 FEB 1998

- 7 -

FIG. 6 shows how typical IF-THEN-ELSE program constructs are represented in SUM-Frame and SUM-Graph form(s).

FIG. 7 shows the steps required to construct the SUM-Model, SUM-Repository and SUM-Source presented next to the main components of the Software Understanding Machine (SUM).

FIG. 8 shows the main components of the Software Understanding Machine (SUM) operating in the traditional object code implementation.

FIG. 9 shows the construction of exit access indicators and the construction of propagated vision points.

FIG. 10 shows the construction of the Propagated Vision Point of a process terminal.

FIG. 11 shows a target process with faults and their potential origin.

FIG. 12 shows three different types of target process failure(s).

FIG. 13 shows how (SUM) minimizes the area of uncertainty of the position of a fault within a target process, which fault is responsible for a specific failure of the target process.

FIG. 14 shows terminated event definition space.

FIG. 15 shows the propagation of correct algorithm space.

FIG. 16 shows the accumulation of knowledge about the correctness of a process element located within a specific process address.

FIG. 17 shows the knowledge propagation process implemented by constructing event definition spaces built only in specific directions and only up to the knowledge propagation terminals.

FIG. 18 shows the main components of SUM operating in the SUM-Object code implementation, the two components which comprise SUM-Object code and an alternative implementation of SUM through interpretation of the SUM-Repository by the SUM-Driver controlled by the SUM-Model.

FIG. 19 shows the construction of an analyzed calculation base.

25 FEB 1998

- 8 -

FIG. 20 shows an SUM-Graph, SUM-Repository of a sample process.

FIG. 21 shows a sample process and its SUM-Source code.

FIG. 22 is a flow chart depicting the steps which comprise the Next Lowest Potential method or rule of parsing a target process.

FIG. 23 is a pseudocode listing which describes the steps necessary to create an SUM-Frame of the target process by the next lowest potential method.

FIG. 24 is a system block diagram showing the overall analysis system and the components parts which it comprises.

FIG. 25 shows the construction of event definition space terminated by knowledge propagation terminals.

FIG. 26 is a flow chart depicting the steps necessary to create an SUM-Graph of a target process by the next lowest potential method.

FIG. 27 shows a sample format of Incremental System Output Process Synchronized map (ISOPS map).

To facilitate understanding, identical reference numerals have been used, where possible, to designate identical elements that are common to the figures.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

After considering the following description, those skilled in the art will clearly realize that the teachings of this invention can be readily utilized in the analysis of software programs or processes.

FIG. 24 is a block diagram of the Software Understanding Machine (SUM) analysis system 240 which comprises the present invention. Some of the elements contained within the analysis system are well known in the art as they are typically found in conventional personal computers. Well known components of the analysis system includes, keyboard 257, a CRT display 258, CPU 241, memory 242, and some form of mass storage facility 272 (e.g., a hard disc drive). These systems communicate via and are frequently capable of receiving a wide array of options via interface bus 256.

25 FEB 1998

- 9 -

It must be noted, that whether different components of the system reside on the mass storage or in the memory is often not important and is frequently controlled by the operating system.

A target software process 250 is defined by a target source program comprising one or more program statements. The target source is typically stored on mass storage device 272. Target process 250 is extracted from target source by SUM-Builder program 259. The SUM-Builder 259 examines the loaded target process 250 and constructs an SUM-Graph 262 of that target process 250.

An SUM-Repository 246 of the target process is built by the SUM-Builder 259 at the same time the SUM-Graph 262 is constructed. The SUM-Repository 246 so constructed is "control-less", in other words, it does not contain any process control statements which were present in the target process 250. Instead, process control information is preserved within the SUM-Model 243.

The SUM-Model 243 comprises one or more elements, each element corresponds to an executable statement of the target process 250. Multiple SUM-Models 243 are constructed when the target process 250 comprises multiple components, i.e., main, subroutines, functions, etc.

After the SUM-Model 243 and SUM-Repository 246 are built, SUM-Source Generator 249 constructs SUM-Source code 247 from the information contained within SUM-Model 243 and SUM-Repository 246. The SUM-Source 247 is representative of the original target process 250, since it is constructed from SUM-Model 243 and SUM-Repository 246 which in turn, were constructed from the target process 250.

The SUM-Source 247 is only created when SUM-Analyzer 240 is used in its traditional object code embodiment. The SUM-Source 247 permits the SUM-Analyzer 240 to operate with the traditional, object code representation of target process 250, which should be very familiar to those skilled in the art. The SUM-Source 247 so constructed contains "synchronization signals", i.e., CALL() statements or WRITE() statements which are inserted into SUM-Source 247 by SUM-Source Generator 249.

The synchronization CALL() statements reference the SUM-Interface library 263. The SUM-Source code 247 is compiled by compiler/linker 260 and the resulting code is linked with SUM-Interface library 263 thereby resolving the CALL() references. The output code produced by the

25 FEB 1998

- 10 -

compiler/linker 260 operating on SUM-Source code 247 and SUM-Interface library is then executed by CPU 241. During the execution of this code, the synchronization statements embedded into SUM-Source and now compiled/linked, are executed which, in turn invoke the execution of SUM-Analytical Engine 244.

The creation of SUM-Source 247 and compiling/linking it with SUM-Interface library in order to send synchronization signals to the SUM-Model is not done in the embodiment utilizing the SUM-Object code or in the embodiment utilizing a language interpreter operating on SUM-Repository. In the two last mentioned embodiments the model controls the execution of the target process.

In a first embodiment utilizing traditional object code and upon receipt of a later synchronization signal from the SUM-Interface 263, SUM-Analytical Engine 244 traces the SUM-Model 243 from the point at which a previous synchronization signal was received up to the position within the SUM-Model 243 where the later synchronization signal was received. In this manner, the SUM-Analytical Engine 244 effectively "models" the target process by "executing" SUM-Model 243.

In the other two embodiments, i.e. SUM-Object Code 243 embodiment or SUM-Repository 246 interpretation by the Language Interpreter 268 embodiment, the SUM-Model 243 is not "executed" after the target process by the SUM-Analytical Engine 244, but instead, the SUM-Model is "executing" the target process by controlling the SUM-Driver 269, that passes corresponding elements of Executable Component of SUM-Object Code 248 for execution or corresponding statements of SUM-Repository 246 for interpretation. In these two embodiments the synchronization signals representing the state of the SUM-Model 243 are passed to SUM-Interface 263 by SUM-Driver 269.

In all of the mentioned embodiments the SUM-Analytical Engine 244 constructs the Analyzed Calculation Base (ACB) 273 by tracing effects to their cause(s). This process is called in the present invention a Knowledge Induction process. It is done in the direction opposite to the direction of the target process execution and only as far as defined by the Knowledge Propagation Terminals, that will be explained further. In this manner, the knowledge of the correctness or uncertainty of the effect of one statement execution event is propagated backwards to define the knowledge of the correctness/uncertainty of the effect of another statement execution event. The ACB 273 comprises Analyzed Calculation Words (ACW) which describe the knowledge of the correctness/uncertainty of target process elements (statements

25 FEB 1998

- 11 -

and parameters within the statements) as a function of time within the process.

System Execution History, 253, System Input History 252 and System Result History 251 are all produced during the execution of the SUM-Model 243. These structures will be described in greater detail later in this description.

The system receives user input through keyboard 257 or other input device and informs the user of analysis results and/or progress via CRT 258.

FIG. 1 is a flow chart representing a sample target source process 250 that is a single module although our results can be easily extended to a process comprising a system of modules. The module could be either a main program or a single subprogram (i.e., subroutine, function, or paragraph) with one entry point. Although the example shown has only a single exit, the invention will work in an identical manner for modules with multiple exits. Those skilled in the art could readily extend our teachings to accommodate modules having multiple entry points.

During the initial examination of target source 250 by SUM-Builder 259, SUM-Model 243 may be built from two predecessor structures, SUM-Frame 261 and SUM-Graph 262. A representative SUM-Frame, so constructed from the target source 250 is shown in FIG. 2.

In the following description of the construction of the SUM-Frame, the below given designations and definitions will be used:

Table 1 SUM-Object Set

Designation	Object	Example
Process Entry	[	
End of process description	]	
Arithmetic or Assignment Statement	A	
Function Call other than Graphics Library	X	CALL, PERFORM
Logic Condition	L	IF
Loop Logic Condition	D	DO, WHILE, FOR
User Input Statement	E	READ
System Output Statement	W	WRITE(); CALL MOVETO(x,y,xy)
Null Statement	N	
Stop	=	STOP
Return	R	RETURN, EXIT

25 FEB 1998

- 12 -

Table 1 (continued)

Unconditional Branch	*	GO TO
Loop Terminal Statement	C	CONTINUE; ENDDO
Assigned GO TO	+	
ASSIGN	S	
Entry of GO TO	/	
ENDIF	!	
Opening of Logic Condition	(	
Closing of Logic Condition	)	
Logic Condition Alternative	V	ELSE; OR

Definition F1: A Reduced Logic Condition (LC) is a process element with one control input and two control outputs. One of the two control outputs are assigned to Boolean 0 (False) and the other control output is assigned to Boolean 1 (True) which correspond to a solution to the logic condition.

Definition F1.1: The Low Potential Solution of a logic condition is the "0" Boolean solution (False) to the logic condition.

Definition F1.2: The High Potential Solution of a logic condition is the "1" Boolean solution (True) to the logic condition.

Definition F2: The Process Terminals are comprised of the following members of the SUM-Object set shown in Table 1: *, +, C, R, =.

Definition F3: The Main Branch of a process is the path traversed through a process from an origin of the main branch (process entry or high potential solution of logic condition) to a process terminal when all of the logic conditions which comprise that path are passed through their low potential solutions.

Referring to FIG. 9, the Main Branch is the path from entry 900, through logic conditions 904, 914 and 924, until terminal 930 is reached.

Definition F4: Exit Terminals result in an exit from a process and are comprised of the following members of the SUM-Object set shown in Table 1: R, =. As shown in FIG. 9, terminal 960 is an exit terminal.

25 FEB 1998

- 13 -

Definition F5: Rewind Terminals are terminals that rewind the process to an element that is in the same main branch or in a main branch to the left of the current main branch. An exception to this rule is when the rewind terminal is an exit of a loop. In such a case, the rewind terminal could reset the process to an element that is within a branch located to the right of the current branch. As shown in FIG. 9, terminals 908, 930, 944, 950, 952 and 958 are all rewind terminals.

Definition F6: A t-pass is a pass from the process entry to the process terminal. In any process there are $(n+1)$ t-passes, where n is the number of logic conditions in the process. Each t-pass is uniquely identified by its terminal.

In FIG. 9 there is shown a single entry 900, and seven terminals; 908, 930, 944, 952, 950, 960, and 958. Of those seven, 930, 944, 952, 950, 958, and 908 are all rewind terminals while terminal 960 is an exit terminal. Shown further in FIG. 9 are six logic conditions; 904, 914, 924, 938, 946 and 956. A single t-pass would be represented by entry at 900, progressing along line 902, taking the high potential path from logic condition 904 along line 906 until terminal 908 is reached. This t-pass would be uniquely identified as, for example, t-pass 908 where 908 identifies the terminal of the t-pass.

Definition F7: A branch of a logic condition is that part of a t-pass which begins at that logic condition. Each logic condition has at least one 0-branch and at least one 1-branch. Referring once again to FIG. 9, a high-potential or 1-branch of logic condition 904 is represented by line 906 and terminates at terminal 908. There are two 1-branches of logic condition 914 and they are ended by terminals 960 and 958.

Three of the four elements contained within dotted block 964, namely 954, 956, and 960, constitute a main branch of logic condition 914. The other element contained within dotted block 964, namely 958, constitutes a main branch of logic condition 956. The combination of all the main branches of a process necessarily and sufficiently represent all the elements which comprise that process. Further, there could exist many paths through a process element but that process element can belong to only one main branch.

Definition F8: The 0-subfield of a logic condition is a combination of all elements situated on the 0-branch(es) of that logic condition.

25 FEB 1998

- 14 -

Definition F9: The 1-subfield of a logic condition is a combination of all elements situated on the 1-branch(es) of that logic condition.

Definition F10: The field of a logic condition is a combination of the 0-subfield and 1-subfield of that logic condition.

Definition F11: The level of the logic condition field or logic level, is the numeric order in which that logic condition was encountered in the t-pass going through the logic condition.

Referring once again to FIG. 9, logic condition 904 is situated at logic level 1. Logic condition 914 is situated at logic level 2. Logic conditions 924 and 956 are both situated at logic level 3. Logic condition 938 is situated at logic level 4. Logic condition 946 is situated at logic level 5.

Definition F12: A process segment is that part of a main branch which is positioned between process entry, process terminals and logic conditions.

Process terminals and logic conditions terminate process segments. They belong to the process segments that they terminate.

An SUM-Frame is constructed in the present invention by representing the main branches of the target process in a special order along with representing the levels of the logic conditions and their corresponding fields.

Construction of the SUM-Frame

The first step performed by SUM-builder 259 is the construction of the SUM-Frame which is representative of the target process 250. This step is not needed if the SUM-Graph is constructed directly from the target process. The target process is processed by the SUM-Builder through the "Next Lowest Potential" rule or alternatively, through the complementary "Next Highest Potential" rule. FIG. 22 is a flow chart depicting the steps associated with the analysis of a software process by the next lowest potential rule. The SUM-Builder proceeds by beginning its analysis at start block 2200. Successive statements of the process are parsed by block 2202 until a logic condition or process terminal is met. If a logic condition is met as determined by block 2204, the low potential solution of this logic condition is taken as dictated by block 2206. When a rewind or exit terminal is encountered as determined by block

25 FEB 1998

- 15 -

2208 and untraversed high potential paths remain in the software process as determined by block 2210, the block 2214 will be executed. The SUM-Builder locates the logic condition, that was last passed through its low potential solution and which high potential path was not yet taken. SUM-Builder then takes its high potential path and proceed via block 2202, therefore building the Main Branch of that logic condition.

These steps repeat until no untraversed high potential paths are left in the target software process.

In order to provide the reader with additional clarity, FIG. 23 shows a PSEUDOCODE implementation of the steps involved in analyzing a target software process by the next lowest potential rule. Additionally, FIG. 23 shows at what points within the steps modifications are made to the SUM-Frame under construction and at what points in the process statements are stored in the SUM-Repository.

FIG. 2 shows an SUM-Frame 20 which was constructed from the flow chart of a target process shown in FIG. 1 by the next lowest potential rule. By inspection of FIG. 2, the reader can readily see that the SUM-Frame 20 is built in the form of an algebraic expression containing several levels of parenthesis. In SUM-Frame 20, the parenthesis are representative of the logic levels. The same general rule is applied to this SUM-Frame 20 that applies to any algebraic expression, that is, the lower (outer) level logic field cannot be closed until the higher (inner) level logic field is first closed.

Referring once again to FIG. 2, the different levels of SUM-Frame 20 become apparent. Dotted line 21 identifies that portion of SUM-Frame 20 which is the first level. Line 22 represents the second level. Dotted lines 23 show the third level. Dotted line 24 shows the fourth level and dotted line 25 shows the fifth level. Within the SUM-Frame, the fields of logic conditions are between the corresponding parenthesis and integers corresponding to the parenthesis represent the logic levels of those fields.

The labeling of the target process elements which are represented within the SUM-Frame 20 proceeds using the SUM-Object set members previously defined. Table 2 shows the SUM-Object set members of the resulted SUM-Frame and their respective process segments.

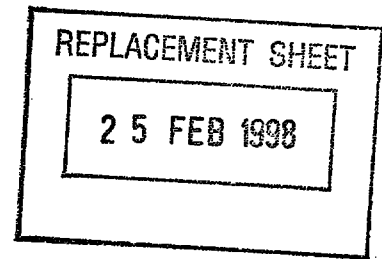


Table 2.

Process Segments	SUM-Object Set Representation
1	L1
2	/7, A1, /1, L2
3	A2, L3
4	X1, *1
5	A3, /2, L4
6	A4, /3, X2, *2
7	/4, A5, /6, L5
8	A6, *3
9	W1, *4
10	A7, L6
11	R1
12	*6
13	E1, *7

Upon completion of the construction of the SUM-Frame 20, an SUM-Graph is built by the SUM-Builder 259. An SUM-Graph of target process 250 shown as a flow chart in FIG. 1, is shown in FIG. 5(A). The SUM-Graph is constructed along two perpendicular axis which are designated as k and b in the graph and are referenced by 501(A) and 500(A) respectively. The origin of the two perpendicular axis is at the top-left of the graph.

Main branches of the graph are directed vertically down and represent tracks by which the target process 250 execution progresses, unless the track is switched as the result of the high potential solution to a logic condition encountered in the track. As shown in FIG. 5(A), every track has a unique b coordinate. Every track of the graph is parallel to one another.

The graph is constructed and populated such that an SUM-Object set member representative of the target process 250 is inserted at appropriate k,b coordinate(s). For example, as shown in FIG. 5(A) at k,b coordinate 1,1 there is placed an SUM-Object set member '[' (left brace) which is indicative of a process entry as defined previously in Table 1.

In the following description of the construction of the SUM-Graph, the below given designations and definitions will be used:

Definition G1: A Reduced Flow Chart is an SUM-Graph represented in the form of a flow chart. FIG. 4 shows a reduced flow chart for the target process represented by the flow chart of FIG. 1.

25 FEB 1998

- 17 -

A reduced flow chart is presented along two perpendicular axis, k and b which are shown as 401 and 400 in FIG. 4, respectively. The origin of the two axis is at the top-left-most portion of the chart. The k axis increases as one proceeds down the chart and the b axis increases as one proceeds to the right of the chart.

All possible changes in the direction of control is reduced to one - to the right as the chart is traversed. In this manner, all high potential solutions to a logic condition are represented as a shift to the right in the graph. Only backward (up and left) jumps of unconditional control statements are possible when a software process is represented by a reduced flow chart. An exception to this general rule is an exit from a loop. Such an exit may be made forward.

Definition G2: The negative state of a logic condition is the state of a logic condition which, when evaluated results in its 0-solution or low potential solution. With reference to FIG. 5(A), logic condition L1 is represented at k,b coordinate 2,1. Traversing the SUM-Graph shown in FIG. 5(A) through L1 with L1 in its negative state would result in relocating to k,b position 3,1 which is represented by SUM-Object set member /7 in the graph.

Definition G3: The positive state of a logic condition is the state of a logic condition which, when evaluated results in its 1-solution or high potential solution. With reference to FIG. 5(A), traversing the SUM-Graph shown in FIG. 5(A) through L1 with L1 in its positive state would result in relocating to k,b position 3,7 which is represented by SUM-Object set member E1 in the graph.

Definition G4: The binary address (BA) of an SUM-Graph element shows that elements position relative to process logic constructs and is represented by the integers 1 and 0 where the integers comprising the BA represent the solutions of the logic conditions in the t-pass up to that element with the process entry assigned the binary address of 1.

The binary address of those elements within the same process segment are the same. The binary address of a 0-segment of a logic condition is generated by shifting the address to the left by one bit position and placing 0 at the rightmost position of the binary address. The binary address of a 1-segment of a logic condition is generated by shifting the address to the left by one bit position and placing a 1 at the rightmost position of the binary address.

25 FEB 1998

- 18 -

Referring once again to FIG. 5(A), the binary address of k,b position 8,1 (L3) is 100. This is constructed from the process address (binary address 1), the negative solution (0) to the logic condition L1 located at k,b position 2,1 and a negative solution to logic condition L2 located at k,b position 6,1 in the graph. As a further example the binary address of A7 (located at k,b position 7,5 in FIG. 5(A)) is 101. This is constructed from the process address (binary address 1), the negative solution (0) to the logic condition L1 located at k,b position 2,1 in the graph and the positive solution (1) to the logic condition L2 located at k,b position 6,1 in the graph.

Construction of the SUM-Graph

The SUM-Graph 262 which is representative of target process 250 is built by the same next lowest potential rule previously described. The SUM-Graph 262 may be built from a previously constructed SUM-Frame 261 or built directly from analyzing the target process 250.

FIG. 26 is a flow chart depicting the steps associated with the analysis of a software process by the next lowest potential rule and the construction of the resulting SUM-Graph. The analyzer proceeds by beginning its analysis at start block 2600. The position on the graph is assigned to k,b position of 1,1 2601, and the entry binary address is assigned to 1 as indicated in block 2602. Successive statements of the process are examined by block 2603 with the SUM-Graph(k,b) position being populated with the SUM-Object set member representative of the examined statement as required by block 2604. The repository is updated in block 2605 and if the statement under examination presently is a logic condition 2606, then the low potential solution to the logic condition is taken 2609, the present k,b position is updated by $k = k+1$ as stated in block 2612 and the statement reexamination block 2603 is reentered.

If upon examination by block 2606 it is determined that the statement is not a logic condition, then the statement is tested in block 2607 to see if it is a terminal. If the statement so tested is not a terminal statement then the present k,b position is updated by $k = k+1$ (block 2608) and the statement reexamination is reentered.

If the statement examined by block 2606 is not a logic condition and it is a terminal (block 2607) then if it is a rewind terminal other than a loop terminal the rewind entry is located. If it does not exist, it is inserted in the SUM-Graph (block 2614). A determination is made at block 2616 as to whether any further untraversed high potential paths exist (i.e., a logic condition for which a main branch is not yet built). If there are no remaining untraversed high

REPLACEMENT SHEET

2 5 FEB 1998

- 19 -

potential paths in the target process under examination, then the graph is built and the SUM-Builder terminates 2615.

If there were untraversed high potential paths as determined by block 2616, then the SUM-Graph position is updated to the last logic condition passed through its 0-solution and not yet passed through its 1-solution (block 2617) and a new main branch of that logic condition is begun at SUM-Graph position(nextk, nextb) where nextk is assigned to k(logic condition) + 1 and nextb is assigned to the last used b+1 (block 2618). The last used b is updated and the process then continues with the examination statement 2603.

FIG. 5(A) shows an SUM-Graph constructed by the SUM-Builder 259 using the above stated method of the present invention. Once the target process 250 is represented as SUM-Graph 262 as constructed by the SUM-Builder 259, a number of useful characteristics of SUM-Graph 262 as shown in FIG. 5(A) emerge.

By examination of the SUM-Graph shown in FIG. 5(A) it should be apparent that every element represented within the SUM-Graph may be addressed by a unique coordinate pair (k,b). For example, logic condition L1 is located at k,b position 2,1 in the graph. Logic condition L6 is located at k,b position 8,5 in the graph. Table 3 shows the elements, their corresponding and unique k,b coordinates and their resultant binary addresses which are created during the analysis of target process 250.

In the present preferred embodiment, the SUM-Graph is built in only two directions - down and to the right as one looks at the graph. When the SUM-Graph so constructed is traversed, the traversal proceeds down the graph and can only be interrupted by shifting to the right as a result of a high potential solution to an encountered logic condition.

Referring once again to FIG. 5(A) the graph traversal would begin at k,b position 1,1 (process entry) and proceed through logic condition L1 (k,b position 2,1), through label /7 (k,b position 3,1), through arithmetic A1 (k,b position 4,1) and so on until terminal *1 (k,b position 10,1) is reached. Such a traversal would proceed in a single direction, down the graph. If however, a high potential solution to a logic condition was passed through during traversal then a shift to the right would have taken place. For example, during the above traversal, if the logic condition L1 (k,b position 2,1) were passed through its high potential solution then a shift in flow would take place to E1 (k,b position 3,7).

25 FEB 1998

- 20 -

Further, every segment is assigned a binary address that uniquely identifies the logic position of that segment within the target process 250. FIG. 5(B) shows the binary addresses which are assigned to the elements which are contained within SUM-Graph depicted in FIG. 5(A). For example, logic condition L1 (k,b position 2,1) is assigned to binary address 1, logic condition L2 (k,b position 6,1) is assigned binary address 10, output statement W1 (k,b position 16,4) is assigned binary address 100111. Table 3 shows the binary address which correspond to the elements which comprise target process 250.

Table 3

SUM-Object	Set Member	(k,b) Position	Binary Address
[	1,1	1	
L1	2,1	1	
/7	3,1	10	
A1	4,1	10	
/1	5,1	10	
L2	6,1	10	
A2	7,1	100	
L3	8,1	100	
X1	9,1	1000	
*1	10,1	1000	
A3	9,2	1001	
/2	10,2	1001	
L4	11,2	1001	
A4	12,2	10010	
/3	13,2	10010	
X2	14,2	10010	
*2	15,2	10010	
/4	12,3	10011	
A5	13,3	10011	
/6	14,3	10011	
L5	15,3	10011	
A6	16,3	100110	
*3	17,3	100110	
W1	16,4	100111	
*4	17,4	100111	
A7	7,5	101	
L6	8,5	101	

25 FEB 1998

- 21 -

Table 3 (continued)

R1	9,5	1010
*6	9,6	1011
E1	3,7	11
*7	4,7	11

Lastly, once SUM-Graph 262 construction is completed by SUM-Builder 259, the accessibility between two elements of target process 250 may be effectively determined. Accessibility between two elements s1, and s2, is defined as the presence of a path from s1 to s2. That is, s2 is accessible from s1 if there is some possible input data combination that results in s2 being executed some time after the execution of s1. Accessibility between two process elements is required if one element is capable of effecting the result of the second element.

Referring once again to FIG. 5(A), it becomes obvious that statement A7 (k,b position 7,5) is not accessible from statement W1 (k,b position 16,4). That is because there is no possible path from statement W1 that would proceed through statement A7. Statement A7 is however, accessible from statement X1 (k,b position 9,1). That is because statement *1 (k,b position 10,1) repositions the path taken to /1 (k,b position 5,1) which immediately precedes statement L2 (k,b position 6,1). The high potential solution of L2 leads directly to statement A7. Therefore, the present invention is able to determine whether statement(s) or elements of target process 250 could be influenced by the correctness of a particular statement or statements within that process. When the target process 250 is represented in SUM-Graph form as shown in FIG. 5(A) and is used subsequently for the construction of SUM-Model 243, the invention is capable of evaluating the possibility for a statement s1 of target process 250 to be responsible for the misbehavior of statement s2 of target process 250 by defining accessibility between s1 and s2 as a necessary condition for existence of cause-effect relationship between those two elements. Alternatively, the analyzer is capable of evaluating the potential for statement s2 to be effected by the modification of statement s1.

The accessibility between statements that comprise the SUM-Graph may be defined with respect to three potentials: 1) the free fall potential within a main branch; 2) the shift potential of a logic condition; and 3) the rewind potential of a terminal.

REPLACEMENT SHEET

25 FEB 1998

- 22 -

Free fall potential is determined by one process element becoming active after another unless the direction of execution is changed by a logic condition in a high potential state. Quite simply and with reference to FIG. 5(A), a free fall potential exists between arithmetic statement A1 (k,b position 4,1) and arithmetic statement A2 (k,b position 7,1) because A2 follows the logic condition if target process 250 fell through the logic condition L2 (k,b position 6,1) in its low potential state until terminal *1 (k,b position 10,1) is reached.

The shift potential is the potential of a logic condition to increment the SUM-Graph b coordinate as a result of a positive solution of the logic condition. With reference to FIG. 5(A), logic conditions L3 (k,b position 8,1), L4 (k,b position 11,2), L5 (k,b position 15,3) and L6 (k,b position 8,5) all have a shift potential of 1 because a positive solution to any of them results in a right shift by one of the b coordinate. Similarly, logic condition L2 (k,b position 6,1) has a shift potential of 4 and logic condition L1 (k,b position 2,1) has a shift potential of 6.

In order to appreciate the significance of the SUM-Graph structure and the information which the analyzer derives from its inventive structure, some additional definitions are required to facilitate understanding.

Definition G6: *bn will refer to the terminal with b-coordinate equal n. With reference to FIG. 5 *b3 refers to the GO TO terminal on the b=3 and *b5 refers to "exit" terminal on b=5.

Definition G7: Mbn will refer to a Main Branch built on the b-coordinate equal n. By way of example and with reference to FIG. 5, MB3 refers to the Main Branch built on coordinate b=3.

Definition G8: The shift potential of LC is the potential of LC to increment b-coordinate as a result of a positive solution of the LC.

Definition G9: The rewind potential of a terminal is the potential of that terminal change the next active Segment Address (SA).

Definition G10: The segment address (SA) is defined as a pair (BA,k), where BA is the binary address of the segment and k is a k coordinate within the segment. For example with reference to FIG. 5(A), the segment address of SUM-Object set member /3 is 10010,13. Further, segment addresses can be compared by the present invention in a manner similar to the comparison of binary addresses.

25 FEB 1998

- 23 -

To compare binary addresses, the software analyzer left justifies the binary addresses, then corresponding bit position contents are compared from left to right until a mismatch occurs. When comparing binary address bit position contents, 1 > 0 > empty. By examining FIG. 5(A), 5(B) and Table 3, the comparison of binary addresses becomes apparent. The binary address of logic condition L2 (k,b coordinate 6,1 ; binary address 10) is greater than the binary address of logic condition L1 (k,b coordinate 2,1; binary address 1). The binary address of E1 (k,b coordinate 3,7; binary address 11) is greater than the binary address of L2 (binary address 10).

To compare Segment Addresses (SA), the binary address (BA) component of the SA is given greater priority, i.e., a segment address of 11,7 is greater than a segment address of 10,10 and segment address 11,7 is greater than a segment address of 11,6. Therefore, the segment address of L5 (FIG. 5) is greater than the segment address of A5 because their binary addresses are equal and the k value of L5 is greater than the k value of A5.

Definition G11: The vision of an element is the set of all elements which lie within any t-pass which passes through that element. An element s2 is within the vision of element s1 (or element s2 is seen from element s1) if the segment address (SA) of s2 is a descendant of the SA of s1. A statement s2 is within the vision of statement s1 if there is a path from s1 to s2 that is part of a t-pass.

Definition G12: A segment address of a statement is a descendant of the segment address of another statement if the binary address of the second statement is built from the binary address of the first statement by concatenating binary digits at the right, or if the BA(s1) is equal to the BA(s2) and the k coordinate of s2 is greater than the k coordinate of s1.

With reference to FIG. 4., A6 424 is within the vision of A3 412 since the segment address of 424 (100110) is a descendant of the segment address of 412 (1001). By way of an additional example, the binary address 1001 can be mapped onto 10011, therefore any element within segment 10011 is within the vision of an element within segment 1001. Further, element X2 418 (binary address 10010) lies within the vision of element A4 415 (binary address 10010) since they have the same binary address and the k coordinate of X2 (14) is greater than the k coordinate of A4 (12) when mapped to the SUM-Graph.

25 FEB 1998

- 24 -

Definition G13: The vision of a rewind terminal is defined by the vision of its rewind entry. The vision of the exit terminal is empty. As an example, refer to FIG. 9. The vision of rewind terminal 1 930 is the vision of its rewind entry 912. The vision of exit terminal 960 is empty.

Definition G15: The access field of a process element is the combination of all the visions that the element successively owns. When an element sees a terminal, a vision of that terminal, which is the vision of its rewind entry may see some terminals which were not originally seen. And so on...therefore expanding the access field of the examined process element.

Definition G16: The rewind binary address (RBA) is the binary address of the rewind terminal entry. The rewind binary address of the exit terminal is 0. Referring to FIG. 5(A), the RBA of terminal *1 (k,b coordinate 10,1) is defined by its terminal entry /1 (k,b coordinate 5,1). Therefore the RBA of terminal *1 is the binary address of /1, which is 10. The position of rewind terminal entry is identified by a pair {BA,rk}, where rk is the k coordinate value of the rewind terminal entry. The rk value of the exit terminal is zero.

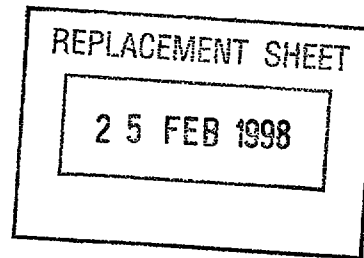
Definition G19: An open terminal is an exit terminal or rewind terminal that has access to an exit terminal.

Definition G18: A closed terminal is a rewind terminal that does not have access to an exit terminal.

Definition G17: The exit access indicator (EAI) is an indicator which is set to 0 for closed terminals and set to 1 for open terminals.

Definition G20: A closed region is a combination of adjacent main branches which are identified by closed terminals. Once a target process 250 enters a close region, that process cannot terminate. Referring to FIG. 9, terminals 944, 952 and 950 identify a closed region.

Definition G21: The propagated vision point (PVP) of a process terminal is defined by the vision of all the terminals that the process terminal sees. As previously defined, the vision of a terminal is the vision of its rewind entry. If the rewind entry of a first process terminal can see a second terminal whose rewind entry can see the rewind entry of the first process terminal, then the vision of the second rewind entry becomes the propagated vision point for both the first and second terminal.



- 25 -

By way of example and with reference to FIG. 10, if rewind entry /n 1001 of process terminal *n 1004 can see another terminal *i 1002 whose rewind entry /i 1000 can see the rewind entry /n 1001, then the vision of *n 1004 is propagated by the vision of *i 1002 and the vision of *i 1002 becomes the propagated vision point of both terminals *i 1002 and *n 1004. Therefore the propagated vision point of terminals *i 1002 and *n 1004, is the rewind entry of /i 1000.

Further, the terminal *k 1005 has a rewind entry /k 1003 which sees another terminal *n 1004. The propagated vision point of terminal *k 1005 is therefore propagated to the vision of terminal *n 1004, that had been already established as propagated to /i 1000.

The propagated vision point of a terminal is represented by a segment address of a specific rewind entry. If a terminal vision is not propagated by the visions of other terminals, then the propagated vision point of that terminal is its own vision, i.e., the vision of its rewind entry.

When all of the terminals within the vision of a rewind entry have that entry as their propagated vision point, the result is a closed region (Data Independent Endless Loop). This is because none of the terminals within the vision of that rewind entry have had an access to an exit terminal since the vision of an exit terminal would have been zero.

Once the SUM-Graph 262 is constructed, the exit access indicator (EAI) table is constructed. FIG. 9 shows a sample EAI table constructed from the terminal binary address, the rewind binary address and rk of each process terminal, all of which are known once the SUM-Graph is constructed.

FIG. 9 shows an EAI table which is constructed according to the following method. First, the original value of the propagated vision point of a terminal is built from that terminal rewind entry, that is, the rk and RBA of the terminal. Second, the EAI of all rewind terminals are set to 0 (set closed) and the EAI of all exit terminals are set to 1 (set open). Lastly, and proceeding from $b = 1$ to the maximum b of the SUM-Graph, each terminal PVP is compared with the Binary Address and PVP of every other terminal in an attempt to propagate (decrement) the PVP of the current terminal under examination and attempt to open that terminal if closed. If the PVP of a closed terminal can see any open terminal, then the examined terminal is defined as open by setting its EAI to 1.

25 FEB 1998

- 26 -

By way of further explanation and with reference to the EAI table shown in FIG. 9, the construction of the EAI table proceeds as follows:

1) Set the PVP of terminals to the segment address of their rewind entries:

BA of PVP of terminal *b1 (PVP*b1) is set to 10;
 BA of PVP*b2 is set to 1001;
 BA of PVP*b3 is set to 10010;
 BA of PVP*b4 is set to 10011;
 BA of PVP*b5 is set to 0, since *b5 is an exit terminal;
 BA of PVP*b6 is set to 10011;
 BA of PVP*b7 is set to 10;

2) Propagate the PVP and Set the EAI. If the PVP of a closed terminal can see any open terminal, then the examined terminal is defined as open by setting its EAI to 1.

a) PVP*b1 {10,rk1} sees *b2,*b3,*b4,*b5, but neither of those terminals PVPs can see PVP*b1. Therefore
 PVP*b1 is not changed; PVP*b1 {10,rk1} sees open
 *b5 (1010);
 open *b1: set EAI*b1 to 1.

b) PVP*b2 {1001,rk2} sees *b3,*b4 whose binary addresses are 100110 and 100111, but neither of these terminals can see the PVP*b2.
 Therefore,
 PVP of *b2 is not changed;

PVP*b2 {1001,rk2} does not see any open terminals;
 EAI*b2 is kept 0;

c) PVP*b3 {10010,rk3} sees *b2 (10010) and PVP*b2 {1001,rk2} sees PVP*b3, that is, {10010, rk3}. Therefore,
 set PVP*b3 to PVP*b2: {1001,rk2};
 propagated PVP*b3 still ~~does~~ not see any opened

terminals;

EAI*b3 is kept 0;

25 FEB 1998

- 27 -

d) PVP*b4 {10011,rk4} sees *b3 (100110), and PVP*b3 {1001,rk2} can see PVP*b4 {10011,rk4};
set PVP*b4 to {1001,rk2};
propagated PVP*b4 still does not see any open terminals;
EAI*b4 is kept 0;

e) PVP*b5 is not changed, since it is an exit terminal;

f) PVP*b6 {10011,rk6} sees *b3 and *b4, whose current PVPs {1001,rk2} can see PVP*b6;
set PVP*b6 to {1001,rk2};
propagated PVP*b6 still does not see any opened terminals;
EAI*b6 is kept 0;

g) PVP*b7 {10,rk7} can see terminals from *b1 to *b6, but neither of them can see PVP*b7;

PVP*b7 is not changed;
PVP*b7 {10,rk7} can see open terminals (1000 and 1010);
open *b7: set EAI*b7 to 1;

In some target processes and with reference to FIG. 10(B), terminal *i 1006, that is propagated by the vision of *n 1007, is examined before the PVP of *n 1007 is finally set, since the PVP of *n 1007 may in turn be upgraded by the vision of the other terminals. Therefore, the only way to make sure, that all the possible upgrades of PVP and EAI were performed, is to repeat the step (2) of examining all the terminals in the order from *b1 to *bn once again. If no upgrades were done within the last pass, the process of upgrading the PVPs and EAIs is finished.

In the worst case, the number of required passes through the terminals, i.e. executions of step (2) is equal to the number of Main Branches, NB, since each pass will finally set the PVP and the EAI of at least one terminal. In the above example all upgrades were performed in one pass, and the second pass through *b1 to *b7 will not change any PVP or EAI.

Once the exit access indicator table is constructed, Propagated Vision Points built into this table allow one to define the accessibility between any two elements of the graph. As a consequence, all data independent endless loop constructs are located, as those constructs do not have accessibility to an exit terminal.

REPLACEMENT SHEET

25 FEB 1998

- 28 -

It is commonly accepted that the question of whether or not a program will terminate is proven to be unsolvable. We can divide this question of process "terminability", i.e. ability to terminate, into two categories: 1) data independent endless loops and 2) data dependent endless loops.

A data independent endless loop consists of process elements that do not have an access to "exit" terminals. A data dependent endless loop is such a construction that after entering it the control will never reach an exit (STOP or RETURN) by the reason other than the absence of control passes to these exit terminals. The reason in this case is either: a) the absence of the manipulation of program variables, that could potentially change the control pass, on the specific control passes, or b) change of these program variables in the direction that takes the values of LCs further from changing their states between Low Potential State and High Potential State.

In both cases a and b above, the control will fall into an endless loop, going through the same terminals in the same sequence, without ever being able to reach an "exit" terminal.

As an example of data dependent endless loop and with reference to FIG. 9, if there are no elements on the pass between /1 912 and *1 930 that can redefine the values, being examined by L2 914 and L3 924, then terminal *b1 930 will produce a data dependent endless loop as soon as L2 914 and L3 924 will be resolved by their Low Potential solutions the first time (since the state of L2 914 and L3 924 will never change during the forward execution of the target process.)

As an example of a data independent endless loop and with reference to FIG. 9, a data independent endless loop construction is presented by Main Branches MB2, MB3, MB4. If L3 924 is ever solved positively, the process will go in the infinite loop, no matter, what data manipulation statements are coded within those branches.

While process constructions that could allow for data dependent endless loops present a potential for process "non-terminability", they are not necessarily faulty constructions, since it could be so, that an input data can never be such, that would set LCs in the state of infinite loop.

REPLACEMENT SHEET

25 FEB 1998

- 29 -

Entry /q 922 in FIG. 9 does not correspond to any terminal and is shown only as an example of the fact, that the position of L3 924 in the negative subfield of L2 914 with rewind terminal *b1 rewinding the process control directly in front of L2 914 is not necessarily a fault, even if L2 914 and L3 924 are not dependent on the data manipulations that are coded on the path /1 912 to *1 930. This is, because that if L3 924 had received control through the entry /q 922, returning the control to the entry /1 912 will not necessarily result in L2 914 being solved negatively thus in creating an endless loop.

On the other hand the programs that contain Data Independent Endless Loops constructs are definite faults in the target process construction.

The detection of data independent endless loops is done by evaluating an accessibility between rewind terminals and exit terminals as it was explained on the previous example. The same process would allow evaluation of accessibility between any two elements of SUM-Graph (for example between A2 and A7 in FIG. 9).

As a result we see: EAI's of terminals *b2, *b3, *b4 and *b6 are 0, these terminals are closed. Terminals *b2, *b3, *b4 constitute a closed region (data independent endless loop). Terminal *b6 is also closed and its rewind entry is within the mentioned closed region [*b2 - *b4]. Dotted line 966 represents the vision field of an entry /1 {10,rk1}. This entry is PVP of *b1. This means, that any element of Main Branch b=1 can access any element within the vision of {10,rk1}.

An access field of A2 is represented by the dotted line 966. Dotted line 967 represents the vision field of an entry /2 {1001,rk2}. This entry is PVP of *b2, *b3, *b4 and *b6. This means, that any element of Main Branch MB2, MB3, MB4 and MB6 can access any element within the vision of {1001,rk2}.

By definition, an access field of a process element is the combination of all the visions it successively owns. Therefore, an access field of A7 is combined by the fields represented by dotted line 964 and dotted line 967, as dotted line 964 is the vision of A7 and dotted line 967 is the vision of *b6 that is seen from A7.

From FIG. 9 we can see, that A2 can access A7, and A7 cannot access A2.

25 FEB 1998

- 30 -

As an example from FIG. 9, examining the attributes (BA, RBA and PVP) that are built for every terminal, we can determine the following: a) A2 can not see A7, as 101 is not seen from 100. However, A2 can access A7, since one of the terminals, seen from A2, *b1, can see A7 through its PVP {10,rk1}, as 101 is a descendent of 10; b) A7 can not see A2, as 100 is not a descendent of 101; the terminals that are seen from A7 are ones with BAs 1010 and 1011; Terminal 1010 has no RBA or PVP, being an exit terminal; terminal 1011 has PVP {1001,rk1}, which can not see 100 either. Therefore A7 can not access A2.

Data dependent endless loops could be determined using our method of propagated vision points in the following manner. Once the sequence of terminals executed repeats many times it could become useful to locate the propagated vision point of those terminals and then to see whether or not there are any data manipulation statements on the t-passes from this PVP to those terminals which data manipulation statements affect the variables checked by the logic conditions located on the t-passes from this PVP to those terminals. If no such statements exist, we have a data dependent endless loop. If such statements exist but are constructed in a way such that the variables change in the direction that would take the value calculated by the logic condition further from the value that would change the status of the logic condition. In this case we also have a data dependent endless loop.

Determining Process Correctness

As a result of the structures which are constructed by the SUM-Analyzer it is possible to evaluate the correctness of a target process. In order for those skilled in the art to readily appreciate how the SUM-Analyzer operates, the following definitions are necessary at this point.

Definition P1: The symbol (s), represents a process element or individual statement of a program.

Definition P2: The Process Address, (PA) {k,b} of an element is the position of the element within the reduced flow chart of the process or the position of the element within SUM-Graph of that process. With reference to FIG. 5, the PA of an element is defined by the k,b coordinate pair which describes the position of the element within the reduced flow chart. With reference to FIG. 5, element A2 has a PA of 7,1. Each statement that comprises a process has a unique PA.

REPLACEMENT SHEET

25 FEB 1998

- 31 -

Definition P3: The local time $\{p\}$ parameter is an integer value which represents the present occurrence of a PA within a process execution. Local time is started with the initial invocation of the process. The local time can only be measured at a PA. The SUM-Analyzer does not utilize time as it is normally used, i.e., a measurement of duration. Instead, time as recorded by the SUM-Analyzer is a measurement of the number of occurrences of a particular statement. Therefore each event or occurrence of a statement execution is identified by its own time parameter.

Definition P4: A process event is an event of a process statement execution. A process event is defined by $\{k,b,p\}$, where k,b are the coordinate pair which identify the process address and p is the local time within this process address.

Definition P5: The Process time is represented by process events $\{k,b,p\}$. As an example and with reference to FIG. 5(A), when statement A5 is executed the third time, that moment of execution is referred to as the $\{13,3,3\}$ event.

Definition P6: A process event $\{k,b,p\}$, other than an output event, is correct if that event takes part in a chain of events that effect a correct output event. An output event that is an event of execution of system output statement is correct if it produces expected output result.

Definition P7: A Correct Process Address (CPA) contains a process element (s) that had been executed as a correct event.

Definition P8: An Uncertain Process Address (UPA) contains a process element (s) that has not yet been executed as a correct event.

A state of PA as UPA or CPA, is relative to some other time in the process. This state could be changed at some process time $\{ke,be,pe\}$ which is an event of execution of some other process address $\{ke,be\}$ which event could define (s) event execution as correct or could make it uncertain.

As an example, consider a process defined by the chain of events $e1, e2, e3, e4, e5, e6$ and $e7$ where $e7$ is a correct output event, $e1$ and $e3$ are input events and $e1, e2, e3, e4$ and $e6$ were involved in producing the correct output event $e7$. Therefore, events $e1$ through $e7$ except $e5$ are correct. Event $e5$ may be defined as correct by some other correct output event chain in which $e5$ took place, but until such a time its correctness is uncertain.

25 FEB 1998

- 32 -

Definition P9: The algorithm space of a process is the space defined by the combination of all process addresses $\{k_i, b_i\}$. With reference to FIG. 5, the algorithm space of the target process described by the reduced flow chart is bounded by the maximum k and maximum b shown.

Definition P10: The correct algorithm space of a process is the space of all process addresses which are presently defined as correct process addresses.

Definition P11: The uncertain algorithm space of a process is the space of all process addresses which are presently defined as uncertain process addresses.

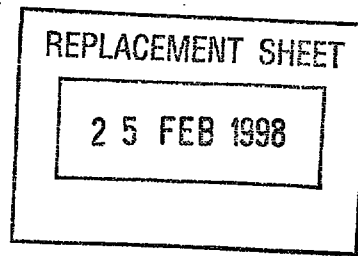
In both of the above definitions the correctness or uncertainty of a space is relative to the present time within the process. This is due to the fact that the SUM-Analyzer has the ability to later redefine a correct event as uncertain (losing the knowledge). The process address may become uncertain if it does not any longer have a history of being executed as a correct event.

Definition P12: The event definition space of an event (e) consists of process addresses of the event's ancestor events.

Definition P13: An ancestor event is a prior event which took part in producing a later event by either: 1) producing a value of the later event (value ancestor event) or 2) producing a parameter for a conditional control event, i.e. an event of executing a conditional control statement (L or D of the SUM-Object Set) in a path leading to the later event (control ancestor event). If either of these conditions are present, then the later event is a descendent of the prior event.

By way of example and with reference to FIG. 14, $We(a)$ 1400 is a descendent of the events shown below $We(a)$ and within its branches. Further, $e(d)$ is a descendent of $e(i)$ 1410, $\#e(j)$ 1413, $e(k)$ 1414, $*e(l)$ 1415, $*e(m)$ 1418 and the other events that are the ancestors of $\#e(j)$ 1413, which are $\#e$ 1417, $\#e$ 1419, $\#e$ 1420, $\#e$ 1421.

The SUM-Analyzer operates by preserving the event definition space of correct events which, in turn, preserves the elementary knowledge of process element execution, i.e., the correctness or uncertainty of this event. The SUM-Analyzer continuously evaluates the event definition space of new output events and expands the correct algorithm space of a target process 250, when possible. On the other hand, when evaluating the event definition space of



- 33 -

output events found to be incorrect, it may lose previously built knowledge of correct algorithm space by losing the status of some CPA to UPA.

The relationship between A,L,D,E,X,*,+,C,!,/,R,=,S and W elements of the SUM-Object Set are such, that an approved event of a W element execution activates a chain of secondary processes. These secondary processes thereby approve the execution events of those SUM-Graph elements that were involved in producing the approved event.

As a consequence of the principles we are using, the fault(s) responsible for the failure of an output result will be located within that part of the output result event definition space that lies within the area of uncertain algorithm space. Likewise, modifications made to a program as a result of new system requirements must be located within that part of the output result event definition space that lies within the area of uncertain algorithm space. If there is no uncertainty within the event definition space of the results that must be modified, then the new branches should be built in order to preserve correct algorithm space.

Definition K1: The approval of an event is the process of validating that events correctness by the SUM-Analytical Engine.

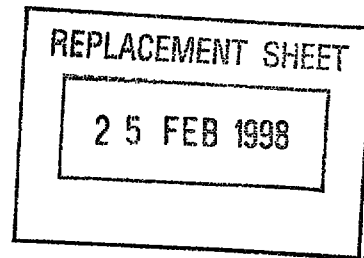
Definition K2: A control event is any event which may potentially change the direction of a process from falling down the main branch.

Definition K3: An unconditional control event is an execution of rewind or exit terminals as defined in the SUM-Object set by {*,+,C,R,=}.

Definition K4: A conditional control event is an execution of a logic condition (L or D elements of the SUM-Object Set).

A positive solution of a conditional control event results in a control shift of a process to the right as viewed within an SUM-Graph or reduced flow chart. A negative solution to a conditional control event results in the direction of a process to proceed downward as viewed within an SUM-Graph or reduced flow chart.

Definition K5: A data manipulation event is an execution of an (A) or (E) or the arithmetic part of a (D) element of the SUM-Object Set.



- 34 -

Definition K6: An initial data manipulation event" is one of the following: E-event, A-event of data initialization or D-event of the loop index initialization.

Definition K7: A Secondary Data Manipulation event is any data manipulation event other than an initial data manipulation event.

Definition K8: An elementary knowledge of a process element located within $PA\{k,b\}$ is obtained, when one of its execution events is approved. Accumulating knowledge about a process is performed by accumulating information about the correctness or uncertainty of the process element's execution events.

Definition K9: Elementary knowledge propagation is a process by which a elementary knowledge of an event is applied to different process addresses. Elementary knowledge propagation is performed by locating and analyzing all of the ancestor events up to a knowledge propagation terminal events.

Definition K10: A knowledge propagation terminal (KP terminal) is defined as an event that either has no ancestors within the target process (i.e., an initial data manipulation event) or is a previously approved event.

Terminating knowledge propagation at previously approved events stabilizes the process of knowledge propagation and prevents the knowledge propagation process from becoming increasingly complex with the accumulation of ancestor events during the forward execution of a target process. When a knowledge propagation process encounters a previously approved event, all of the ancestor events of the approved event were previously approved as well.

FIG. 14 shows events $We(a)$ 1400 and $We(n)$ 1401 above a tree of their ancestors. The process direction is indicated within FIG. 14 by arrow 1422, therefore event $We(a)$ 1400 occurs at some time before event $We(n)$ 1401. In this example, $*e$ represents a KP terminal event, that is an initial data manipulation event and $\#*e$ represents a KP terminal that is a previously approved event.

In FIG. 14, the value of $We(a)$ 1400 was calculated from three parameters whose three values were produced by the three events $e(b)$ 1402, $\#*e(c)$ 1403 and $e(d)$ 1404, respectively. Further, a value associated with $e(b)$ 1402 was calculated from two parameters whose two values were produced by the two events $*e(f)$ 1407 and $\#*e(g)$ 1408 respectively. Event $*e(f)$ 1407 does

REPLACEMENT SHEET

25 FEB 1998

- 35 -

not have any ancestors, since it is an initial data manipulation event and is therefore a KP terminal. Event $\#*e(g)$ 1408 was previously approved (as indicated by #) and therefore it is also a KP terminal.

There is no attempt by the SUM-Analyzer to locate the ancestors of a KP terminal. This is because such ancestors either: 1) are not present (as is the case with $*e(f)$ 1407) or 2) have been previously approved ($\#*e(g)$ 1408).

With reference to FIG. 17, event $e(0)$ 1700 is shown as occurring at some process time $(k0, b0, p0)$. Further, if event $e(0)$ 1700 were the first approved event of $s0$ 1701, then the PA of $s0$ 1701 is upgraded from UPA to CPA as indicated by arrow 1710. This example shown in FIG. 17 indicates that a value of variable y is a function of three other parameters, $x1$ 1702, $x2$ 1703 and $x3$ 1704.

The immediate ancestors of $s0$ 1701 are the events which produced the value of the parameters involved in the $s0$ 1701 calculation. The value of the parameters of $s0$ 1701, namely $x1$ 1702, $x2$ 1703 and $x3$ 1704 are determined by ancestor events $\#*e1$ 1705, $\#*e2$ 1706 and $e3$ 1707. If events $\#*e1$ 1705 and $\#*e2$ 1706 were previously approved as shown, they are treated as KP terminals and we do not examine their ancestors.

Event $e3$ 1707, which calculates parameter $x3$ 1704 was not previously approved. The CPA status 1708 which was a state of the process address of statement $s3$ 1709 during the process time $(k0, b0, p0)$, means that some other event(s) of this PA execution was previously approved. Therefore, it is necessary to examine all of the ancestors of the event $e3(k3, b3, p3)$ 1707 until a KP terminals are met during such examination on each direction.

Definition K11: The immediate owner of an event is another event which approved the event.

Definition K12: A W-owner of an event is a descendent of the event which originated the chain of elementary knowledge propagation that approved the event.

Each event has only one W-owner event and only one Immediate Owner event. In some situations, the W-owner of an event and the immediate owner of an event are one and the same.

REPLACEMENT SHEET

25 FEB 1998

- 36 -

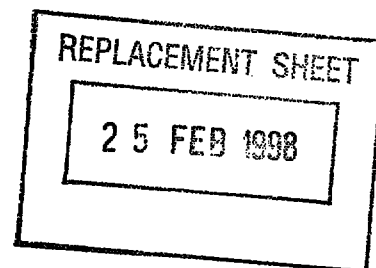
As an example of event approval and with reference to FIG. 16, a process address (k,b) 1600 is shown during five different process times t1 1605, t2 1606, t3 1607, t4 1608 and t5 1609, respectively. These process times are defined by the execution of events at some address or addresses that could be other than (k,b). At time t1 1605, the PA 1600 had no occurrences of its execution. At time t2 1606, there has been two occurrences of PA 1600, e1 1610 and e2 1611, neither of which has been approved. At time t3 1607, there has been three occurrences of PA 1600, e1 1610, #e2 1611 and e3 1612. Further, event #e2 1611 has been approved during the interval which occurred between t2 1606 and t3 1607. The status of Process address 1600 therefore, is known to be CPA at time t3 1607. The W-owner attribute of CPA helps to maintain the current knowledge of the target process 250. When a previously approved W-event is disapproved, the target process knowledge is dropped to the level determined by the position of the disapproved W-owner event in the target process time.

The process of approving an event generates a secondary process which approves the immediate ancestors of the event unless the immediate ancestors are already approved or do not exist. Therefore, the propagation of elementary knowledge of the process continues in the direction of all ancestors until a knowledge propagation terminal encountered in each direction.

Building the SUM-Model of the Target Process

After the SUM-Builder 259 finishes construction of the SUM-Graph previously described, an SUM-Model 243 of target process 250 is built. The invention constructs a different SUM-Model for each distinct software process analyzed where the process is procedure i.e., program, subroutine, function, paragraph. Although the preferred embodiment of SUM-Model 243 discussed in this detailed description is a three dimensional model, those skilled in the art could readily adapt the model of the present invention to additional dimensions. Two of the three dimensions on which the SUM-Model is constructed have already been described, namely the k,b coordinates of the SUM-Graph 262.

The third dimension of the constructed SUM-Model comprises a structure, that is populated with attributes of process addresses. The process address of an element is the position (k,b) of that element within the reduced flow chart, or alternatively within the SUM-Graph 262 of the target process 250. Table 4 is an example of the implementation of the structure constituting the third dimension of SUM-Model for the elements of the SUM-Object set *, C or +.



- 37 -

Table 4
Attributes

* C + ind	re-wind ID	loc.time	BA	tar.rec#	rep.rec#
SUM-Srec#	RBA				

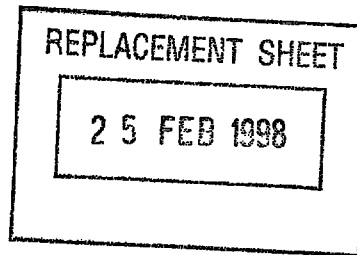
Different attributes are assigned to different process addresses, depending upon the type of process element (i.e., A,L,D,W,E) that is positioned at this process address. Referring to Table 4, one member of the structure is populated with the type of process element. Another member of the structure contains ind, which is the index of the corresponding process element in the SUM-Frame and SUM-Graph.

Each element of the SUM-Model corresponds to an executable statement and contains three distinct attributes which establish a relationship between the target process 250, the SUM-Model 243, the SUM-Repository 246 and SUM-Source 247. Those attributes providing this relationship are: 1) a record number of the corresponding statement within the target process; 2) a record number of the corresponding statement within the SUM-Repository; and 3) a record number of the corresponding statement within the SUM-Source. Referring once again to Table 5, these three attributes are present in the structure members.

For each executable process element, the SUM-Model keeps an attribute of local time. Local time is an integer value of the occurrence of a statement within a target process. For example, the first time that a statement is executed during a program execution, the local time would be 1. The second time that the same statement is executed during a program execution, the local time would be defined as 2. The local time attribute is present as a structure member of the sample SUM-Model structure shown in table 4.

The binary address attribute is assigned to each process address of the SUM-Model. In Table 4, the binary address is shown as a structure member.

Structures representing rewind terminals *,+,C also possess an attributes of Rewind ID and Rewind Binary Address that represent the Process Address and BA of the position in the SUM-Graph, that will be active after that statement execution.



- 38 -

The ID value is an alternative representation of (k,b) coordinate pair. Each process element represented by a (k,b) coordinate pair in the SUM-Graph could be alternatively represented by one integer value ID, where the integer is the numerical order of the element as it is encountered during a traversal of the two dimensional table on which the SUM-Graph is built in order of traversing columns, left to right. The ID of any particular element with process coordinates {k,b} could be defined by the following formula:

$$ID = ((b-1) * k_{max}) + k$$

where k_{max} is the maximum value of the k dimension represented on the SUM-Graph.

From the other side knowing the ID of element s {k,b} and knowing the k_{max} of the model, one could define {k,b} by the following formulas:

$$b = (ID - 1) / k_{max} + 1$$
$$k = ID - (b-1) * k_{max}$$

Referring to SUM-Graph shown in FIG. 5(A), graph element *1 (k,b coordinate 10,1) has an ID of 10. Element L6 (k,b coordinate 8,5) has an ID of 76.

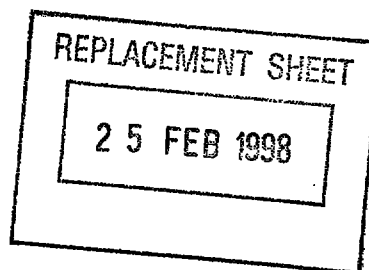
Table 5 shows the representative SUM-Model structure for a process element represented by the following SUM-Object set members, R and =

Table 5
Attributes

R =	ind	loc.time	BA	tar.rec#	eXS.rec#
-----	-----	----------	----	----------	----------

Exit terminals, when represented in the SUM-Model, and in Table 5 do not contain the following attributes of rewind terminals; Rep.rec#. Rewind ID and Rewind BA.

Process addresses containing process elements represented by the SUM-Objects A,L,D,E,W,X and S each have an ACB Pointer attribute. The ACB Pointer attribute is a pointer to a corresponding Analyzed Calculation Base construction that contains information on the history of approved and uncertain execution events of the current process elements in relation to the knowledge of



their effects correctness.

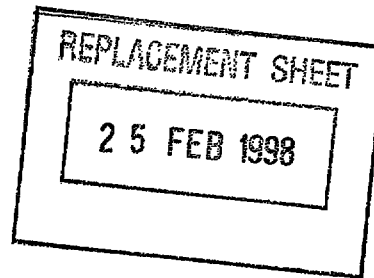
In addition, structures representing L and D elements have an additional attribute which represents the increment in the b coordinate when traversed through their positive, or high potential solution. This corresponds to the shift potential previously described with respect to the SUM-Graph. For example, element L1 (k,b position 2,1) shown in FIG. 5(A) would have a shift potential of 6 recorded in its shift potential attribute field.

The Analyzed Calculation Base (ACB) 273 is a collection of structures, each corresponding to a different elements (statements) of the target process 250. The information recorded within the ACB indicates the knowledge of correctness or its uncertainty of the effects of the different process address execution events at different times within the process.

FIG. 19 shows a number of ACB structures with each ACB construction corresponding to an executable process address of the target process 250. Every member or element of an individual ACB structure represents an event of a target process element execution.

In a present embodiment, the ACB pointer element of the SUM-Model previously described points to a binary file that is divided into two parts. The first part of the ACB structure is a control area that contains two distinct fields; Next Parameter Word (NPW) and Uncertainty Displacement Word (UDW). The NPW is only used for elements that have parameters, i.e., E and X. Referring once again to FIG. 19, an ACB structure for element A1 is shown as 1900. This ACB structure contains three fields, 1901, 1902 and 1903. The NPW is shown as 1901 and the UDW is shown as 1902.

The second part of the ACB structure, shown in FIG. 19 as 1903, is the information area which stores knowledge state bits. Each execution event of the element that is represented by this ACB structure is represented by a bit in the information area 1903. The position of a particular bit within the information area corresponds to the local time of the event. That is to say, the 20th bit of the information area 1903 corresponds to a local time of 20, or the 20th time that particular element was executed. The state of the binary digits (0 or 1) comprising the information area correspond to the current knowledge of the events effects correctness, where 1 represents a correct effect of an event and zero corresponds to an uncertain effect of an event. When an event is represented by zero in the ACB structure existing at some process time, it means that at that process time, the effect of the examined event is not known to be correct.



- 40 -

Those skilled in the art can readily appreciate that the monitoring of a target process 250 can generate enormous volumes of data even when recorded in a structure as simple as the ACB. While many forms of compression could be used, the present implementation UDW 1902 stores the relative displacement of the first bit that is in a zero state. For example, UDW 1902 contains the value 2000. This represents that the first bit position stored in information area 1903 is for a local time value of 2001 and that all 2000 previous events of that element execution were approved or disregarded as a blank events, i.e., events which did not produce any results which were used subsequently.

Each parameter of an input statement (E) is represented by a ACB structure. Referring once again to FIG. 19, two parameters of an input statement are shown represented by two ACB structures 1905, and 1906. The NPW of 1905 points to structure 1906.

Similarly, each parameter of a call statement (X) is represented by an ACB structure. FIG. 19 shows three ACB structures 1907, 1908 and 1909 which correspond to three parameters of the call statement. The NPW of 1907 points to structure 1908. The NPW of 1908 points to 1909. The NPW of 1909 is empty since there is no next parameter in this CALL() statement.

Building the SUM-Repository

The SUM-Repository 246 of target process 250 is built at the same time as the SUM-Graph 262 or the SUM-Frame 261. Each element of the main branch of the target process that corresponds to an executable statement, i.e., represented by SUM-Object set members A, L, D, E, W, X and S, is stored in the SUM-Repository 246.

The size of the SUM-Repository 246 is less than the corresponding target process source code. This is because the SUM-Repository 246 does not contain any process control statements, therefore the SUM-Repository does not contain any unconditional branches, stop, return, then, else, endif, etc. The control structures are instead, preserved by the SUM-Builder 259 in the SUM-Model 243. To demonstrate how particular statements of target process 250 are represented in the SUM-Repository, consider the following examples.

If target process source 250 contained the statement:
if (d < 10) go to label

REPLACEMENT SHEET

25 FEB 1998

- 41 -

The SUM-Repository 246 would contain the statement:

```
if (d < 10)
```

Alternatively, if target process source 250 contained the following statements:

```
if (a .gt. b) then
    c = d
else
    e = f(g)
endif
return
```

The SUM-Repository 246 would contain:

```
if (a .gt. b)
c = d
e = f(g)
```

Each statement of the SUM-Repository is referenced by the repository record number (rep rec.#) attribute of the corresponding process address within the SUM-Model. For example if the SUM-Repository is kept as a Relative Record Data Set (RRDS) then the repository record attribute of the process address within the SUM-Model must contain the relative record number of the corresponding statement within SUM-Repository 246.

Building the SUM-Source Code

Subsequent to the construction of the SUM-Graph 262, the SUM-Model 243 and the SUM-Repository 246, SUM-Source 247 is generated by the SUM-Source generator 249. This step is only done if the SUM-Analyzer is implemented in an embodiment which uses traditional object code.

The SUM-Source 247 provides a synchronization mechanism between the target process 250 and the SUM-Analytical Engine 244. This synchronization mechanism is implemented through the use of synchronization signals, which are inserted within the SUM-Source 247, by the SUM-Source Generator 249. In addition to the inserted synchronization signals, the SUM-Source 247 may differ from the target process 250 in that the SUM-Source 247 will not contain any unreachable or "dead" code.

REPLACEMENT SHEET

25 FEB 1998

- 42 -

The SUM-Source code 247 is built by the SUM-Source Generator 249 from two predecessor structures, the SUM-Repository 246 and the SUM-Model 243. Since the SUM-Source is built from the SUM-Model and the SUM-Repository which were built from the target process 250, there exists a functional equivalence between the target process 250, the SUM-Model 243 and the SUM-Source 247.

During the creation of the SUM-Source 247 by SUM-Source generator 249, the following steps are performed by the SUM-Source generator: 1) the contents of each executable statement which comprises the SUM-Source is taken from the SUM-Repository 246; 2) the control statements contained within the target process 250 which are now represented within the SUM-Model 243 are extracted from the SUM-Model; and 3) the target process 250 - SUM-Model 243 synchronization signals are inserted into the SUM-Source 247.

Those skilled in the art can readily appreciate that the synchronization signals can be implemented through a variety of means. The synchronization signals inserted into the SUM-Source may consist of CALL() or WRITE() statements, which should be immediately familiar to those skilled in the art. The inserted synchronization signals provides the SUM-Interface 263 with three parameters. The three parameters are defined as (k,b,m) where k,b are the k,b coordinates that define the process address, and m defines the model of the target process.

Alternatively, this SUM-Interface 260 could be implemented with only two parameters. For example, instead of the (k,b,m) parameters one could pass only (ID,m) where ID is the unique identification of the element's process address which is calculated from the k,b coordinate as previously described.

The synchronization signal is inserted into the SUM-Source 247 by the SUM-Source Generator 249 during the construction of the SUM-Source for each statement representing a process terminal. As an example, consider the reduced flow chart shown in FIG. 4 which represents target process 250. The flow chart shows a terminal 411 at k,b coordinate 10,1 which represents a FORTRAN GOTO statement contained within target process 250.

When the SUM-Source Generator 249 constructs the SUM-Source for terminal 411 the following program statements are inserted within the program code:

```
call SUMI(10,1,1)
GOTO 'LABEL a'
```


REPLACEMENT SHEET

25 FEB 1988

- 43 -

The call SUMI() statement is made with three parameters as identified above. By way of an additional example, consider terminal 431 shown in FIG. 4. When the SUM-Source Generator constructs SUM-Source for terminal 431, the following program statements are inserted within the program code:

```
call SUMI(9,6,1)
GOTO 'LABEL e'
```

When CPU 241 executes either of the above synchronization signals, the three parameters are passed to the SUM-Interface 263. When the SUM-Interface receives the signal, it records the process address or ID of the calling terminal and the model identifier (m) within the system execution history 253.

While the present embodiment passes the three above parameters with the synchronization signal, an alternative embodiment would pass only two parameters with the synchronization signal. Since the element ID characteristic previously described uniquely identifies a process element and the ID is derived from the k,b coordinates of the process element, the process ID could be passed as a parameter to the synchronization signal instead of the k,b coordinate pair.

When implemented in this manner, the sequence of executed process terminals is sufficient information to allow the analyzer to recreate the target process 250 behavior for any given model of a target process.

This is because, within the model, there is only one pass between the START and a specific terminal or between an entry of a Rewind ID of a Rewind ID of the previous terminal and the next terminal. The Rewind ID has been stored as an attribute of each terminal represented within the SUM-Model 243.

The terminal which sent the synchronization signal is then located by the SUM-Interface in the SUM-Model through its k,b address.

When SUM-Interface 263 working in the traditional object code embodiment of Software Understanding Machine is signaled and the parameters associated with the signal are sent, the SUM-Interface invokes SUM-Analytical Engine 244 which retraces on the SUM-Model a part of the pass executed within the target process 250. The part retraced is from the Rewind ID of the previously executed terminal (or from the START, if the currently executed terminal is the first since the process was entered) to the current terminal. Each

25 FEB 1998

- 44 -

pass through a process address represents an execution event for that process address and the local time of that address is incremented by one.

By way of example, consider the execution where the following three CALL() instructions were made to the SUM-Interface during the course of an analysis.

```
CALL SUMI(10,1,1)
CALL SUMI(10,1,1)
CALL SUMI(9,5,1)
```

With reference to FIG. 5(A), these signals correspond to the execution of terminals mb1, mb1, and mb5 respectively. From this information, the analyzer determines that the target process 250 was executed through the path identified by the following elements in this order Entry, L1, /7, A1, /1, L2, A2, L3, X1, *1, L2, A2, L3, X1, *1, L2, A7, L6, R1 (Exit).

To represent repetition of the same terminal, as is often the situation with control loops structures comprising DO, FOR, While, etc., a negative number notation is used. For example, a notation 27 -5 42 would mean that a loop was passed through 6 times (reached the terminal with IDT=27) before exiting from the loop.

As may be appreciated by those skilled in the art, the synchronization statements could have been invoked after every element or after every exit of a logic condition. Our preferred embodiment minimizes the number of signals utilized to effectively represent the target process.

These above examples demonstrate that recording the target process execution history by recording the sequence of the executed process terminals is sufficient and the absolute minimum recording required to characterize the target process 250 dynamics.

In addition to the synchronization signal previously described that synchronize the execution (retracing) of SUM-Model necessary in this embodiment with the execution of the target process, two other types of synchronization signals are inserted into the SUM-Source by the SUM-Builder. Those other two signals allow the construction of the incremented system output process synchronized map and optionally additional synchronization between the target process and SUM-Model execution processes. Those two types signals are inserted immediately following system output statements (W) and immediately before system input statements (E).

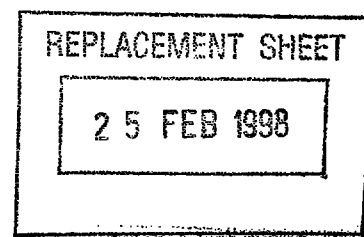
Synchronization of the system output does not invoke the SUM-Analytical Engine. Process synchronization at the system input is optional. Process synchronization at the system input, when performed, invokes the SUM-Analytical Engine and is performed in our preferred embodiment only because it is a convenient place to catch up with the target process by the SUM-Analytical Engine since the system input interrupts the target process anyway by the operator or user "think time".

With reference to FIG. 21A and FIG. 21B the entry for the Incremental System Output Process Synchronized (ISOPS) map corresponding to the system output 2151 will be performed by the call to the SUM-Interface SUMI 2152. Process synchronization at the system input 2154 is performed by 2153, process synchronization at the terminal 2161 is performed by 2160, at the terminal 2159 by 2158 and at the terminal 2157 by 2156. Entry for the System Input Process Synchronized (SIPS) map corresponding to the system input 2154 will be recorded by 2155.

Incremented System Output Process Synchronized Map

The incremented system output process synchronized map (ISOPS) provides a mechanism to associate an output of a target process with the process events which produced that output. The ISOPS map is constructed out of two elements, Incremental Image Monitor (IIM) 267 and ISOPS parameters 264 that in turn in its present embodiment consists of five fields, each containing an entry which defines a characteristic of the map:

In reference to FIG. 27 entry 1 defines a W-event as a set of $\{k,b,p,m\}$ process parameters. $\{k,b\}$ are the synchronization parameters associated with the signal sent to SUM-Interface 263. Parameter $\{p\}$ is derived from the SUM-Model 243 of the target process 250. It is the current time attribute of the process element which corresponds to the $\{k,b\}$ coordinate pair. The time attribute is obtained from the local time attribute kept by the SUM-Model incremented by one since this W event was not yet executed (retraced) in the SUM-Model in the traditional object code embodiment. In the SUM-Object code embodiment, where the SUM-Model drives the execution, the local time attribute will already be up to date and the retracing is not performed. Parameter $\{m\}$, that is also present within the synchronization signal, is SUM-Model identifier.



Entry 2 defines the screen line and column position {sl,sc} at the beginning of each system output signal. The {sl,sc} entry is {1,1} for the very first W-event entry. When SUM-Interface 263 receives a synchronization signal sent by a W-event, SUM-Interface obtains the current screen position that is in effect after the execution of the W-event. The screen position so obtained is stored and later used as {sl,sc} entry for a next W-event.

Entry 3 defines the smallest rectangular area of the system output screen 258 {l1,c1,l2,c2} which represents the area of the screen changed by the W-event execution. This rectangular area is determined by comparing the previous screen image 255 that is saved at the process time after the previous W-event, and the current system output screen buffer 265.

Entry 5 contains the contents of the sysout screen buffer 254 that corresponds to the area defined in Entry 3. Entry 5 is contained in a variable length field and is stored in the present embodiment as a separate file, the Incremental Image Monitor File 267.

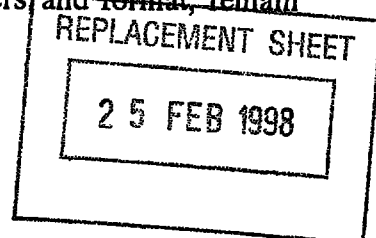
Entry 4 is an integer which defines the displacement of Entry 5 from the beginning of IIM file 267.

Building System Input Process Synchronized Map

A system input process synchronized map (SIPS) map is built by the SUM-Interface during the course of an execution of the target process to be later used for analysis of a specific input to the system made at some process time. The SIPS map is stored in system input history file 252. The entry made to this file consists of two records. The first record is the set of parameters {k,b,p,m} which describe the process time {k,b,p} and the model {m} corresponding to the current input string. The second record is the actual input string.

The SUM-Interface 263 creates the entry made to the system input history file 252 from the synchronization parameters (k,b,p,m) associated with a signal made to SUM-Interface 263 and the system input buffer .

Alternatively, from taking the system input values for the SIPS map from the system input buffer, these values could be taken from some other buffer where they were temporarily stored. They could be placed there by the statement inserted immediately after the system input statement and constructed from this input statement by simply changing READ to WRITE directed to the buffer. The remainder of the statement, i.e., its parameters and format, remain



unchanged. As shown in FIG. 21(B), the READ(*,002)CH is followed by a WRITE(M,002)CH.

FIG. 21(B) shows an example of {ID,m} process synchronization call 2153 that is made before an input from the terminal. The ID of this synchronization signal (21) is translated by SUM-Interface to k,b of 5,3 which corresponds to the process address containing the element E1 (2154 in FIG. 20(A)). If done as shown in the example, the system input is recorded by this means of statement 2155 to the input monitor buffer file 254. There could be several system input strings recorded in the system input monitor buffer by several E elements before the next terminal is met. During the next process synchronization signal performed at the process terminal (which in our example could be performed at = 2157 or C1 2159 in FIG. 20(A)), the system input string entries are taken from the system input monitor buffer 254 and recorded in the system input history file 252. After that time, the system input monitor buffer could be reused.

Alternatively, if the keyboard buffer is used to create an entry in the system input history file (ISOPS file) the presence of the system input monitor buffer is not needed as well as inserting statements 2155 (FIG. 21(B)) within the SUM-Source. Instead, the signals which would read the keyboard buffer and make an entry to the SIPS map would be needed after each system input statement 2154 in FIG. 21(B).

Operational Examples of Software Understanding Machine (SUM) Analyzer

Delayed Sequential execution of the SUM-Model is implemented by one or more CPU(s) which execute the target process and the SUM-Analytical Engine. Synchronization between the execution of the target process and the SUM-Analytical engine takes place through synchronization signals that are placed within the target process by the SUM-Source generator. When the CPU executes such a synchronization signal, program control is transferred to the SUM-Analytical Engine. The target process execution is suspended for the duration of the signal. Target process execution resumes upon return from the signal.

Delayed parallel execution of SUM-Model is implemented through the use of multiple CPUs, where one of the multiple CPUs executes the target process and another of the multiple CPUs executes the SUM-Analytical Engine at the same time. Synchronization between the multiple CPUs takes place through any one of a variety of methods known in the art. The present invention only

REPLACEMENT SHEET

25 FEB 1998

25 FEB 1998

- 48 -

requires communication of the parameters {k,b,m} between CPU(s) working on the target process and CPU(s) working on the SUM-Analytical Engine analyzing the target process.

Both, methods of execution of the SUM-Model are implemented through the use synchronization signals from process terminals {idt,m} and optionally, system input {ide,m}. The synchronization signals {idt,m} and optionally {ide,m}, bring the SUM-Model "up to date" with the target process. The synchronization signal {idw,m} is designed to bring ISOPS map "up to date" with the target process and Process Time within SUM-Model. The SIPS map is created either: 1) at the terminal synchronization points if the system input monitor buffer is used; or 2) by the synchronization signals performed after each system input statement.

When SUM-Interface 263 receives a synchronization signal that is implemented in the form {id,m}, the SUM-Interface translates the {id} parameter of the corresponding model {m} into {k,b}. The process address {k,b} will point to one of the following: a process terminal {idt}; a system output signal {idw}; or an input from the terminal {ide}.

If the synchronization signal received by SUM-Interface 263 originated from system output element {idw,m}, SUM-Interface 263 initiates the process previously described which updates ISOPS map.

If SUM-Interface 263 receives an {idt} synchronization signal, the {idt} parameters are written into system execution history file 253.

In the traditional object code embodiment, if the process synchronization signal received by SUM-interface 263 is {idt,m} or in the alternative embodiment {ide,m}, SUM-Interface 263 then initiates SUM-Analytical Engine 244. Once initiated, SUM-Analytical Engine 244 will retrace, i.e. execute, the SUM-Model 243 from the point of the model at which a previous synchronization signal {k,b} was sent or from the entry point of the SUM-Model to the {k,b} of the current synchronization signal. As each element of the SUM-Model is traced, the time attribute {p} of that element is incremented by one, thereby keeping the SUM-Model 243 time current.

When during the model execution in any of the embodiments, traditional object code embodiment, SUM-Object Code embodiment or embodiment using interpreting of the SUM-Repository, a process element is met that is a system output element (W), SUM-Analytical engine 244 initiates a knowledge induction process. Alternatively the same Knowledge Induction

25 FEB 1998

- 49 -

Process could be initiated directly from the ISOPS map synchronization signal.

The SUM-Repository entry corresponding to the current (W) element is analyzed and each parameter of the W element initiates a cycle of knowledge induction chains. Each of these chains traces back events to their ancestors, up to knowledge propagation terminals.

Tracing backwards is analogous to, but in a different direction from the earlier explained principle of tracing the target process forward. It is done from the present position of the synchronization signal within the target process SUM-Model to the previous position of the synchronization signal within the target process SUM-Model. It is possible within the present SUM-Model construction which knows the rewind entry of each terminal.

The construction of terminated event definition space of a W element that is terminated by knowledge propagation terminals is called knowledge induction.

The terminated event definition space is built with reference to FIG. 25 or alternatively with the reference to FIG. 17. When the W element statement stored in the SUM-Repository is examined all of its variable parameters are identified. In the example shown in FIG. 25, there are four parameters, 2511, 2512, 2513, and 2514, that have defined the value of some w {k,b,p} event, which value was saved on the ISOPS map. The branch 2501 represents these four parameters. Starting from the last one, 2514, the branch 2501 is retraced backwards, towards 2511. During the retracing, the SUM-Model is traced against the direction of execution between two events. Originally the first event is the event {k,b,p}. The second event is the event that defined the value of one of its parameters. In this example, 2514. Further with respect to FIG. 25, the parameter 2514 was produced by the knowledge propagation event that is either a previously approved event or an event that had no ancestors, i.e., system input event or initial data manipulation event. Therefore, this parameter 2514 did not produce a new branch.

Retracing the current branch backwards, parameter 2513 is encountered. The process event which produced the value of this parameter is identified by tracing the SUM-Model against the direction of execution from the event {k,b,p}. With reference to FIG. 25, this value was produced prior to {k,b,p} event by some statement s2 executed at the process time {k2,b2,p2}. The examination of the SUM-Repository entry of this statement identified the presence of two variables, 2521 and 2522.

REPLACEMENT SHEET

25 FEB 1998

- 50 -

Tracing back the model from $\{k_2, b_2, p_2\}$ to the events that defined the values of both parameters 2522 and 2521, the SUM-Analytical Engine finds that in the example of FIG. 25, the value of both of the parameters 2522 and 2521 prior to $\{k_2, b_2, p_2\}$ event, were defined by knowledge propagation terminals previously described. Therefore, new branches were not built for those parameters 2522 and 2521 and therefore the knowledge propagation processes were not activated in those directions.

The branch 2502 therefore was the only branch representing the event definition space of the event $\{k_2, b_2, p_2\}$. The SUM-Analytical Engine is brought back to the branch 2501 to continue the examination backwards against the direction in which the branch was built. The next parameters examined would then be 2512. In our example the value of this parameter 2512 was determined to be produced by some knowledge propagation terminal event (KPT event) and therefore the branch that would represent its event definition space was not built.

SUM-Analytical Engine then retraces the current branch back to the parameter, 2511. In our example its value was also defined by KPT event.

The analysis of parameter 2511 concludes the analysis of the value of W-event $\{k, b, p\}$ event definition space which in the example shown in FIG. 25, is defined by events that define the value of parameters located on the two branches 2501 and 2502. If, during the construction of event definition space we pass an event which was previously uncertain, (in our example it was the event $\{k_2, b_2, p_2\}$) its status is upgraded within the corresponding ACB structure.

Every time that an execution event status is upgraded from uncertain to correct by the knowledge induction process, a corresponding bit in the Analyzed Calculation Base structure 273 is updated from a value of 0 to a value of 1, thus increasing the knowledge of the ACB structure and therefore the knowledge of target process SUM-Model.

A user of the SUM-Analyzer views the system output that is recreated from the system result history by the Incremental Image Extractor/Constructor. Operationally, the user positions a cursor, mouse or other pointing device at the system result. The system result so identified, corresponds to $\{k, b, p, m\}$ parameters of the ISOPS map which defines the event of W element execution. By analyzing the pointing device position within the area defined by entry 3 of the ISOPS map, the corresponding parameter of that W element (system output statement) is defined.

25 FEB 1998

- 51 -

If the so identified system output parameter result is defined by the user as unexpected and therefore incorrect, the SUM-Analytical Engine will start the knowledge deduction process.

Knowledge Deduction Process

Knowledge deduction is performed by a process identical to the knowledge induction process which builds the event definition space for the output events. The output events are traced back to their ancestors. This tracing is terminated in each direction by reaching knowledge deduction terminals which are the same as knowledge induction terminals, i.e., previously approved events or events that have no ancestors.

Unlike knowledge induction, there is no upgrading to the ACB structure knowledge during the knowledge deduction process. Instead, when a previously approved W-event is disapproved, the target process knowledge is dropped to the level, determined by the position of the disapproved W-owner event in the target process time as previously described.

There are two effects of the knowledge deduction process. First, there could be a loss of knowledge which is represented by changing the status of some process addresses from CPA to UPA. With reference to FIG. 16, when W owner of element k,b was defined, its position on the ISOPS map was recorded as one of the attributes of that k,b address. When some other position within the ISOPS map is identified as a failure, SUM-Analyzer will change the status of all addresses previously defined as CPA to UPA if their attributes representing W owner position within ISOPS map point to a

place within the ISOPS map representing a later or equal time as compared with the reference failure. This works because the ISOPS map is built in chronological order in one direction and the process time of the W event chronology is preserved in the ISOPS map.

Second, the position of possible faults or position of required modifications is determined as an area of uncertainty within the definition space of the system output result. This area of uncertainty is defined as those process addresses of this results event definition space that lie outside the correct algorithm space.

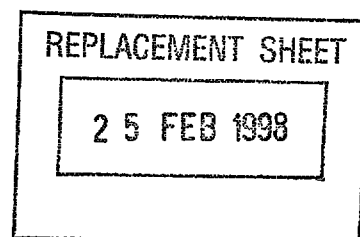
As can be fully appreciated by those skilled in the art, a target process is fully represented by its SUM-Graph and SUM-Model. In that regard, once the SUM-Graph, SUM-Model and SUM-Repository of a target process are generated by the SUM-Analyzer, there is no longer a need for the SUM-Source Generator, SUM-Source, the Compiler/Linker. The Software Understanding Machine can execute and analyze the target process without these above-mentioned components and therefore without the necessity for synchronization signals.

There are two embodiments which could be done. One embodiment uses an interpreter 268 that interprets statements contained within SUM-Repository 246 (shown in FIG. 18(C)). The order of the interpretation is controlled by SUM-Driver which is tracing, i.e. executing, the SUM-Model 243 and therefore is being controlled by the SUM-Model.

Another embodiment uses SUM-Object code which comprises the translated to machine code elements of the SUM-Repository 246. The resulting code, therefore does not contain any control elements such as logic conditions, loop conditions, conditional or unconditional GOTO, RETURN, STOP, or loop terminals. The conditional statements are represented in SUM-Object code only by the expressions to be evaluated. In this embodiment as well as the embodiment which uses an interpreter, the control of the target process is performed by the SUM-Driver tracing (executing) the SUM-Model. In the SUM-Object code embodiment, the SUM-Model becomes a part of the object code as it is traditionally understood since the SUM-Model performs the control functions.

Since the SUM-Model is controlling the process execution the SUM-Model does not need to be informed of the active event in the target process. Therefore, the process synchronization signals are not needed. The only function of the SUM-Interface in these two embodiments is to create the ISOPS and SIPS maps.

With reference to the SUM-Object code embodiment shown in FIG. 18(A) and FIG. 18(B), two components of the SUM-Object code are shown. Shown in FIG. 18(B) are a control component 1800 and an executable component 1801. The control component 1800 has a pointer 1802 to a segment 1804 within the executable component. Each segment within the executable component represents a statement of the SUM-Repository that has been translated into object code.



25 FEB 1998

- 53 -

The Executable Component Address (ECA) attribute 1802 is added to each executable element PA of the SUM-Model. Each segment i.e., 1804, is followed by the Statement Separator Byte (SSB) 1803. SUM-Driver 269 reads the value of ECA from SUM-Model 243, reads the corresponding segment of object code 1803 and passes it to the cpu 241 which is located within 1806, that runs the executable component of SUM-Object Code.

As one of different methods for monitoring system input, the following is implemented: when module 1807 translates a statement representing input from the system terminal into Executable Component of SUM-Object Code, it adds a next statement, that reverses the action of an input from a system terminal to output to the input monitor buffer file 254. For example and with reference to FIG. 21(B), READ (* ,002)CH 2154 is followed by WRITE(M,002)CH 2155.

SUM-Driver then has an option to activate or deactivate this second statement in order to control monitoring of the system input while selecting the next code fragment to be passed to the module 1806.

Since the SUM-Object code architecture consists of two components, control and control-less, this architecture allows for easier modification of the target process. When the change needed is a control change, the control component is changed. When the change needed is a non-control component, a new entry is made in the executable component file and the ECA reference in the control component is updated.

The resolution of the link between the control and non-control components through the ECA is done at a run time. Therefore compiling, recompiling or interpreting is needed only for newly created or modified statements of the non-control component of SUM-Object Code. It is even possible for the interpreter to be invoked only for the newly created or modified statements. This includes those situations where the newly created or modified statement is just a single statement.

The control component of SUM-Object Code can be rearranged by insertion, deletion or modification to the SUM-Model element corresponding to the newly created SUM-Repository element. In some situations, when only the control structure needs to be rearranged, a modification to SUM-Object Code can be made without any changes being made to the non-control component. This is done by rearranging the order of the elements within SUM-Model.

25 FEB 1998

- 54 -

Modification to the SUM-Object code can be done in some situations by deactivating the control component and, therefore, bypassing the corresponding element of the non-control component. Further, if the resolution of the links between the Control and non-control SUM-Object code components is performed at run time, linking or re-linking of a group of two or more target processes implemented by SUM-Object Code is done by linking only the control components of the SUM-Object Code.

Result Oriented Programming

The SUM-Analyzer can address any event of any statement execution through the reference to the system results by using a system pointing device or cursor. Therefore, while building the event definition spaces of the system results, there exists a way to automatically modify the target process code.

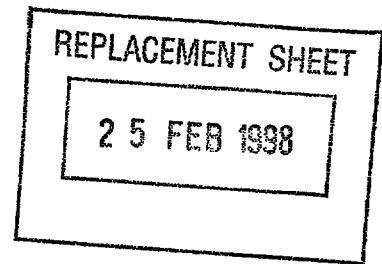
The event definition space of system result events or of the event definition space of an intermediate value calculation event could be reused to create new system results or conditions for new system results. The system results could be rearranged in the process time that corresponds to the system results.

This serves as the basis for the method of Result Oriented Programming (ROP) that is, a very high level language for forward development of a software process. In fact the level of this language grows with the forward development of the target process, since there are more and more events to analyze and more event definition spaces to reuse.

Program Stethoscope

Audio analysis, combined with traditional video analysis allows for better dynamic understanding of programs or system of programs. Execution of any target process can be seen as it is traditionally done by tracing the graph dynamically, but it also can be heard, if one generates sound pitches which correspond to statement positions within the SUM-Graph. Generated this way, "music" would be defined by the process algorithm and is a function of the input data combination.

During the cognitive process of program understanding we are faced with a task of "playing computer". This is by definition a dynamic process. On the other hand a graph (picture) is by definition a static object. The difference between a graphical and sound representation of dynamic process may



- 55 -

be understood by comparing two ways of recreating music: 1) by reading the music score (graphic representation); and 2) by actually playing the music, or hearing it when looking at the score.

Audio effect during a program analysis can be achieved as follows. During parallel or post-life program emulation by applying next recorded terminal address $\{k,b\}$ to the SUM-Model of the program we can trace the program execution on the SUM-Graph of the program algorithm.

The audio effect is added by calling the hardware internal function that allows the generation of an audio frequency and send it to the speaker. A head phone will be a more appropriate addition to the computer when analyzing a system by the described method, and will not affect other programmers.

The frequency can be calculated as a function of two parameters, (k,b) , where a single increment in $\{k\}$ increments the frequency by one discrete step, and a single increment of $\{b\}$ increments a frequency by a larger step, for example by an octave. A branch back will accordingly decrement the frequency.

The duration of beep, corresponding to the tracing of one individual element of the SUM-GRAPH can be set as such that it would be recognized by the human ear. The effect of such audio execution emulation is such, that each process of program execution has its own "music", depending on that process algorithm structure and input data combination. In the case of emulating a loop having of large number of iterations, a code signal can be set to indicate such a loop in order to eliminate a necessity of tracing all the iterations. This code can be for example represented by longer duration signal, followed by different duration signals, each representing let say 10 iterations of the loop. In addition, a graph depicting the process may be made and as each element is executed, the corresponding portion of the graph is highlighted and its characteristic tone is played.

When a particular output result is not being observed for a long time, this analysis can be activated to hear: 1) whether the system is dead; 2) whether the program in loop; 3) or is it waiting for an input. When the last executed element was an input statement a special audio tone can be generated

Most stressful places of the program's algorithm can be heard as a most common frequency and therefore easily identified and the difference in program behavior with different input data combinations can be heard.

A special tone can indicate a call to a child function and a special tone can indicate a return from the call. The audio analysis can be limited to only those signals originally and expanded optionally to indicate process execution within one specific function, if functionality within the function is to be studied.

The Audio analysis can be recorded during the process emulation (alive or post-life) and then replayed without the actual mode emulation in both directions forward and backward.

It can be replayed synthesized with the replay of the system result and thus give a filling of the intensity of the processing necessary for those results achievement;

It can be recorded and replayed for different input data combinations for the same process in order to analyze the difference in intensity of calculations for those different input situations. Given that the potential for the program fault is a function of the number of the elementary processes involved, this analysis will aid in selection simpler (input-process-output) combination during system evolution to a new specifications.

Although various embodiments which incorporate the teachings of the present invention have been shown and described in detail herein, those skilled in the art can readily devise many other varied embodiments that still incorporate these teachings.

REPLACEMENT SHEET

25 FEB 1998

REPLACEMENT SHEET

25 FEB 1998

I claim:

1. An apparatus for automatic analysis of a target process, wherein the apparatus determines the location of faults contained within the target process or location of modifications within the target process required due to a changed expectation of the result, wherein the apparatus generates a secondary process of accumulating knowledge of the target process functionality through knowledge induction, comprising:

an analyzed calculation word attribute for each executable element of the target process;

means for executing the target process and said secondary process,

means for said secondary process to build and propagate an elementary knowledge of the target process functionality through knowledge induction that propagates said elementary knowledge through a place and time within the target process,

wherein said elementary knowledge indicates a knowledge about or an uncertainty about the correctness of an effect of the target process element execution event and,

wherein said elementary knowledge is recorded within said analyzed calculation word and,

wherein the location of fault or location of modification within the target process required due to the changed expectation of the result is obtained through knowledge deduction.

2. The apparatus of the claim 1, wherein the target process is implemented by execution of code in traditional object code form, while synchronization signals are sent from the target process to said secondary process.

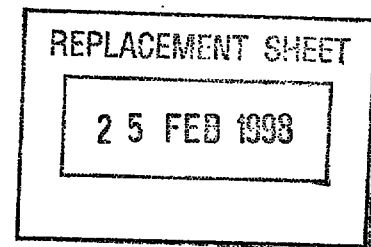
3. The apparatus of the claim 1, wherein the target process is implemented by a driver tracing a model of the target process and presenting corresponding executable elements of the target process represented in their source code form to a language interpreter, while said secondary process is activated by said driver.

4. The apparatus of the claim 1, wherein the target process is implemented by a driver tracing a model of the target process and presenting corresponding executable components of SUM object code to the computer operating system, while said secondary process is activated by said driver.



- 58 -

5. The apparatus of the claim 1, further comprising means for constructing the definitions of modifications within the target process, required due to the changed expectation of the result, wherein said definitions are obtained by reusing event definition spaces of the prior results of the target process.



METHOD AND APPARATUS FOR THE AUTOMATIC ANALYSIS OF COMPUTER SOFTWARE

Abstract of the Disclosure

A method and apparatus for the analysis of a computer process is disclosed. The method consists of the steps of analyzing a target process and constructing a model and repository which are representative of the control and non-control components contained within the original target process respectively. The resultant model and repository are executed, and their execution is representative of the original target process. Structures created during the execution of the model and repository assist a user to identify failures which occur during the execution, and then locate the fault(s) responsible for the failures as well as places within the target source where modifications due to new requirements are needed. An alternative form of object code is proposed which consists of two components, control and non-control components. Previously described model representing the control component constantly increases the intelligence of the resulting object code in respect to the target process being executed through the processes of propagation of elementary knowledge about the correctness or uncertainty of the target process elements execution.

REPLACEMENT SHEET

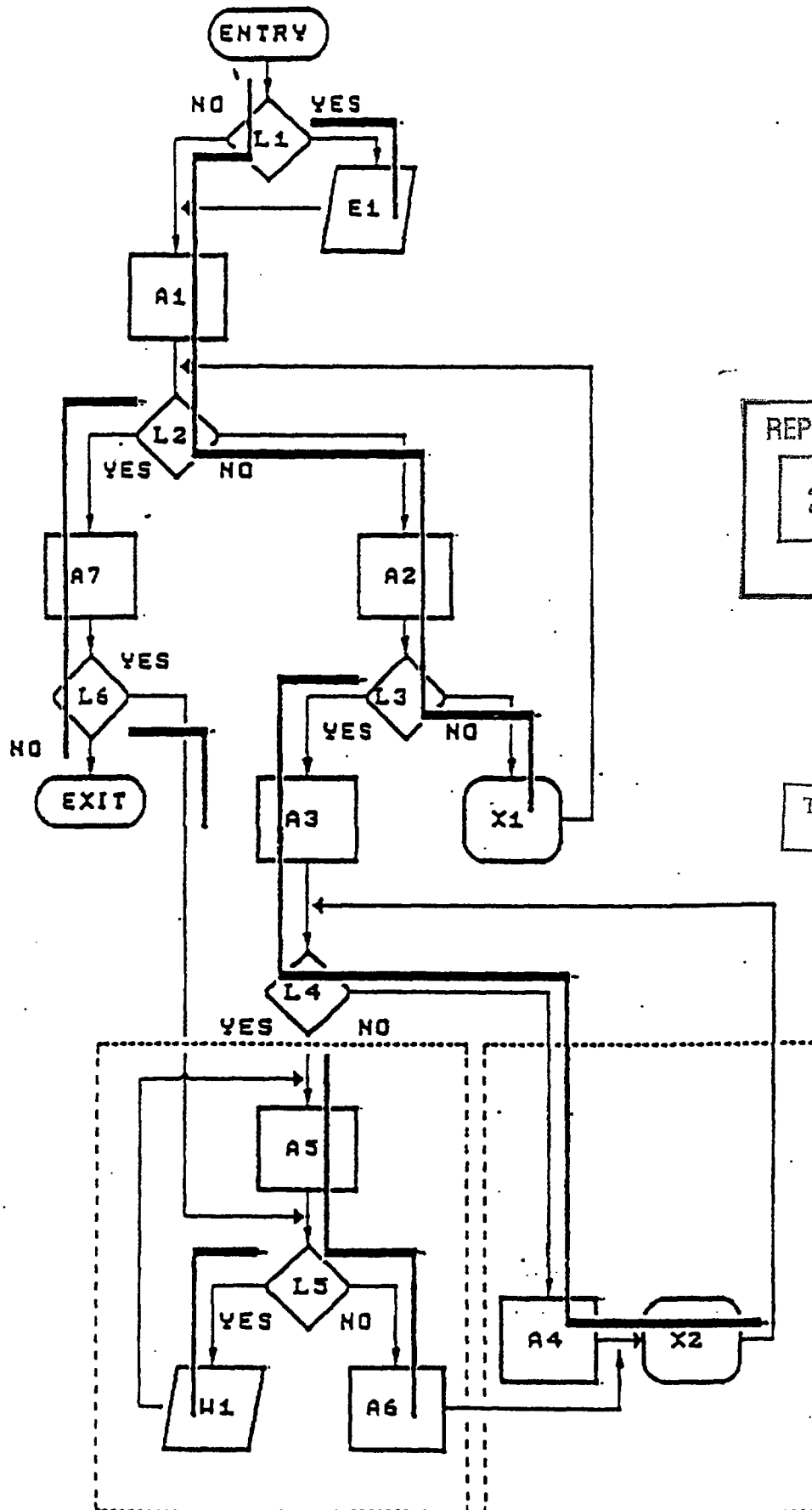
25 FEB 1998

THIS SHEET HAS BEEN
REPLACED

This Sheet Has Been Replaced



FIGURE 1



REPLACEMENT SHEET

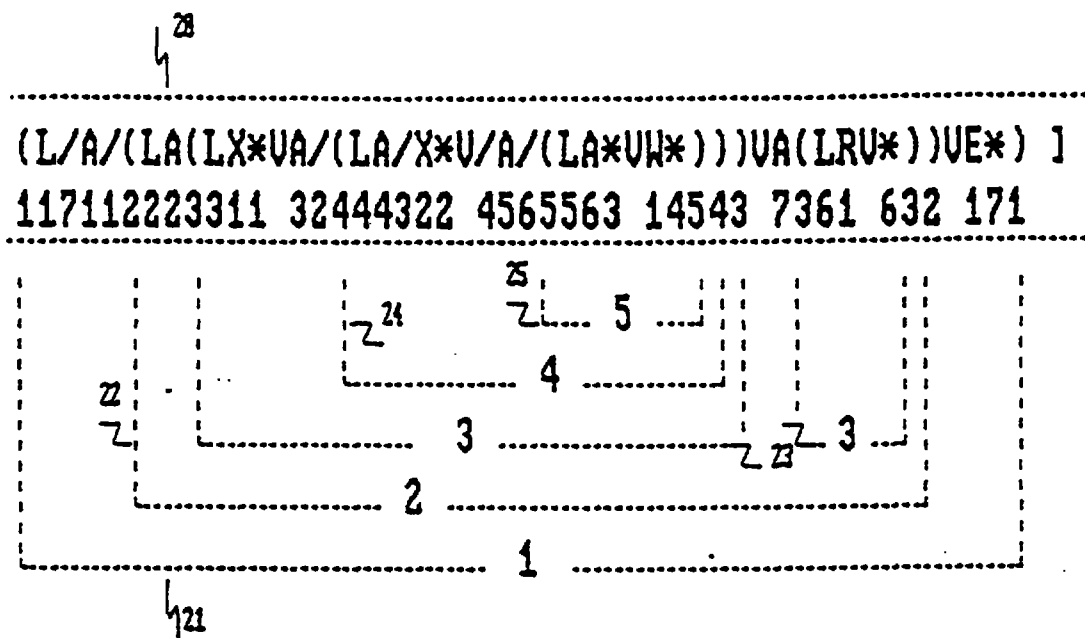
25 FEB 1998

THIS SHEET HAS BEEN
REPLACED

This Sheet Has Been Replaced



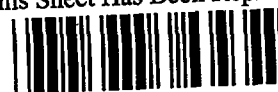
FIGURE 2



REPLACEMENT SHEET

25 FEB 1998

This Sheet Has Been Replaced



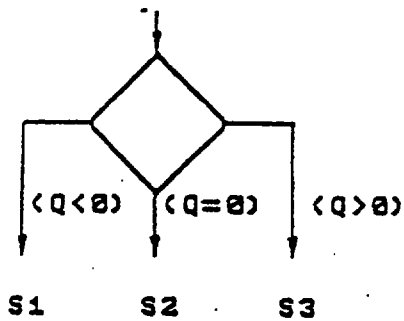
THIS SHEET HAS BEEN
REPLACED

FIGURE 3

EXAMPLE:

IF (Q) S1,S2,S3

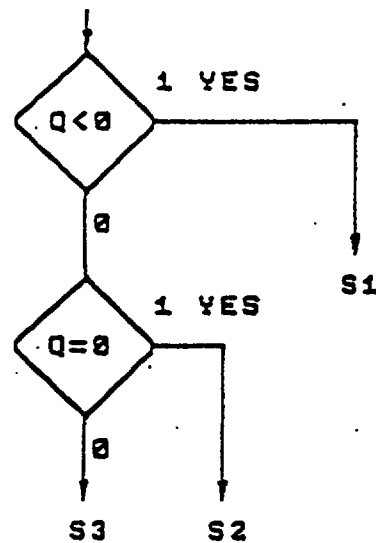
THAT IS :



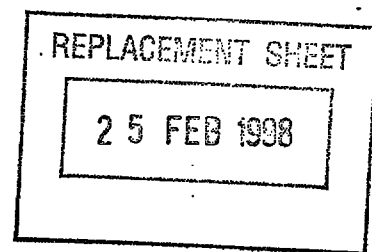
CAN BE REDUCED TO:

IF (Q.LT.0) GO TO S1
IF (Q.EQ.0) GO TO S2
GO TO S3

IS REDUCED TO :

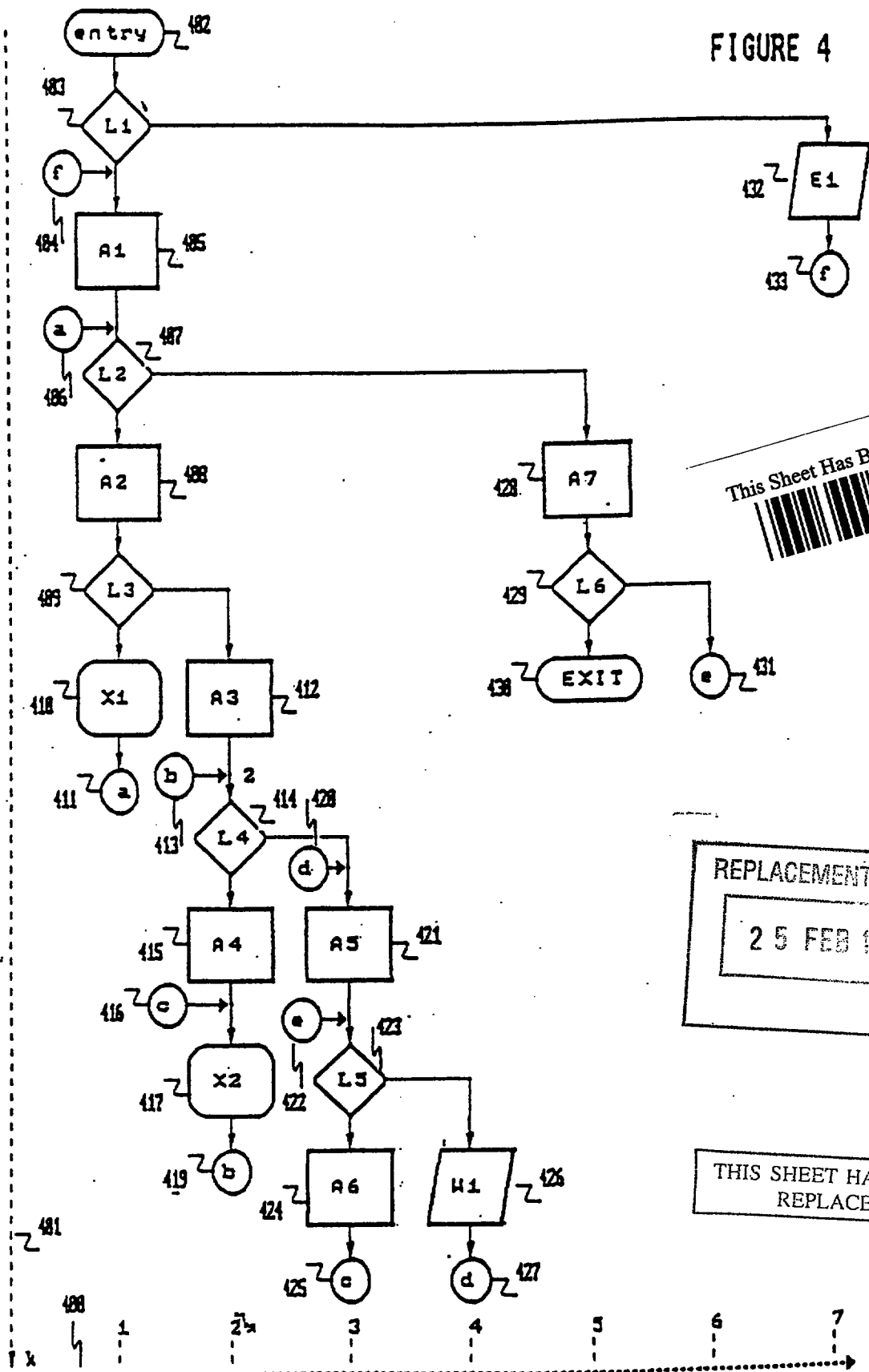


This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

FIGURE 4

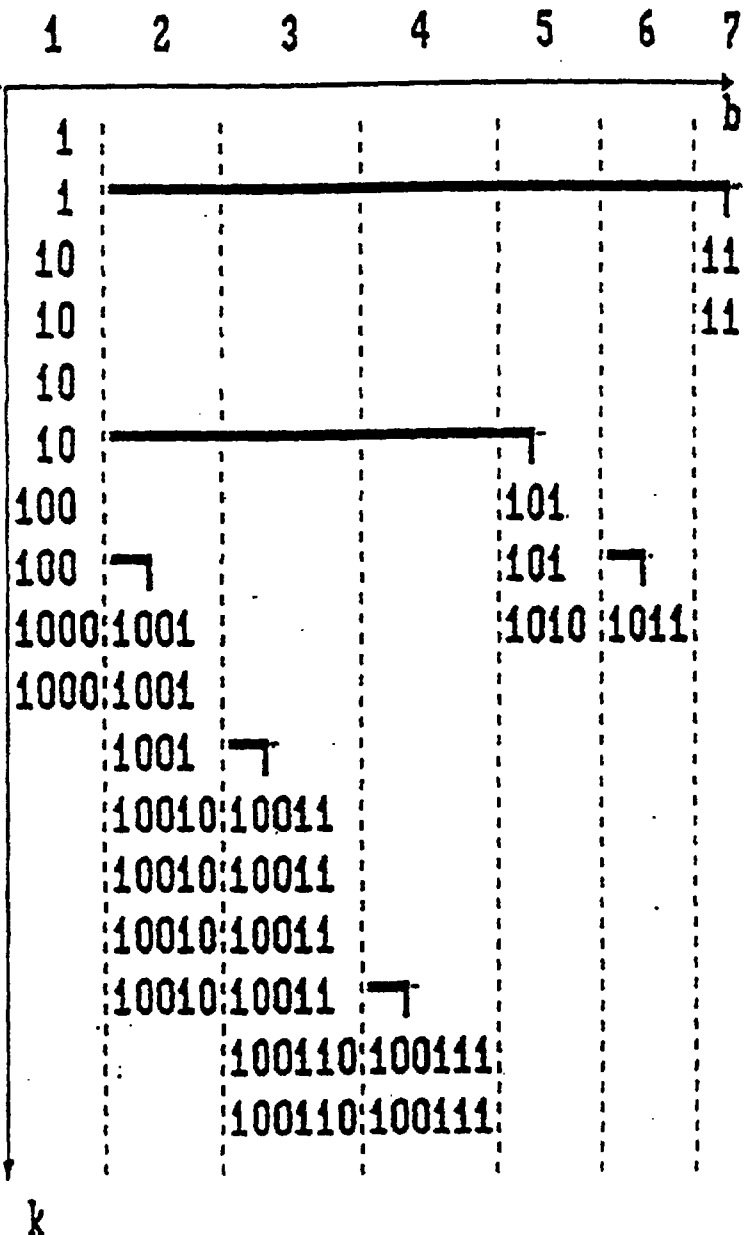
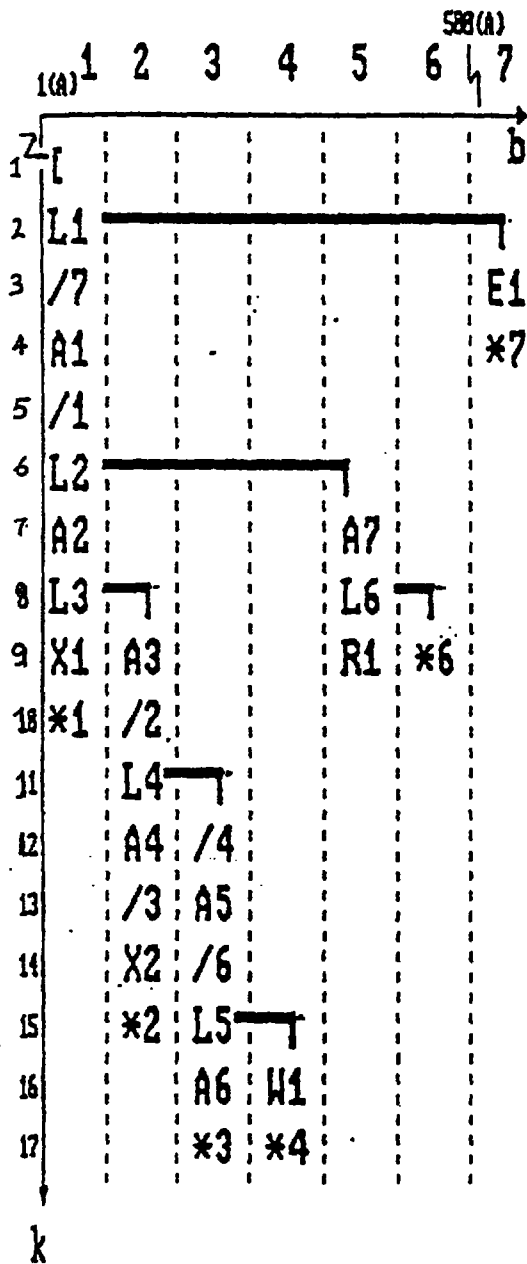


REPLACEMENT SHEET

25 FEB 1998

FIGURE 5 A

FIGURE 5 B

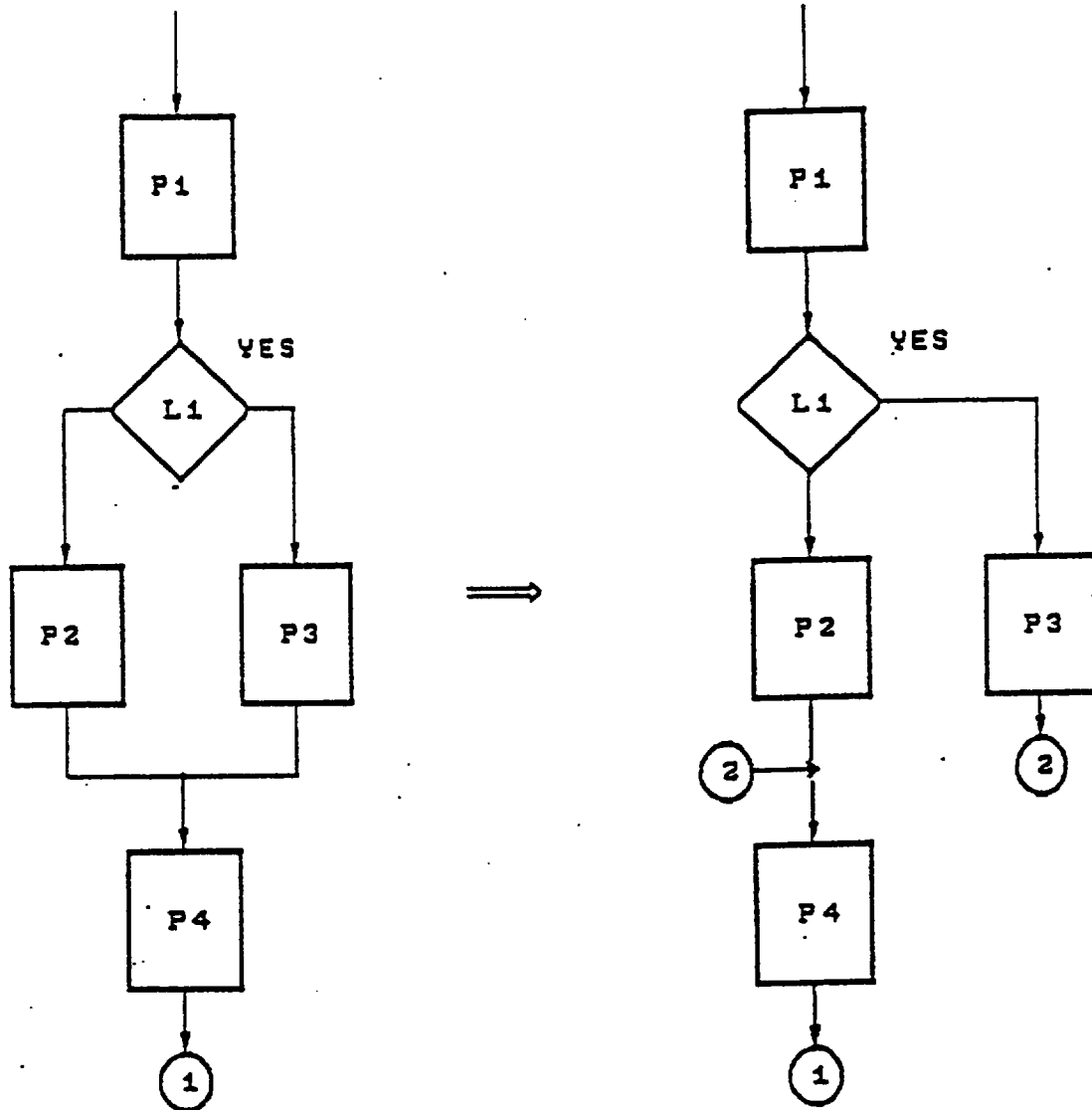


This Sheet Has Been Replaced



THIS SHEET HAS BEEN REPLACED

FIGURE 6



P1
IF (L1) THEN
P3
ELSE
P2
ENDIF
P4

P1 (L1 P2 !2 P4 *1 U P3 *2)
i i

This Sheet Has Been Replaced



REPLACEMENT SHEET

25 FEB 1998

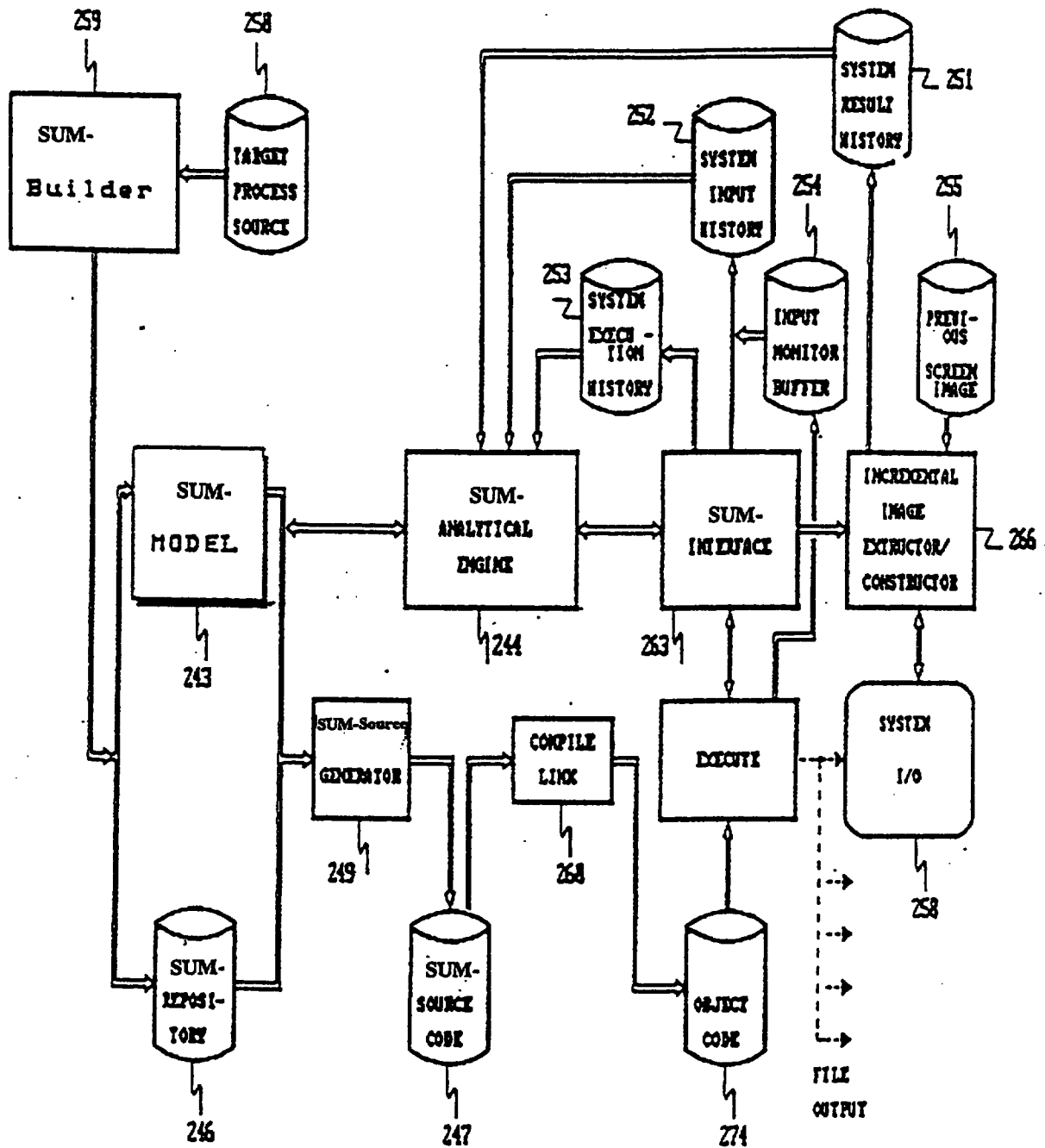
THIS SHEET HAS BEEN
REPLACED

REPLACEMENT SHEET

25 FEB 1998

- 7 -

FIGURE 7



This Sheet Has Been Replaced

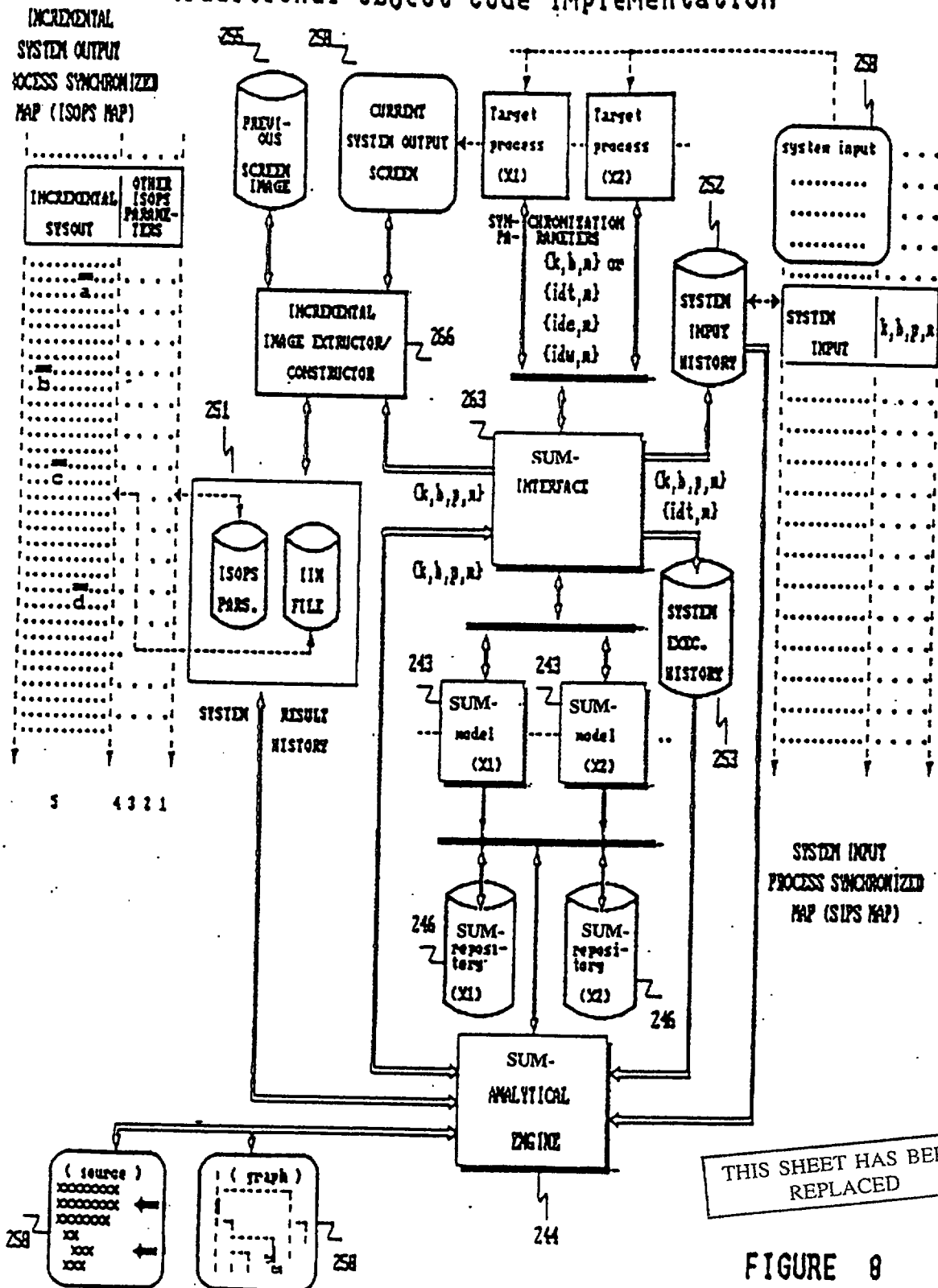


THIS SHEET HAS BEEN REPLACED



25 FEB 1998

Knowledge Induction Machine operation traditional object code implementation



THIS SHEET HAS BEEN REPLACED

FIGURE 8

This Sheet Has Been Replaced

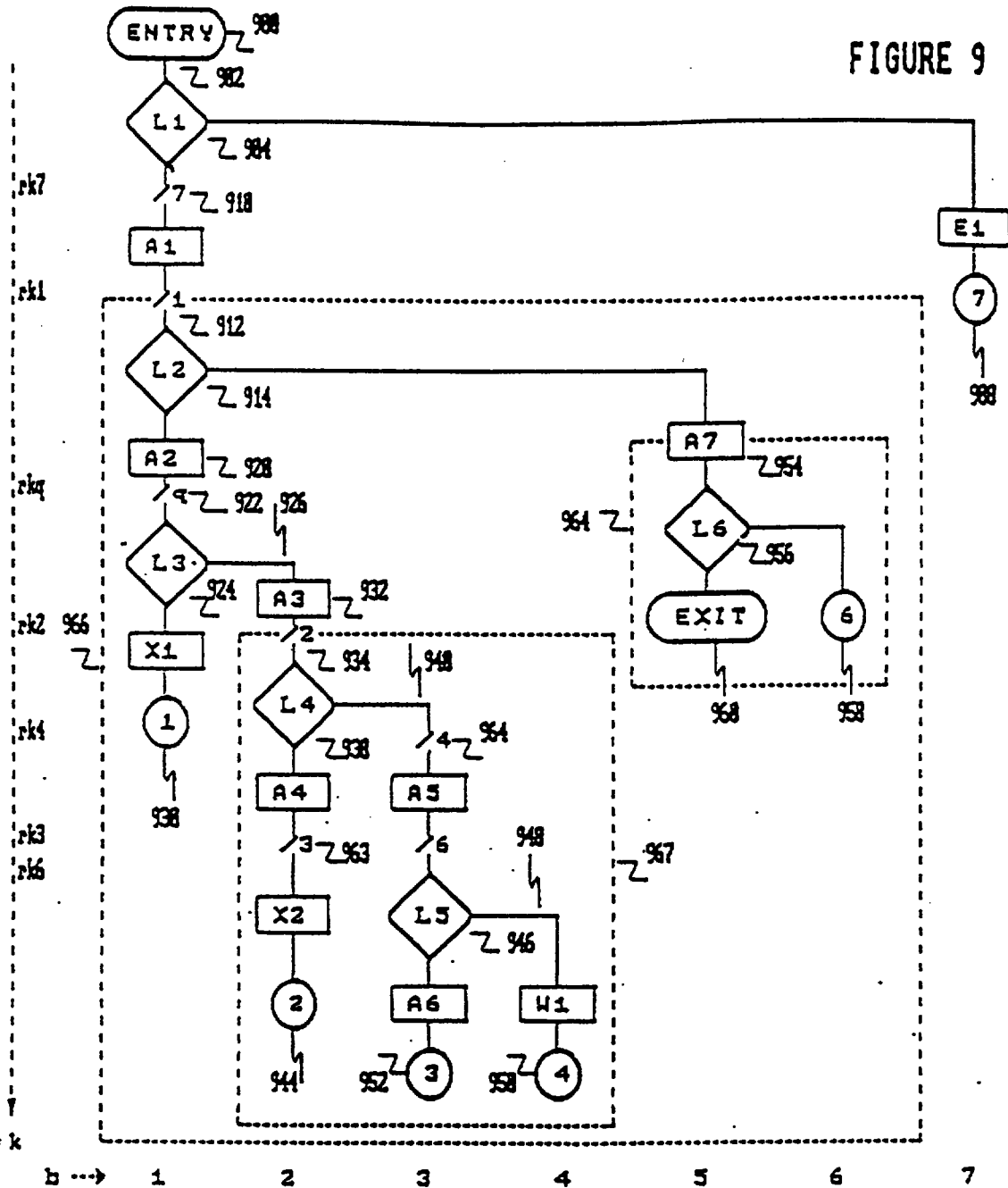


- 9 -

REPLACEMENT SHEET

25 FEB 1998

FIGURE 9



EXIT:	orig.	0	0	0	0	1	0	0
	ACC.	1	0	0	0	1	0	1
IND.	end	rk1	rk2	rk3	rk4	0	rk6	rk7
	orig.	rk1	rk2	rk2	rk2	0	rk2	rk7
PUP	orig.	10	1001	10010	10011	0	10011	10
	end	10	1001	1001	1001	0	1001	10
RBA	orig.	10	1001	10010	10011	0	10011	10
	end	10	1001	10010	10011	0	10011	10
TBA	orig.	1000	10010	100110	100111	1010	1011	11
	end	1000	10010	100110	100111	1010	1011	11

THIS SHEET HAS BEEN REPLACED

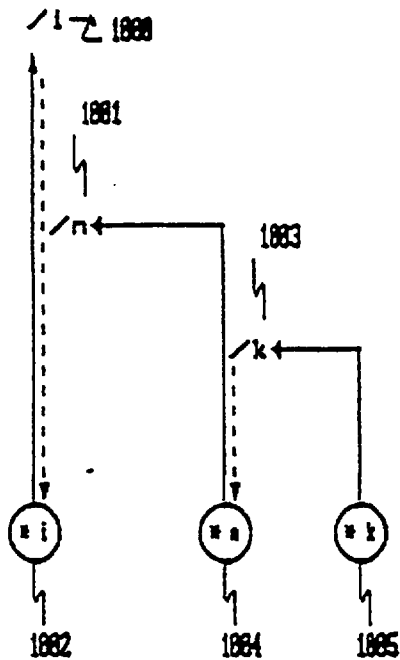


FIGURE 10 B

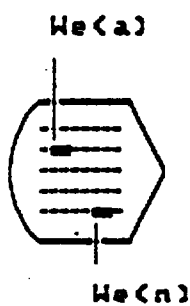
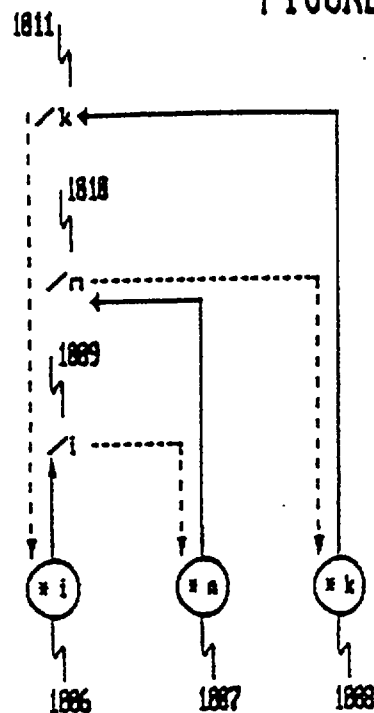
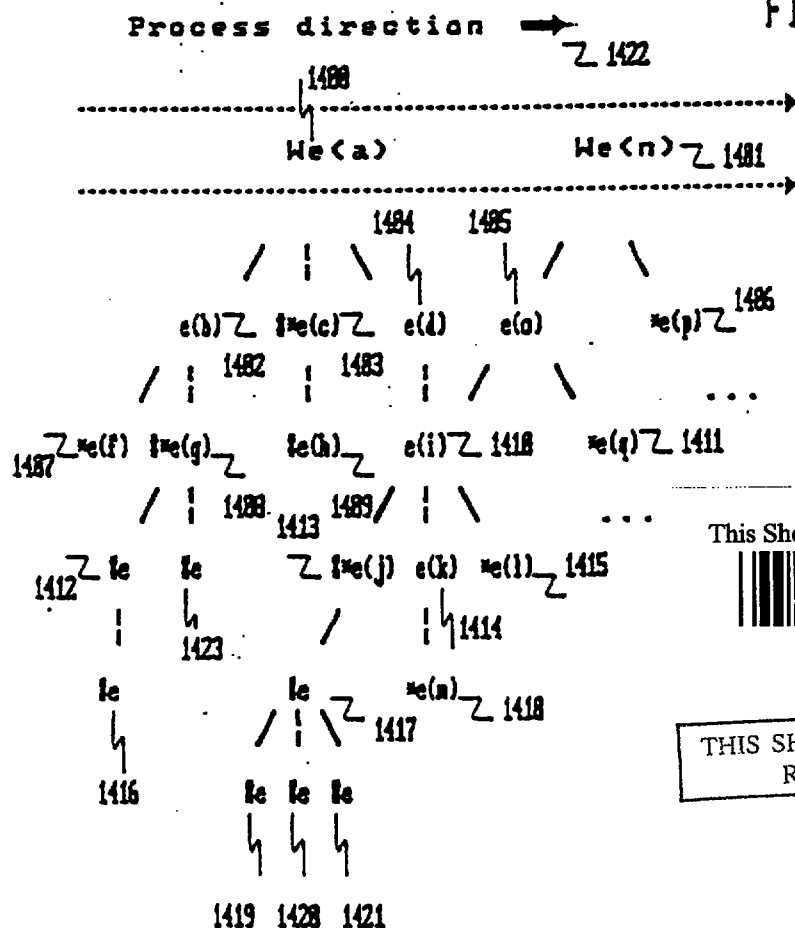


FIGURE 14



This Sheet Has Been Replaced



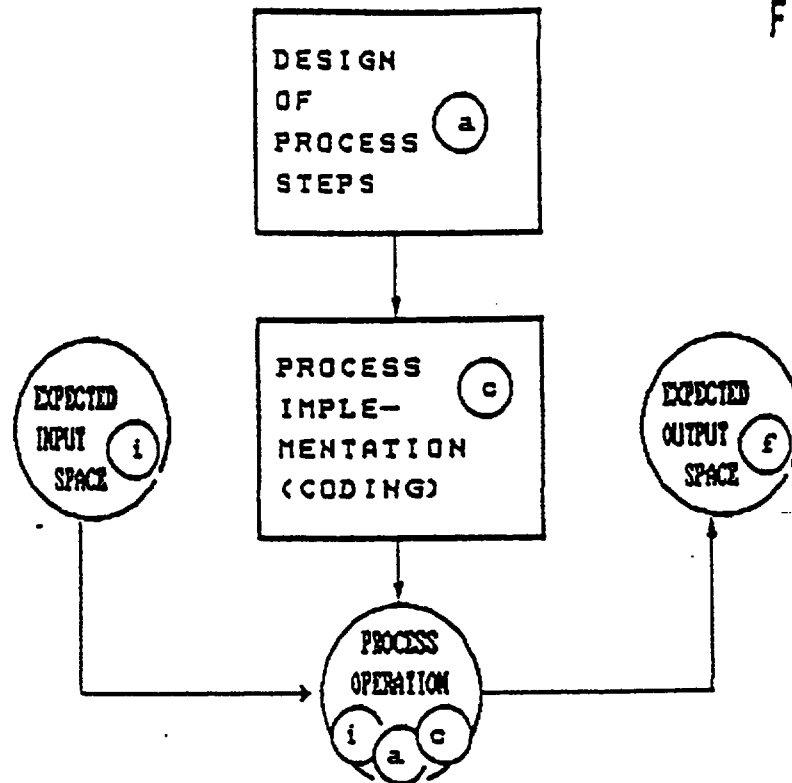
THIS SHEET HAS BEEN
REPLACED

REPLACEMENT SHEET

25 FEB 1998

- 11 -

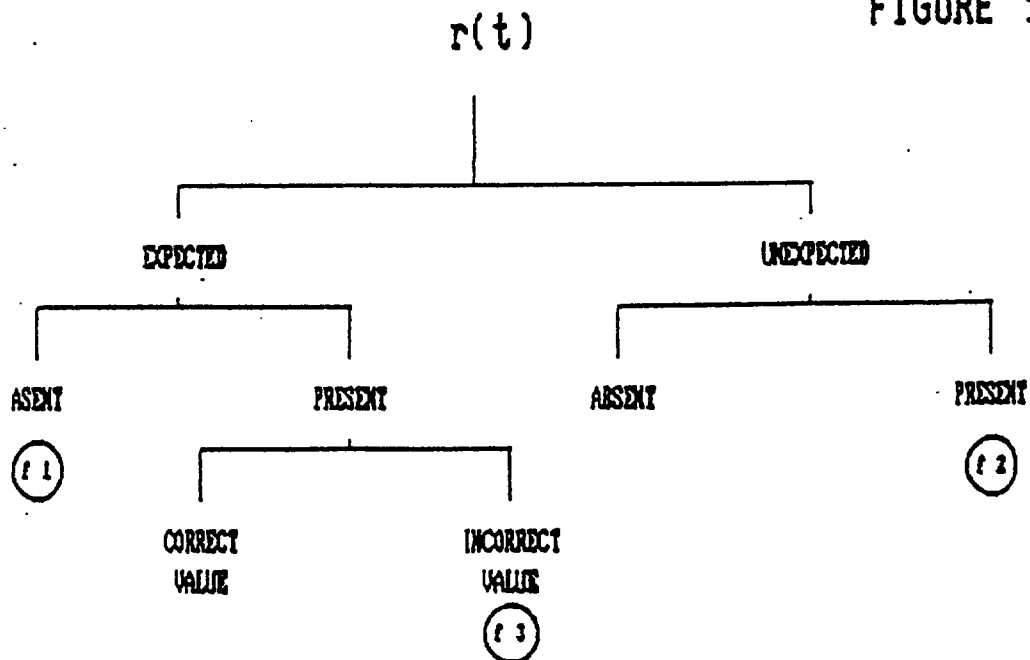
FIGURE 11



This Sheet Has Been Replaced



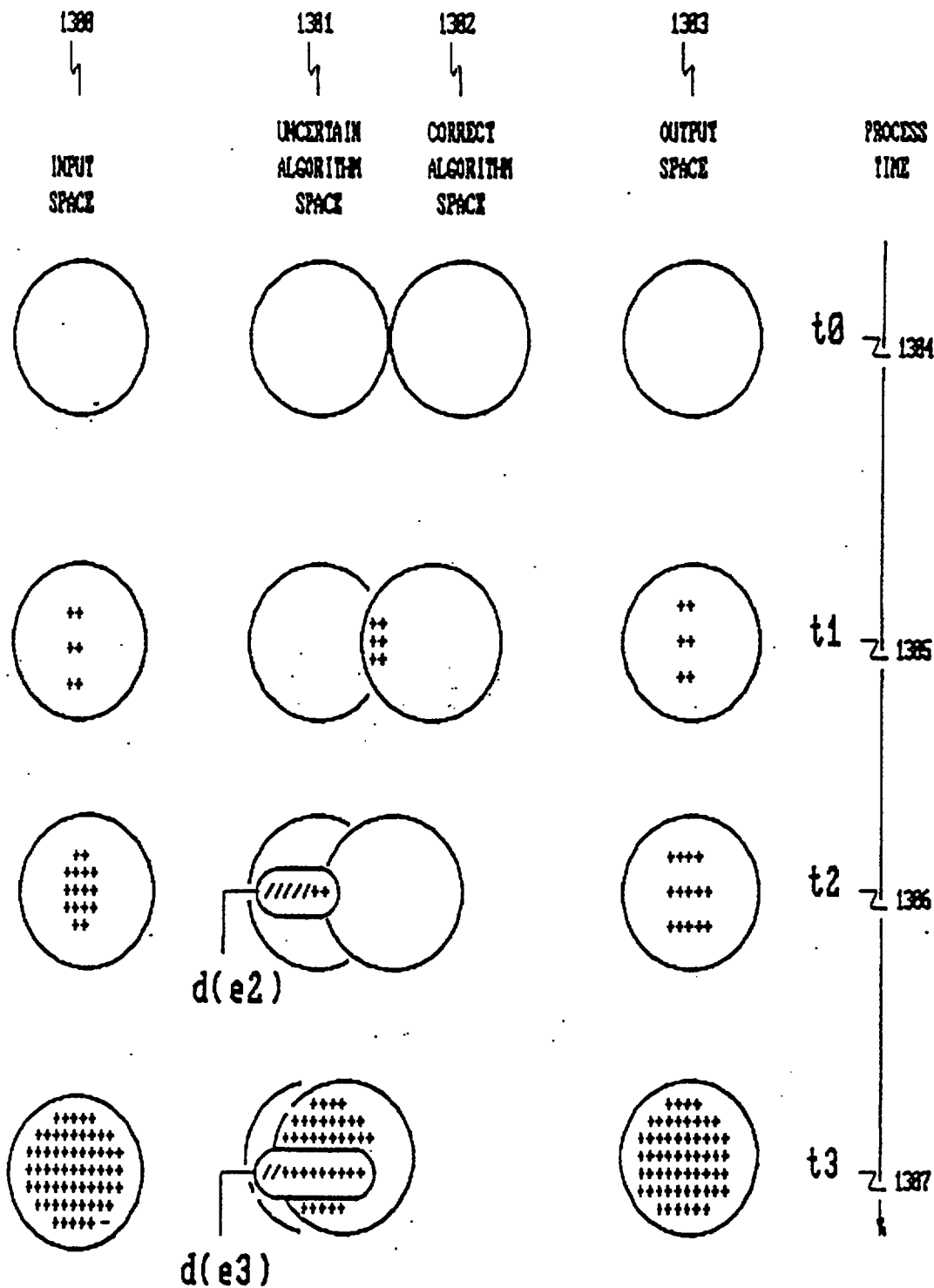
FIGURE 12



THIS SHEET HAS BEEN REPLACED

25 FEB 1998

FIGURE 13



This Sheet Has Been Replaced



THIS SHEET HAS BEEN REPLACED

REPLACEMENT SHEET

25 FEB 1993

FIGURE 15

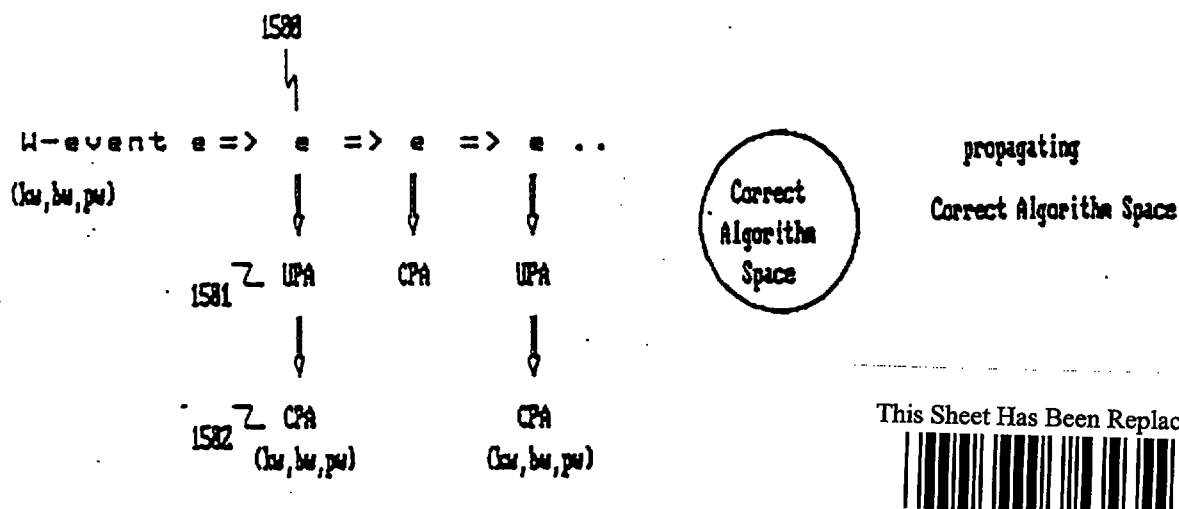
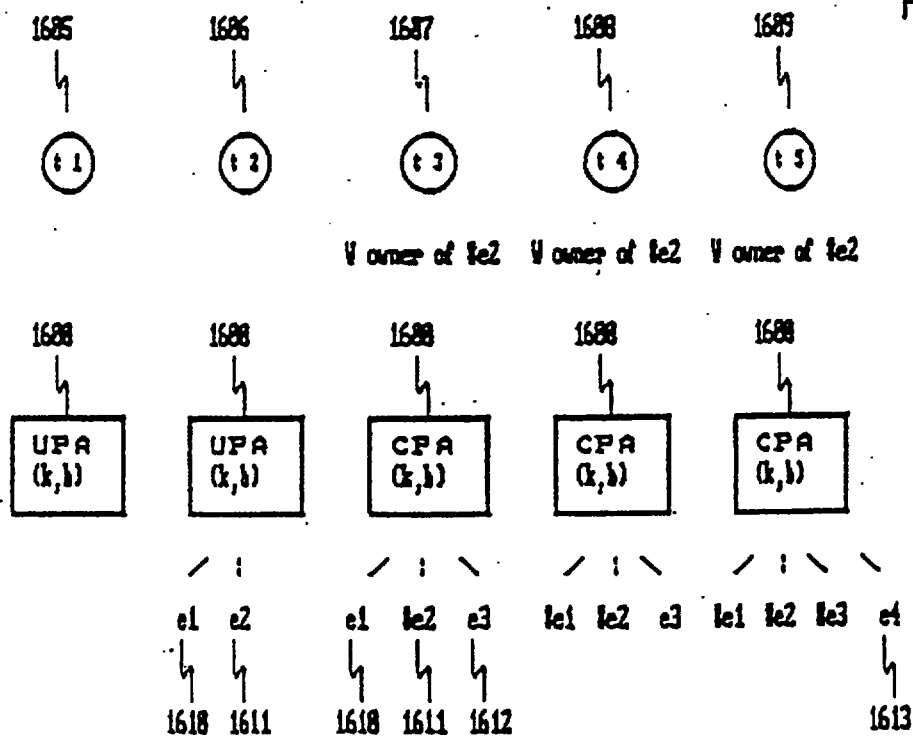
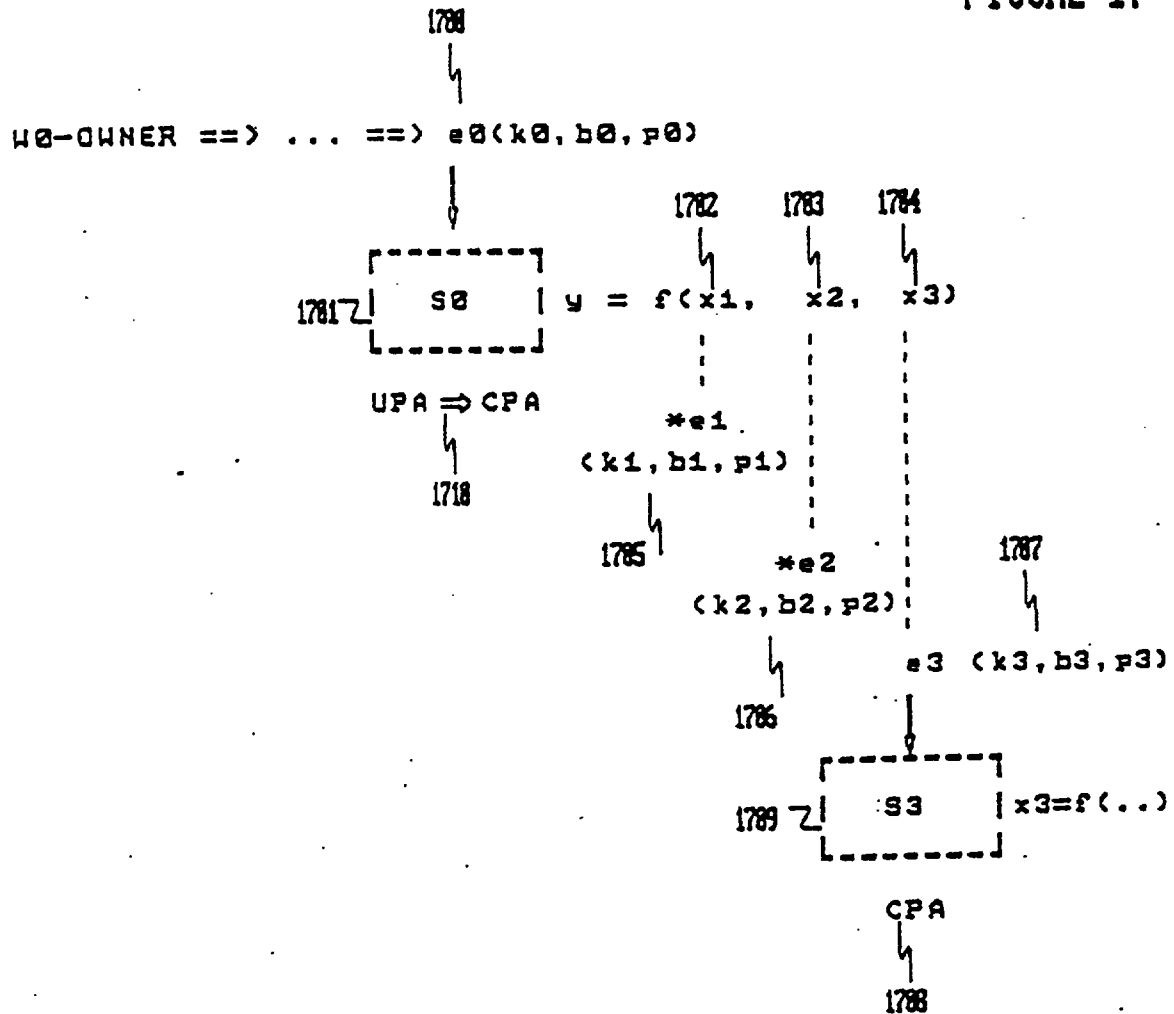


FIGURE 16



THIS SHEET HAS BEEN REPLACED

FIGURE 17



This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

REPLACEMENT SHEET

25 FEB 1993

25 FEB 1998

This Sheet Has Been Replaced

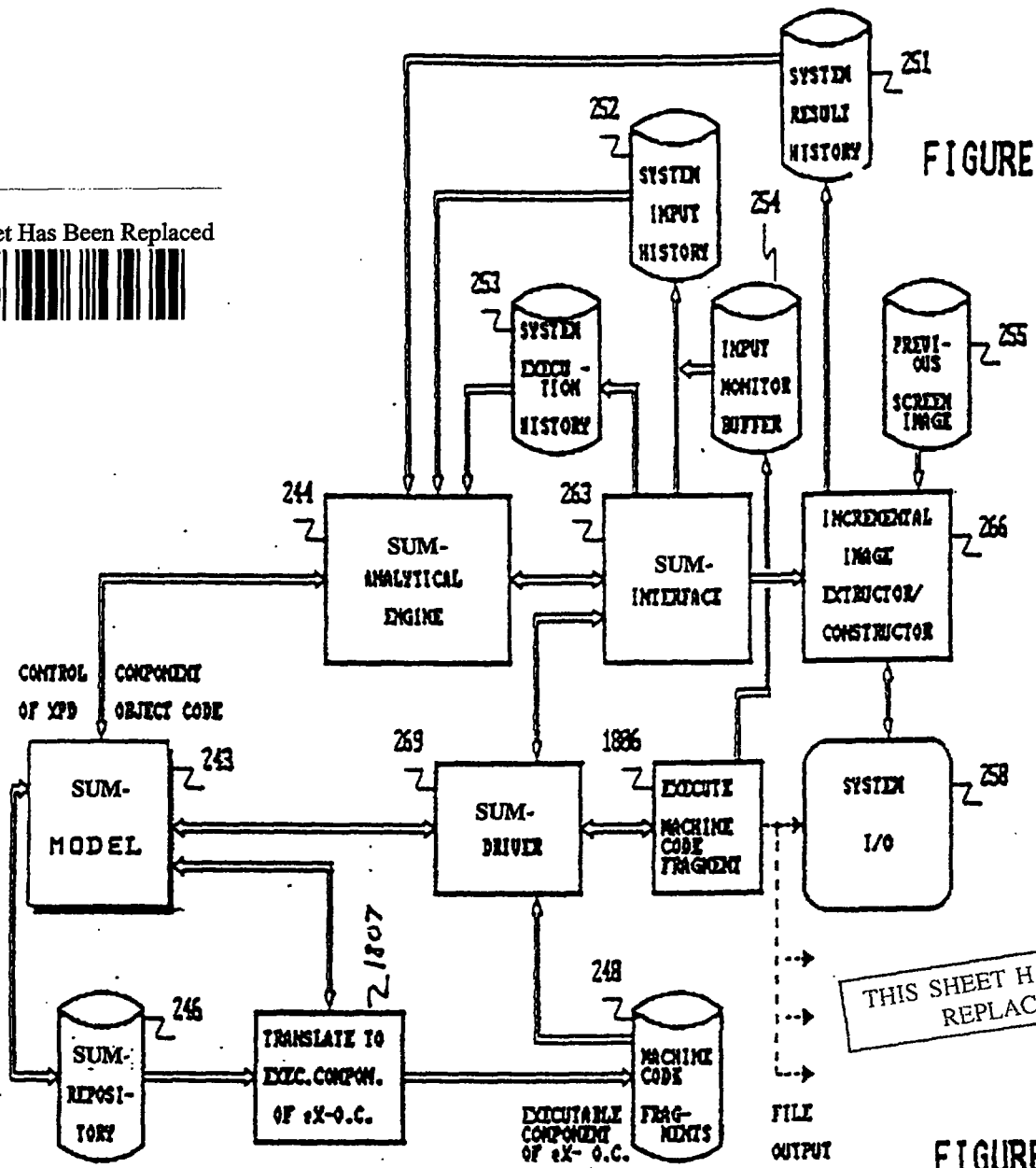
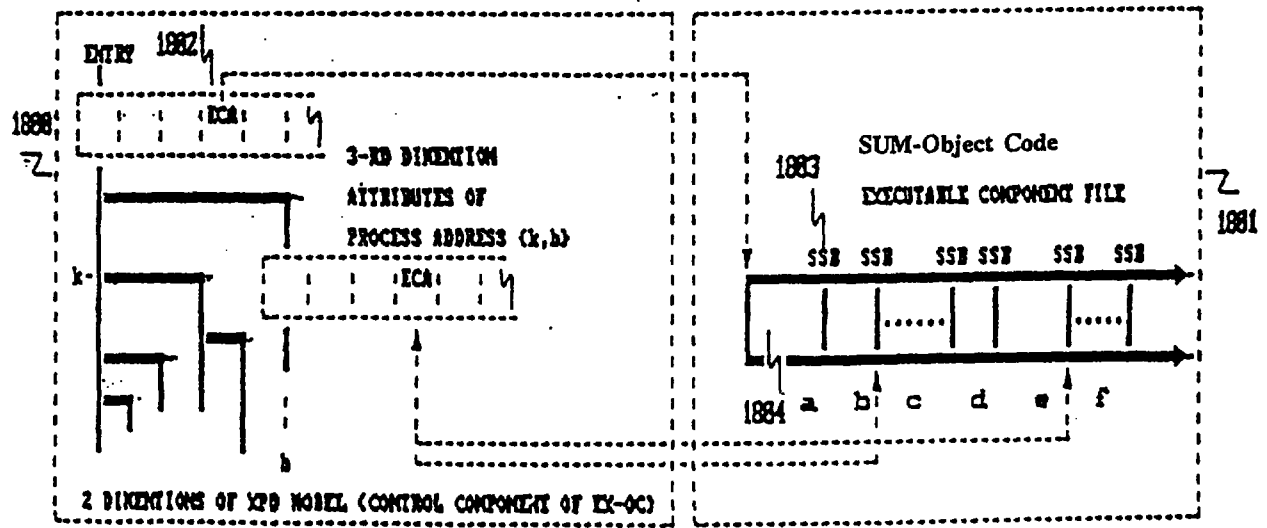


FIGURE 18 A

THIS SHEET HAS BEEN REPLACED

FIGURE 18 B



REPLACEMENT SHEET

25 FEB 1998

- 16 -

This Sheet Has Been Replaced

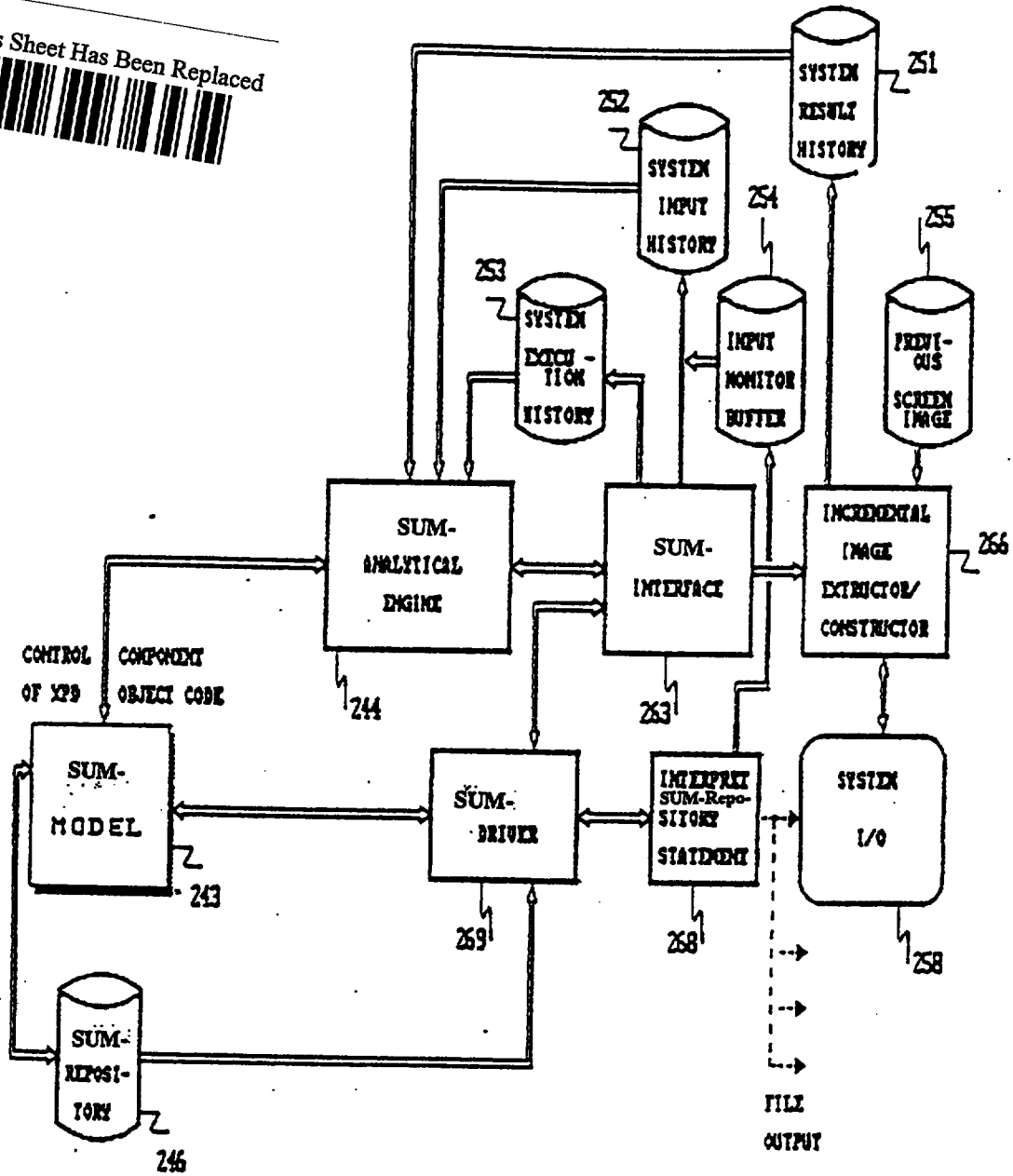


FIGURE 18 C

THIS SHEET HAS BEEN REPLACED

This Sheet Has Been Replaced



- 17 -

REPLACEMENT SHEET

25 FEB 1988

FIGURE 19

Control Area Information Area

	NPW 1981	UDW 1982	1983
1988 Z A1		2000 ₁₀	00011010010001101111
A2		127 ₁₀	00011010010001101111
D1		0 ₁₀	01101110010111101111
1985 Z E1-1	*E1-2	804 ₁₀	00111001100010010001
1986 Z E1-2		0 ₁₀	01111100011110010001
1987 Z X1-1	*X1-2	97 ₁₀	01001010110111001111
1988 Z X1-2	*X1-3	104 ₁₀	011111111111110010001
1989 Z X1-3		0 ₁₀	01000000110111001111

THIS SHEET HAS BEEN
REPLACED

This Sheet Has Been Replaced



- 18 -

REPLACEMENT SHEET

25 FEB 1968

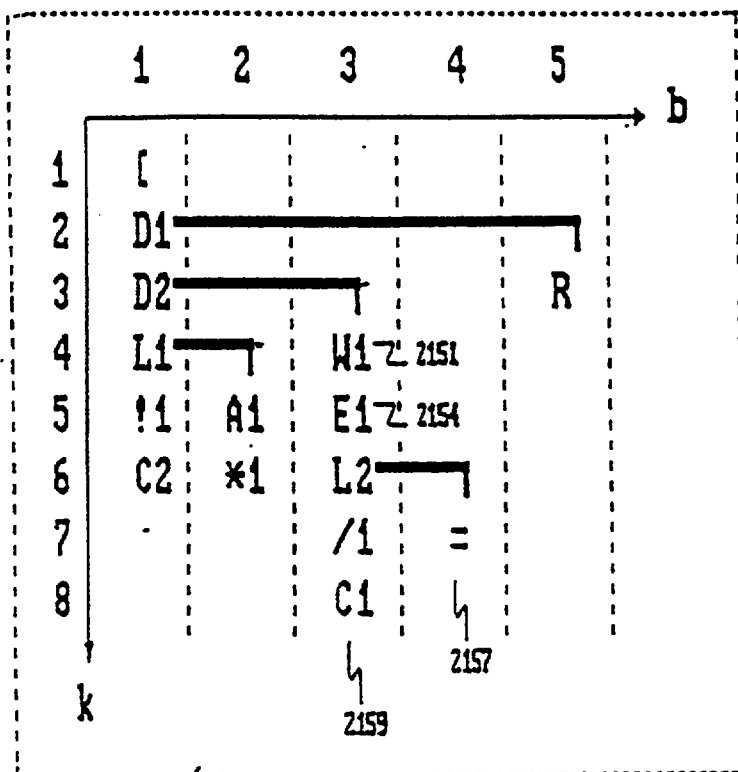


FIGURE 20 A

INTEGER IN(3)

CHARACTER IA(3), IALF(18), CH

DATA IALF/'8','1','2','3','4','5','6','7','8','9'/

881 FORMAT(3X,'IN(1)=' ,I3,' DOESNOT MATCH INTEGER FROM 8 TO 9 ',

C ' TYPE "Y" TO CONTINUE')

882 FORMAT(A)

D1: I=1,3

D2: J=1,18

L1: IF(IN(I).EQ.J-1)

A1: IA(I)=IALF(J)

V1: WRITE(=,881)IN(1)

E1: READ(=,882)CH

L2: IF(CH.EQ.'Y'.AND.CH.EQ.'y')

FIGURE 20 B

THIS SHEET HAS BEEN
REPLACED

25 FEB 1998

- 19 -

FIGURE 21 A

2158

```

SUBROUTINE I2A(IN,IA)
  INTEGER IN(3)
  CHARACTER IA(3),IALF(18),CH
  DATA IALF/'0','1','2','3','4','5','6','7','8','9'/
  DO 18 I=1,3
    DO 55 J=1,18
      IF(IN(I).EQ.J-1)THEN
        IA(I)=IALF(J)
        GO TO 18
      ENDIF
55  CONTINUE
      WRITE(*,188)IN(I)      Z 2151
188  FORMAT(CX,'IN(I)=' ,I3,' DOES NOT MATCH INTEGER ' ,
C 'FROM 8 TO 9 TYPE "Y" TO CONTINUE')
      READ(*,28)CH      Z 2154
28  FORMAT(A)
      IF(CH.NE.'Y'.AND.CH.NE.'y') STOP Z 2157
18  CONTINUE      Z 2159
      RETURN      Z 2161
      END

```

FIGURE 21 B

2159

```

SUBROUTINE I2AXXX(IN,IA)
  INTEGER IN(3)
  CHARACTER IA(3),IALF(18),CH
  $INCLUDE: 'INCL-XPB'
  DATA IALF/'0','1','2','3','4','5','6','7','8','9'/
  881  FORMAT(CX,'IN(I)=' ,I3,' DOES NOT MATCH INTEGER ' ,
C 'FROM 8 TO 9 TYPE "Y" TO CONTINUE')
  882  FORMAT(A)
  DO 3881 I=1,3
    DO 3882 J=1,18
      IF(IN(I).EQ.J-1)THEN
        IA(I)=IALF(J)
        8  CALL XPB(14,2)
        GO TO 3881
      ENDIF
      8  CALL XPB(16,2)
3882  CONTINUE
      WRITE(*,881)IN(I)      Z 2151
      8  CALL XPB(28,2)      Z 2152
      8  CALL XPB(21,2)      Z 2153
      READ(*,882)CH      Z 2154
      8  WRITE(CH,882)CH      Z 2155
      IF(CH.NE.'Y'.AND.CH.NE.'y') THEN
        8  CALL XPB(31,2)      Z 2156
        STOP      Z 2157
      ENDIF
3881  CALL XPB(24,2)      Z 2158
3881  CONTINUE      Z 2159
      8CALL XPB(35,2)      Z 2168
      RETURN      Z 2161
      END

```

This Sheet Has Been Replaced

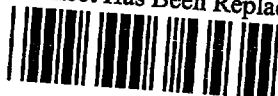
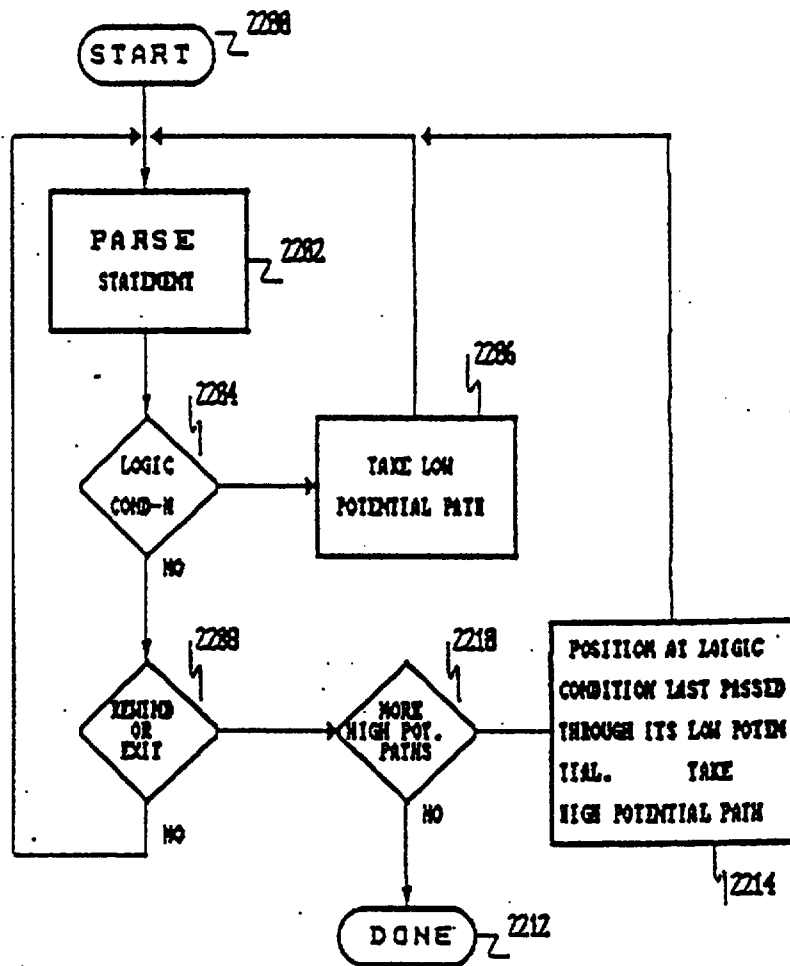
THIS SHEET HAS BEEN
REPLACED

FIGURE 22



REPLACEMENT SHEET
25 FEB 1998

THIS SHEET HAS BEEN
REPLACED

This Sheet Has Been Replaced



REPLACEMENT SHEET

25 FEB 1998

FIG. 23

Assign 0 to the active logic level. Set position to the Process Entry.

Pass through the Main Branch

Assign each passed 'executable statement to SUM-Object

Store assignment in SUM-Repository'

IF Logic Condition

Take Low Potential Solution

Increment Active Logic Level by 1

Represent opened logic level in the frame by opening parenthesis
along with the integer representing Active Logic Level

Start building its 0 subfield by repeating step 2310 (GOTO 2310)

ENDIF

IF reached Process Terminal

The segment is terminated

IF Rewind Terminal

IF corresponding rewind entry position not yet exist in SUM-Frame

Assign next Rewind Terminal index

Insert new rewind entry in corresponding

SUM-Frame position'

ENDIF

ENDIF

Examine Active Logic Level

IF an Active Logic Level is 0

SUM-Frame is built;

STOP

ELSE

Logic Subfield is closed

Examine Closed Subfield

IF the closed subfield = 0-Subfield

Place "V" at the next position of SUM-Frame and therefore open

1-Subfield of the corresponding Logic Condition

Position at the High Potential solution of the corresponding Logic
Condition

Start building 1-Subfield by repeating step 2310 (GOTO 2310)

ELSE

The closed subfield was 1-subfield; both subfields closed;

Close Active Logic field by placing closing parenthesis

with corresponding logic level at the next position
of SUM-Frame

Decrease active logic level by 1

The Segment which terminal was designated in the

SUM-Frame by the corresponding logic condition is now
terminated

Repeat step 2330 (GOTO 2330)

ENDIF

ENDIF

This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

This Sheet Has Been Replaced

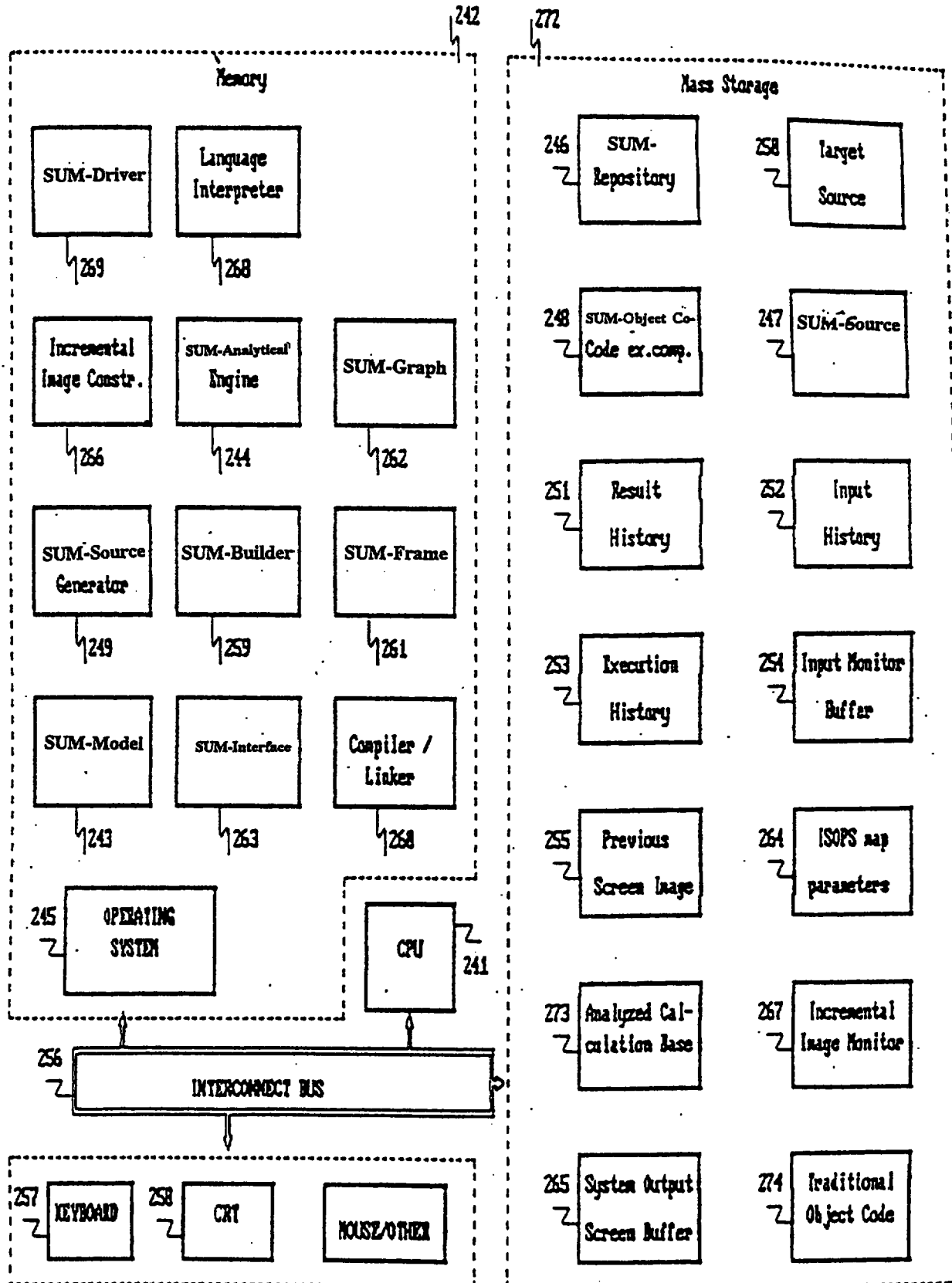


- 22 -

REPLACEMENT SHEET

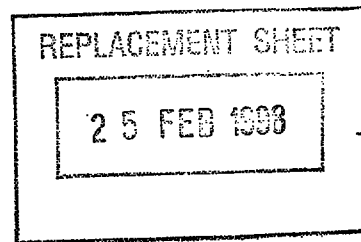
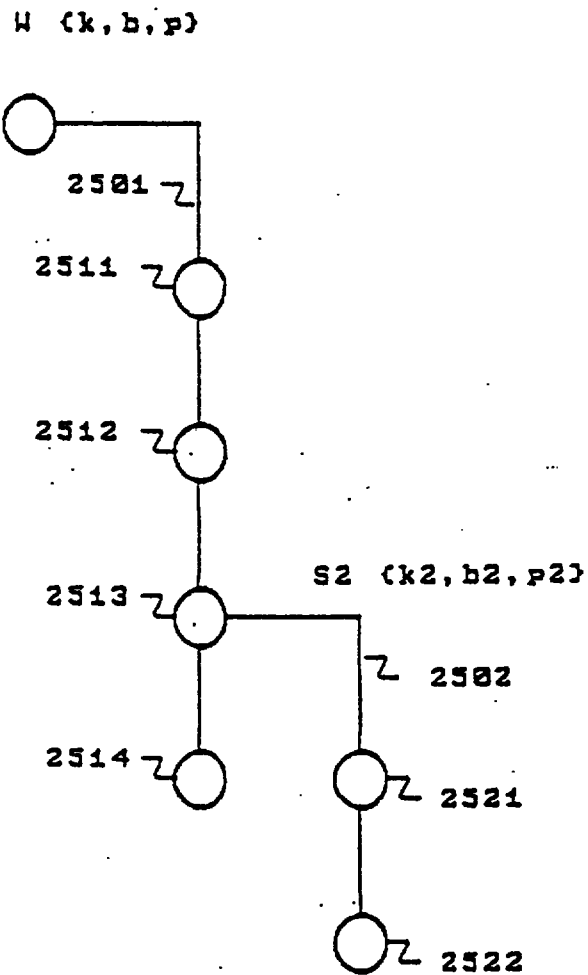
25 FEB 1998

FIGURE 24



THIS SHEET HAS BEEN REPLACED

FIGURE 25



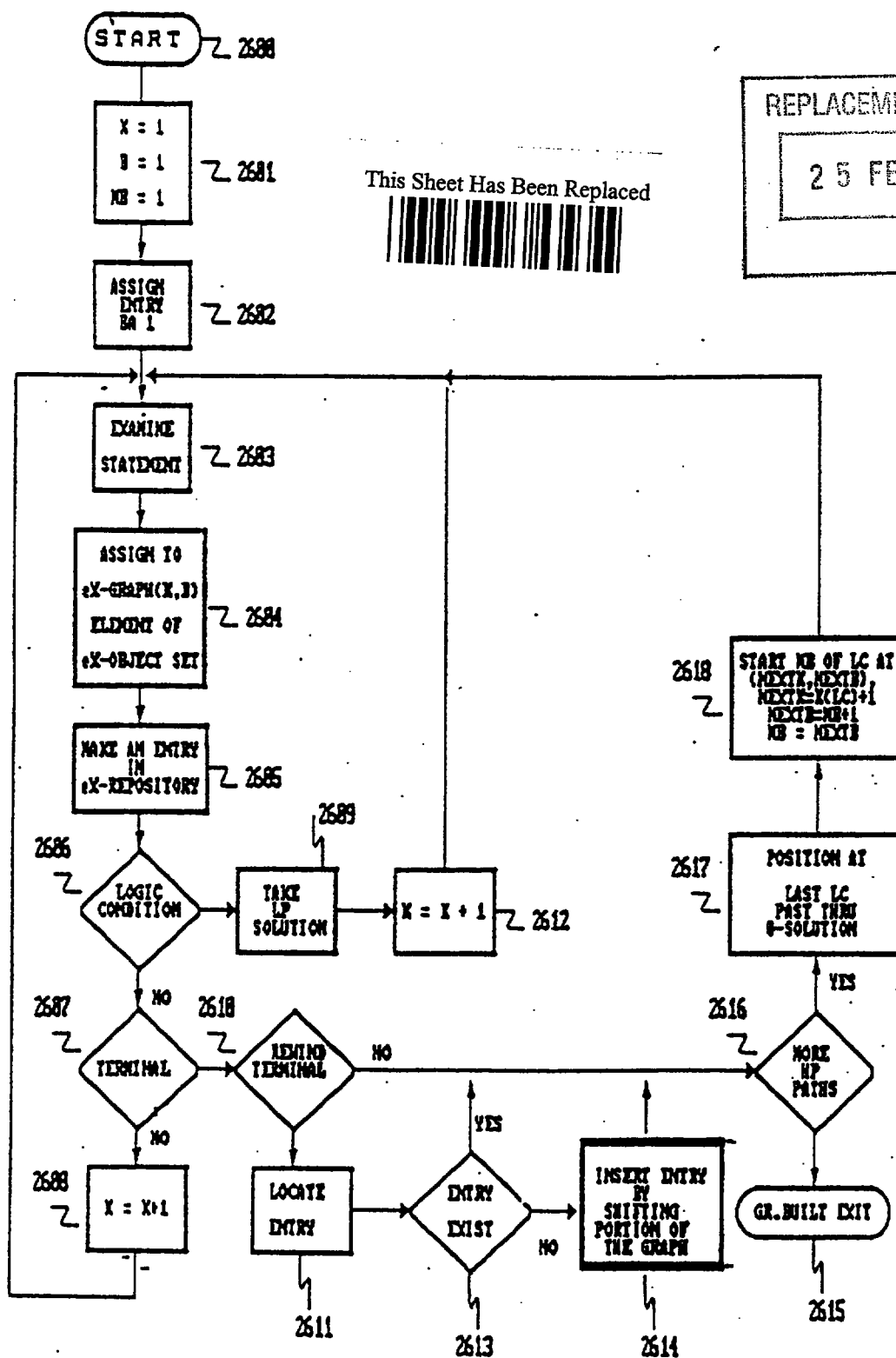
This Sheet Has Been Replaced



THIS SHEET HAS BEEN
REPLACED

THIS SHEET HAS BEEN
REPLACED

FIGURE 26



This Sheet Has Been Replaced



REPLACEMENT SHEET

2 5 FEB 1993

ISOPS map sample format:

1.	2.	3.	4.	5.
W-	screen	screen	displ.	incremental
event -	position	area	in image	image
		changed	monitor	monitor
			file	file

_____ ISOPS parameters file _____ / _____ IIM file _____ /

{k,b,p,m}	{sl,sc}	{l1,c1,l2,c2}	DDDDDD	'xxxxxxxxxxxxxxxxxxxx'
{k,b,p,m}	{sl,sc}	{l1,c1,l2,c2}	DDDDDD	'xxxxxxx'
{k,b,p,m}	{sl,sc}	{l1,c1,l2,c2}	DDDDDD	'xxxxxxxxxxxxxxxxxxxx'
{k,b,p,m}	{sl,sc}	{l1,c1,l2,c2}	DDDDDD	'xxxxxxx'
{k,b,p,m}	{sl,sc}	{l1,c1,l2,c2}	DDDDDD	'xxxxxxxxxxxx'

FIGURE
27

This Sheet Has Been Replaced



REPLACEMENT SHEET

25 FEB 1993

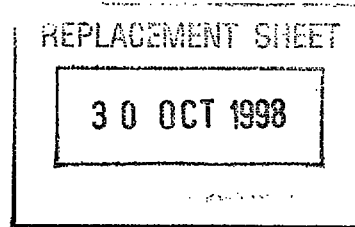
THIS SHEET HAS BEEN
REPLACED

METHOD AND APPARATUS FOR THE AUTOMATIC ANALYSIS OF COMPUTER SOFTWARE

Abstract of the Disclosure

A method and apparatus for the analysis of a computer process is disclosed. The method consists of the steps of analyzing a target process and constructing a model and repository which are representative of the control and non-control components contained within the original target process respectively. The resultant model and repository are executed, and their execution is representative of the original target process. Structures created during the execution of the model and repository assist a user to identify failures which occur during the execution, and then locate the fault(s) responsible for the failures as well as places within the target source where modifications due to new requirements are needed. An alternative form of object code is proposed which consists of two components, control and non-control components. Previously described model representing the control component constantly increases the intelligence of the resulting object code in respect to the target process being executed through the processes of propagation of elementary knowledge about the correctness or uncertainty of the target process elements execution.

Figure 8A and 8B



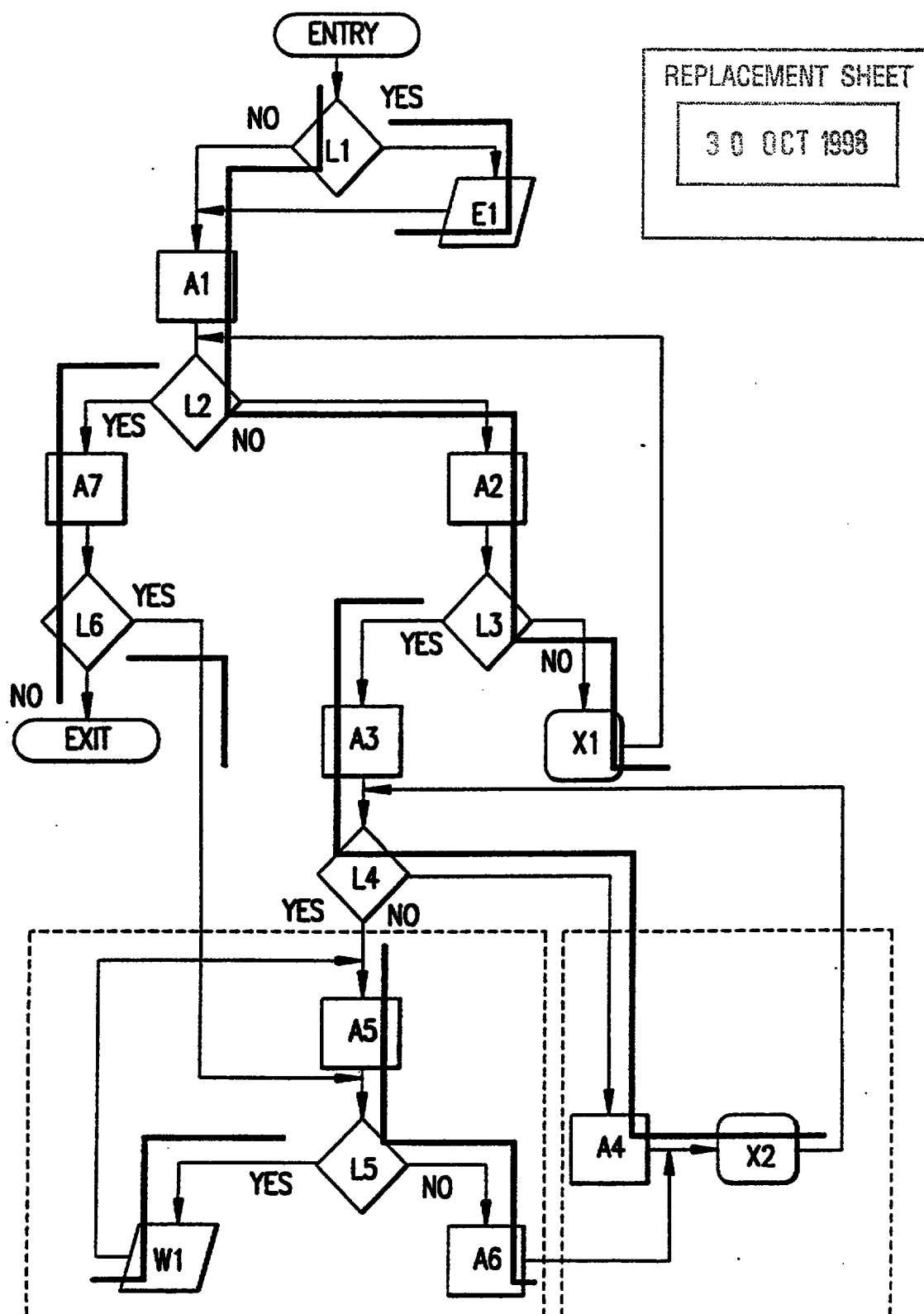


FIG.1

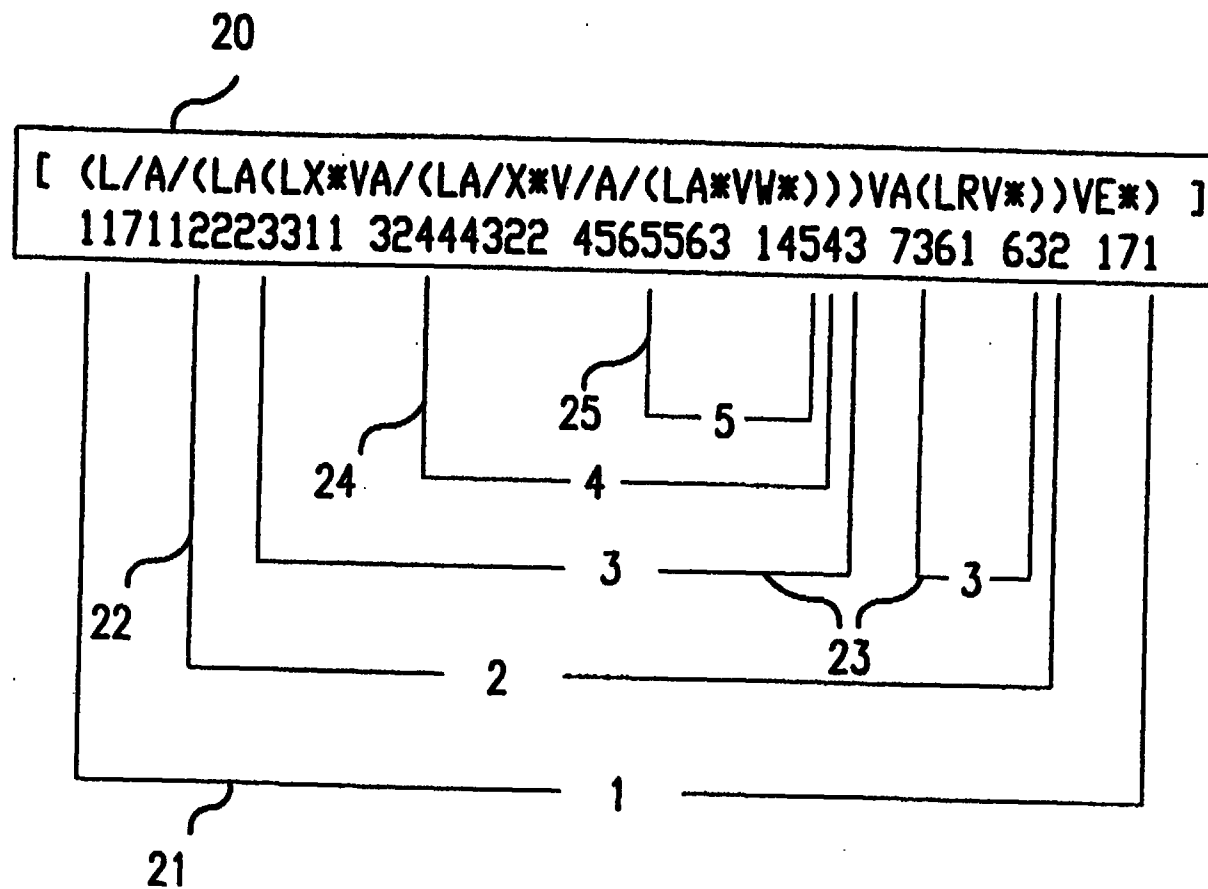
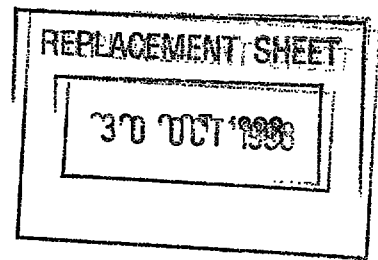


FIG.2

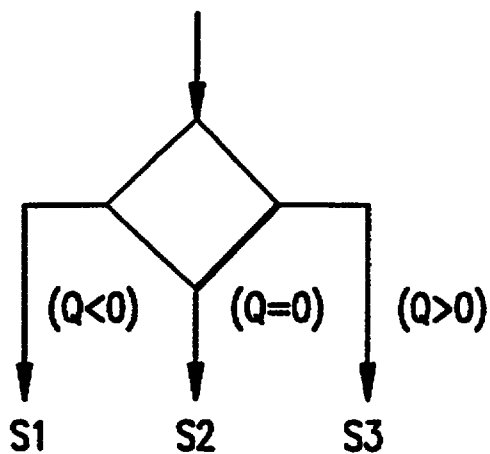
REPLACEMENT SHEET
 30 OCT 1998



EXAMPLE:
IF (Q) S1, S2, S3

CAN BE REDUCED TO:
IF (Q.LT.0) GO TO S1
IF (Q.EQ.0) GO TO S2
GO TO S3

THAT IS:



IS REDUCED TO:

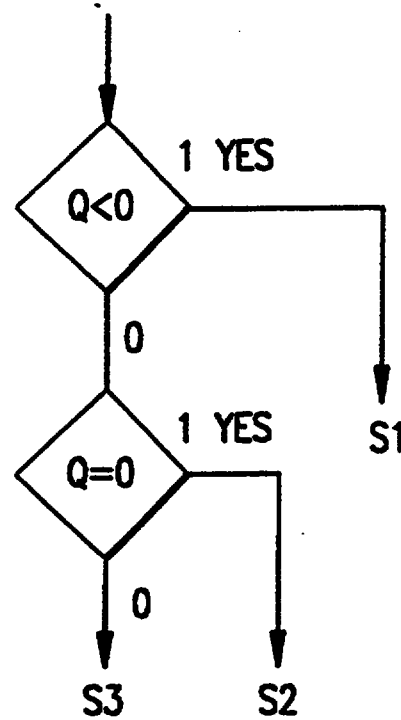


FIG.3

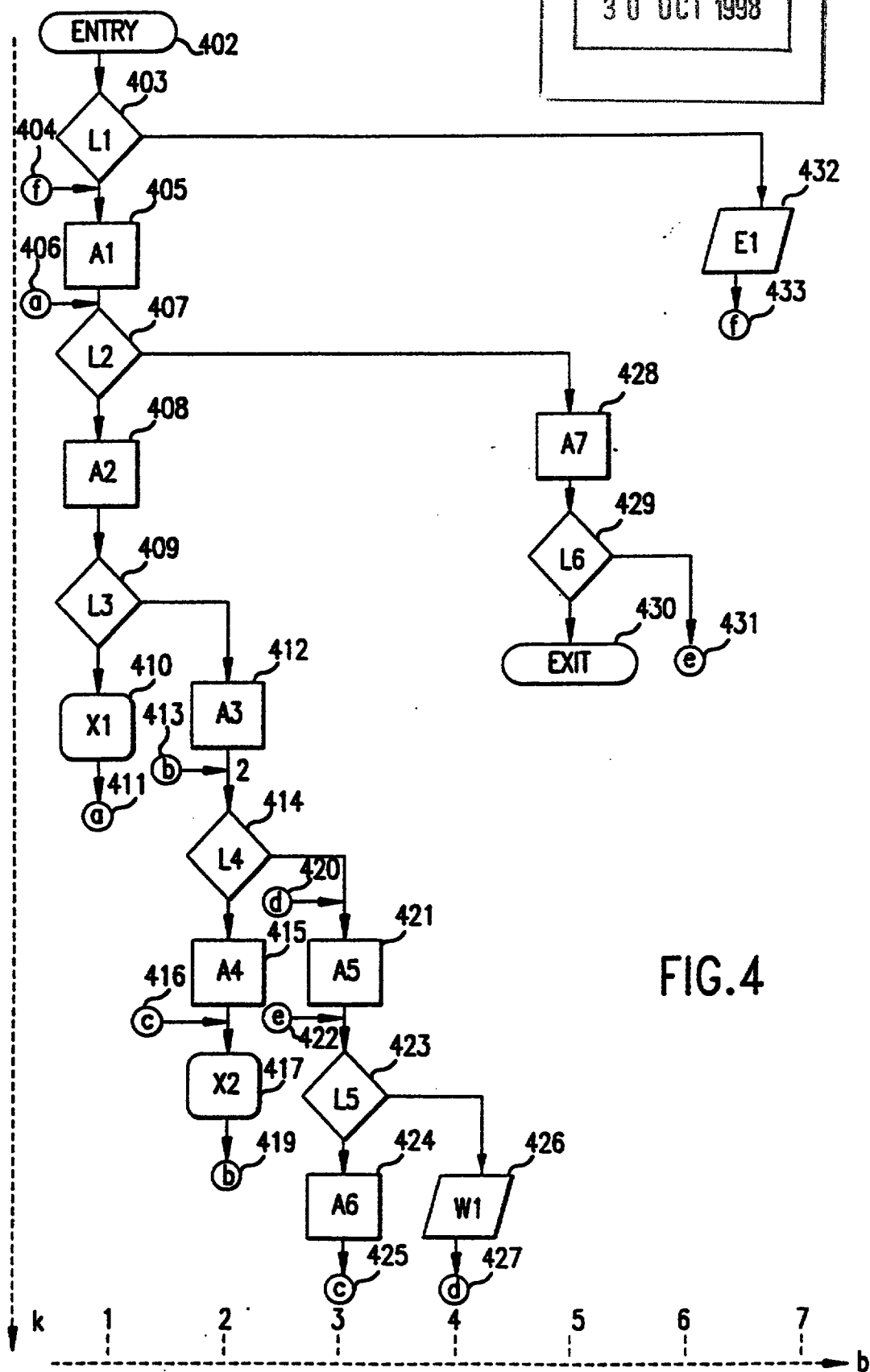
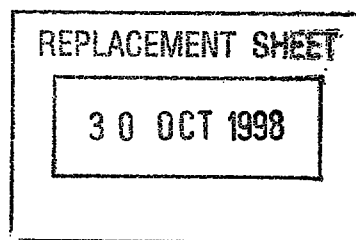


FIG.5B

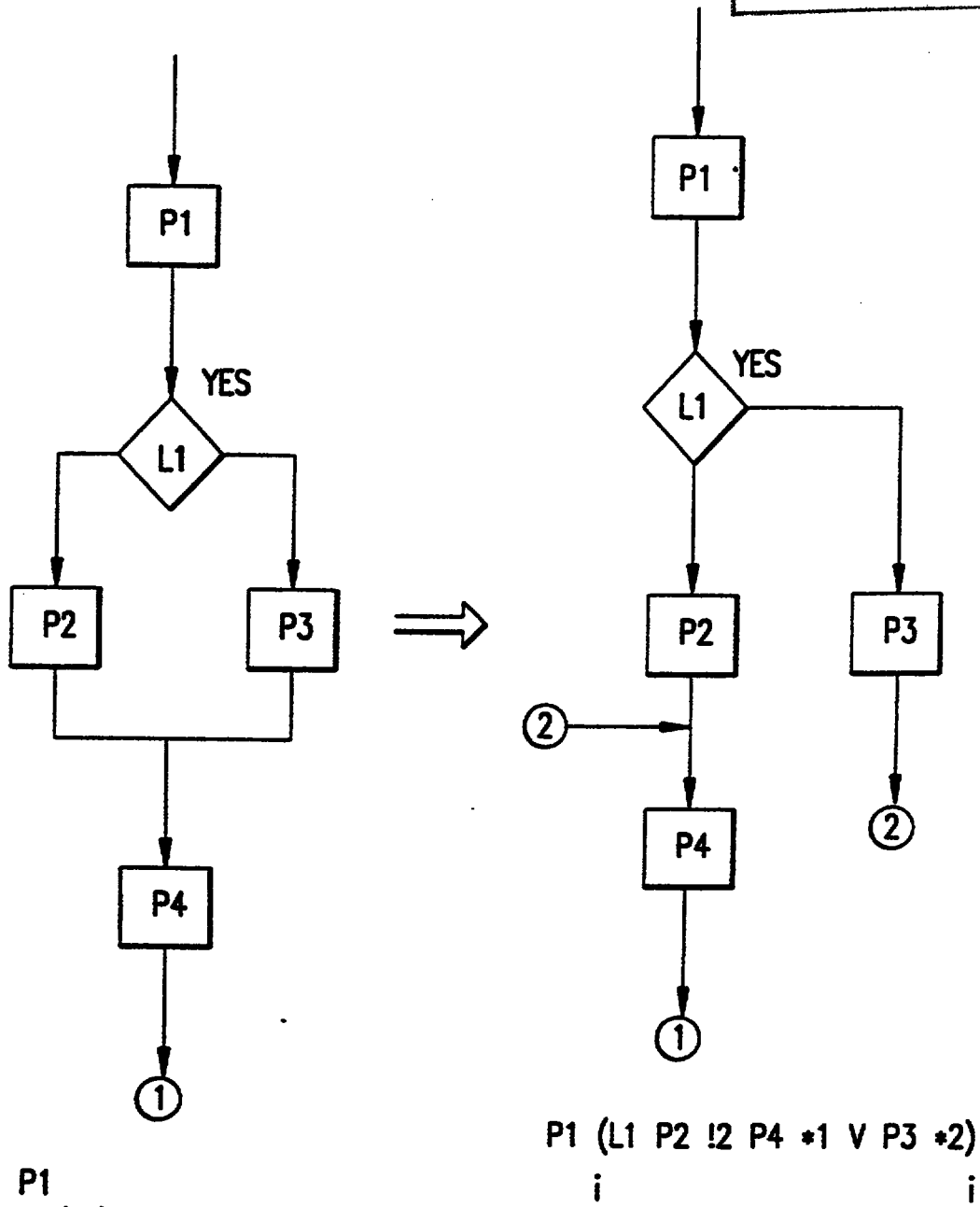
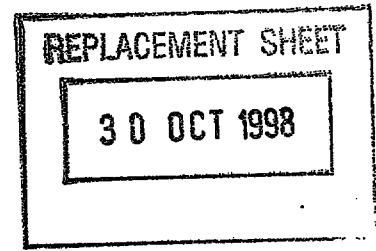
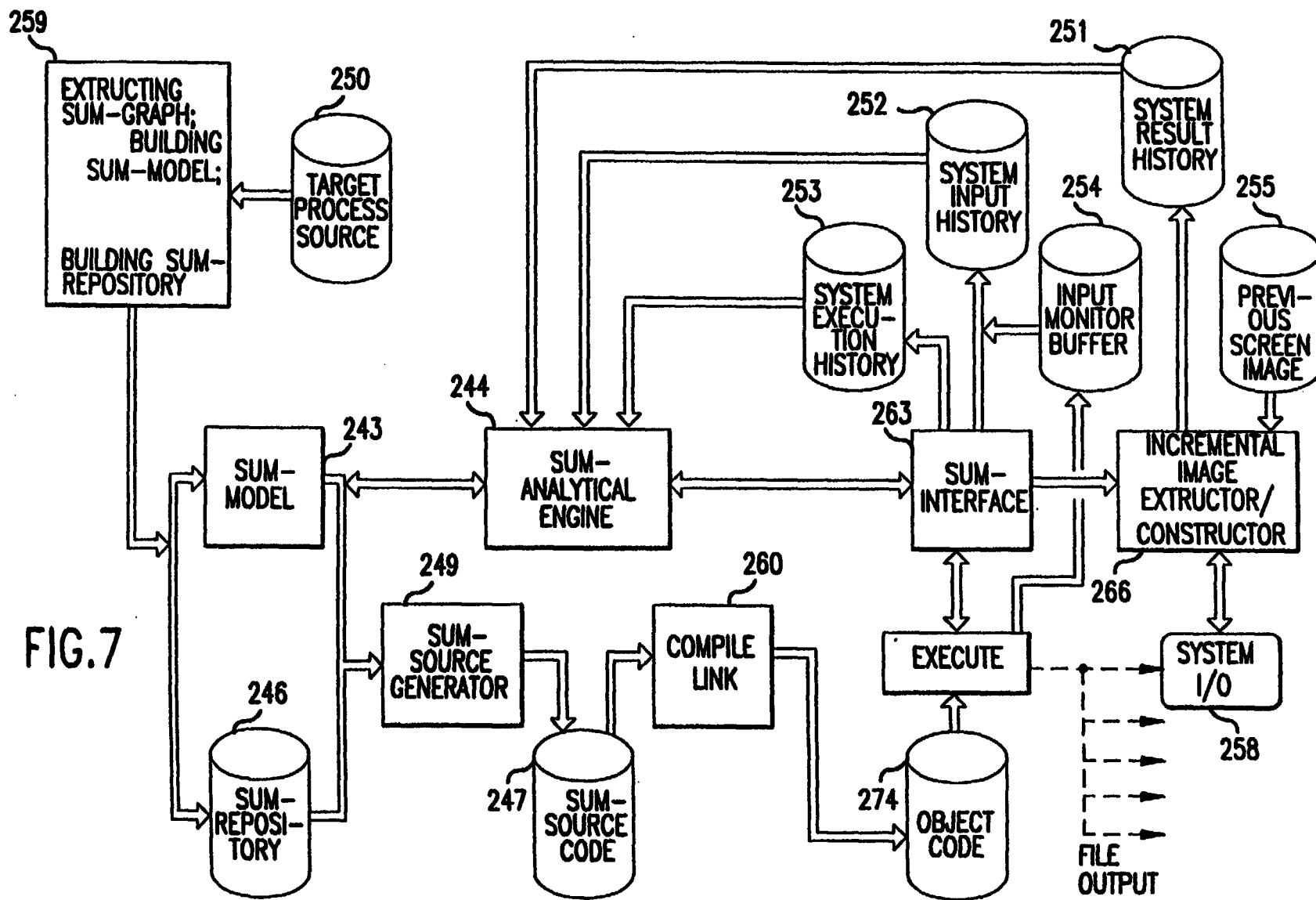
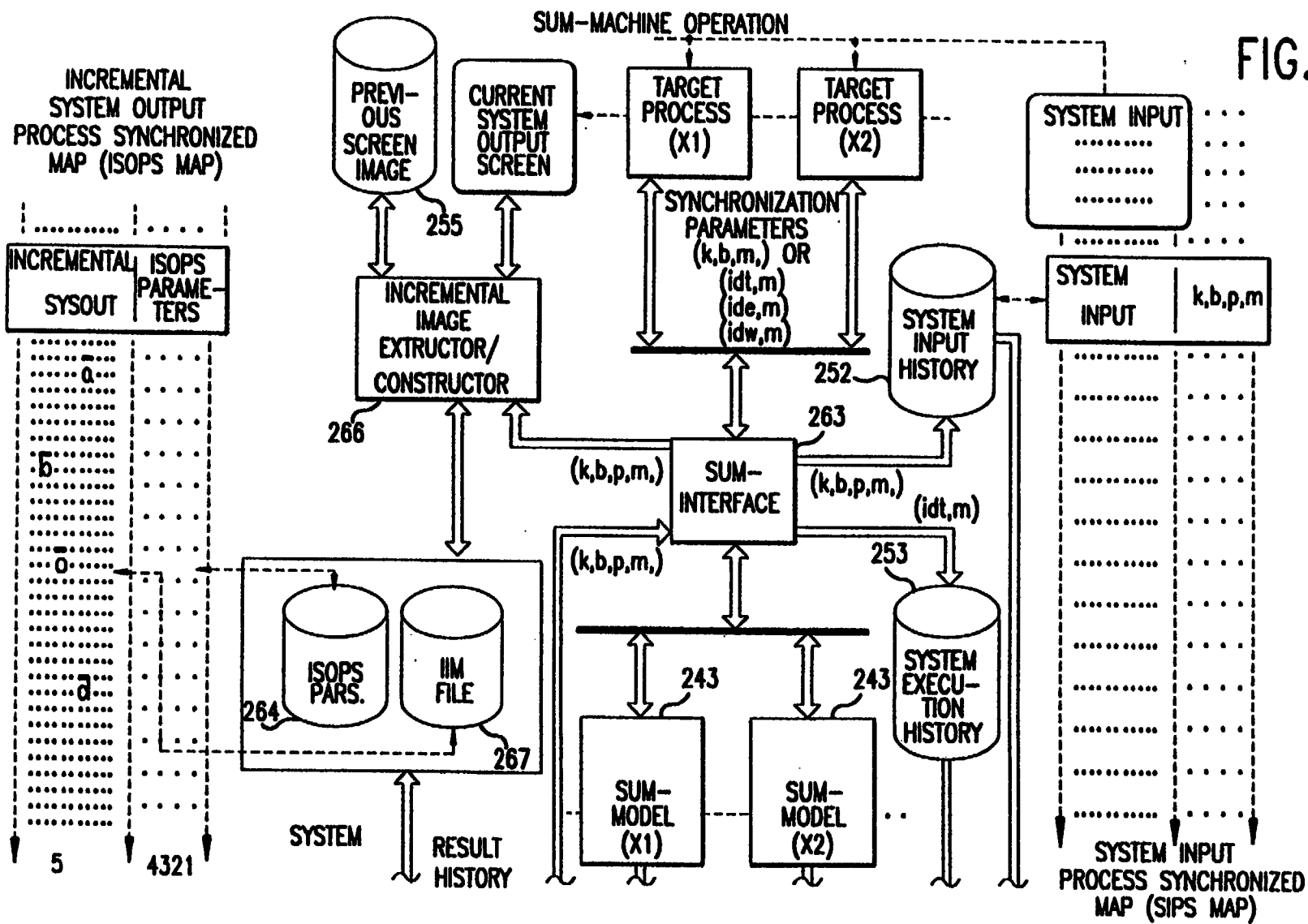


FIG.6





REPLACEMENT SHEET
30 OCT 1998

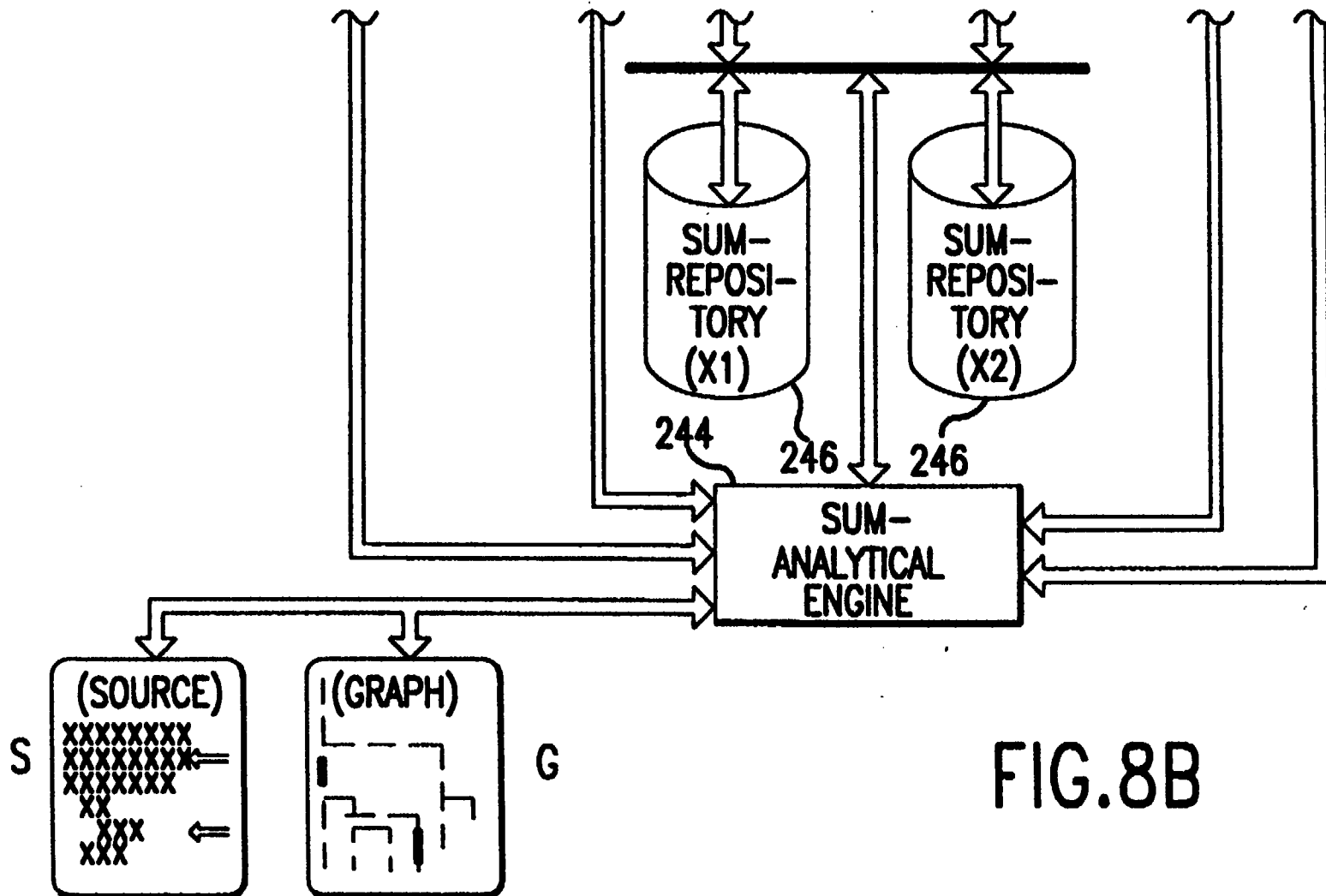
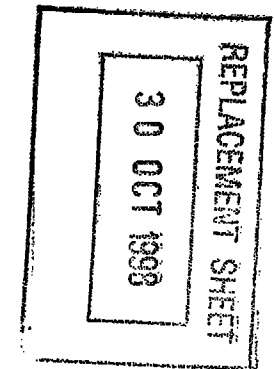
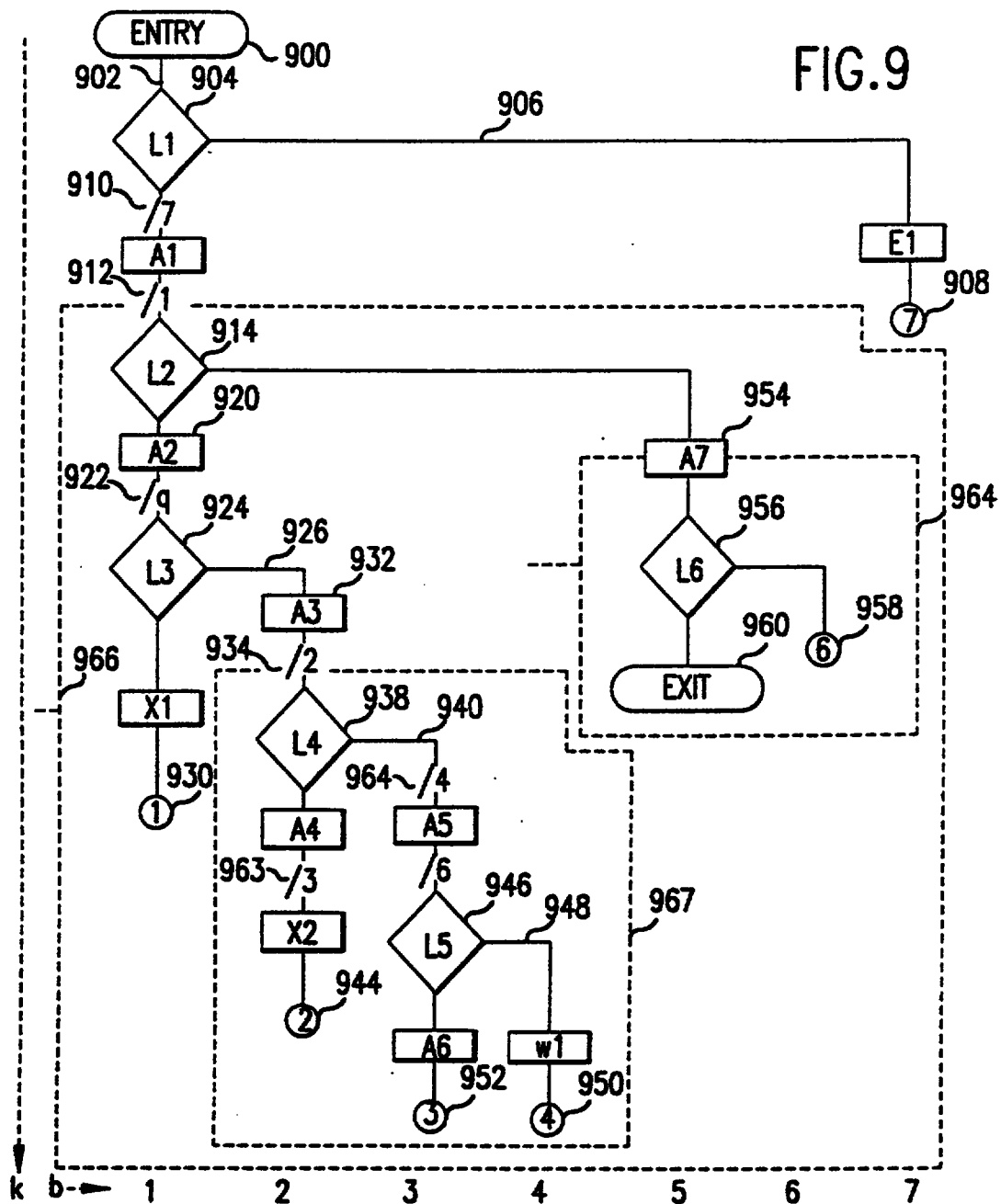


FIG.8B





EXIT ACC. IND.	ORIG	0	0	0	0	1	0	0
	END	1	0	0	0	1	0	1
PVP	ORIG rk	rk1	rk2	rk3	rk4	0	rk6	rk7
	END	rk1	rk2	rk2	rk2	0	rk2	rk7
	ORIG BA	10	1001	10010	10011	0	10011	10
	END	10	1001	1001	1001	0	1001	10
RBA		10	1001	10010	10011	0	10011	10
TBA		1000	10010	100110	100111	1010	1011	11

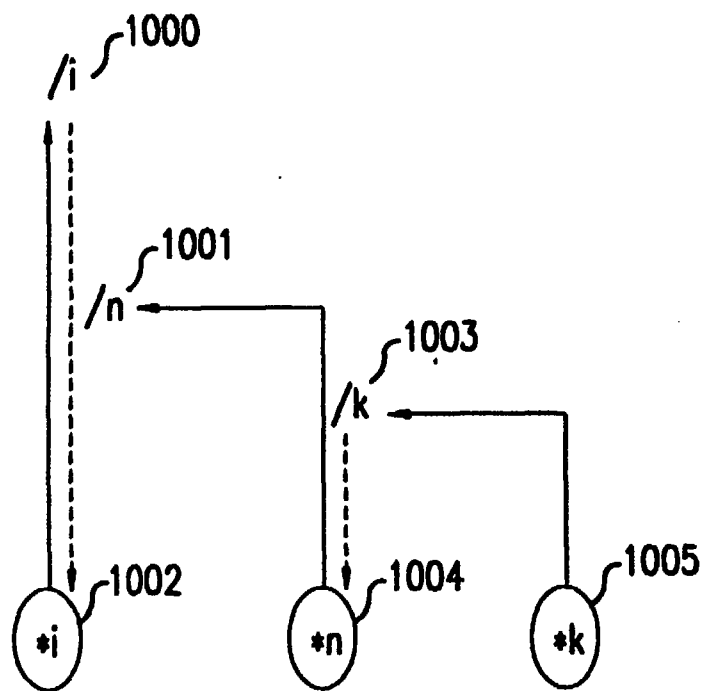


FIG. 10A

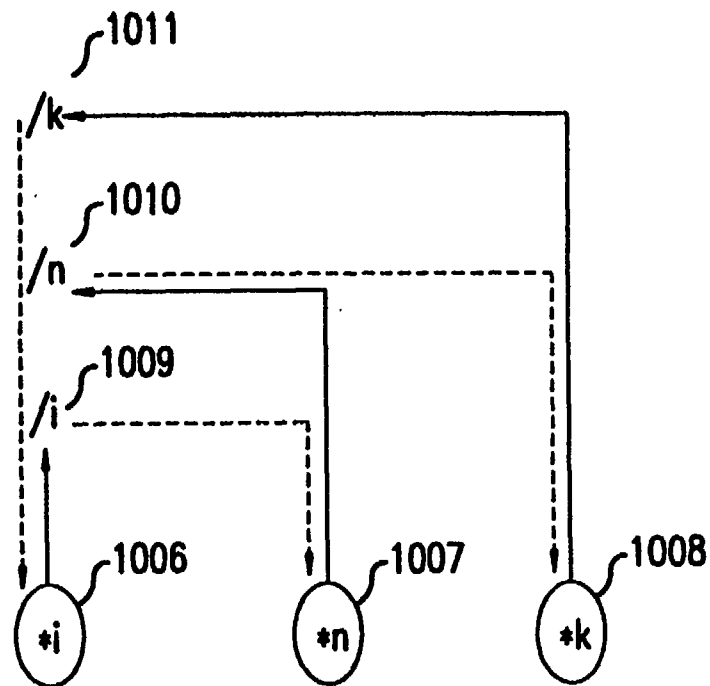


FIG. 10B

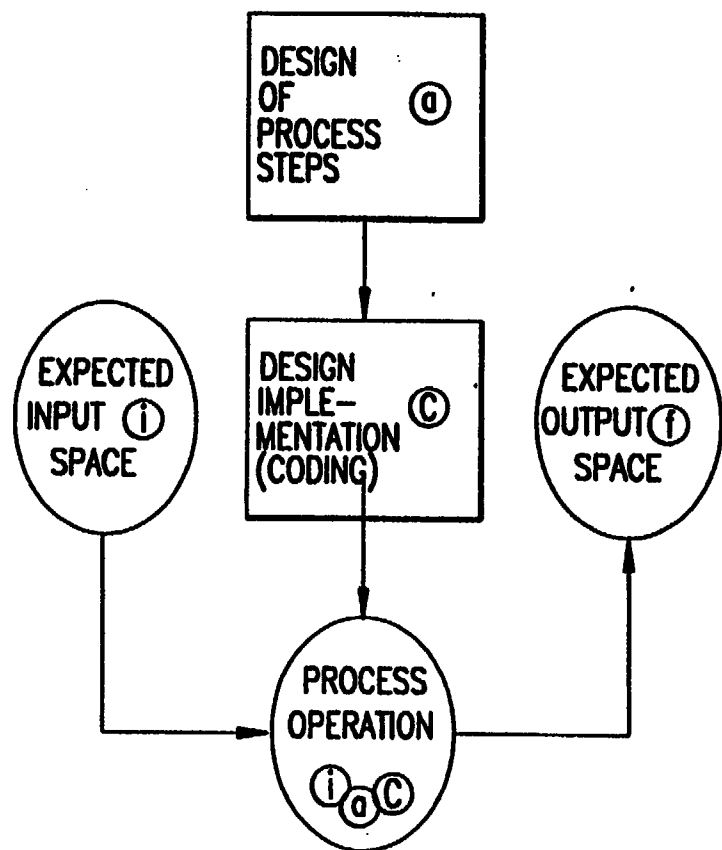


FIG. 11

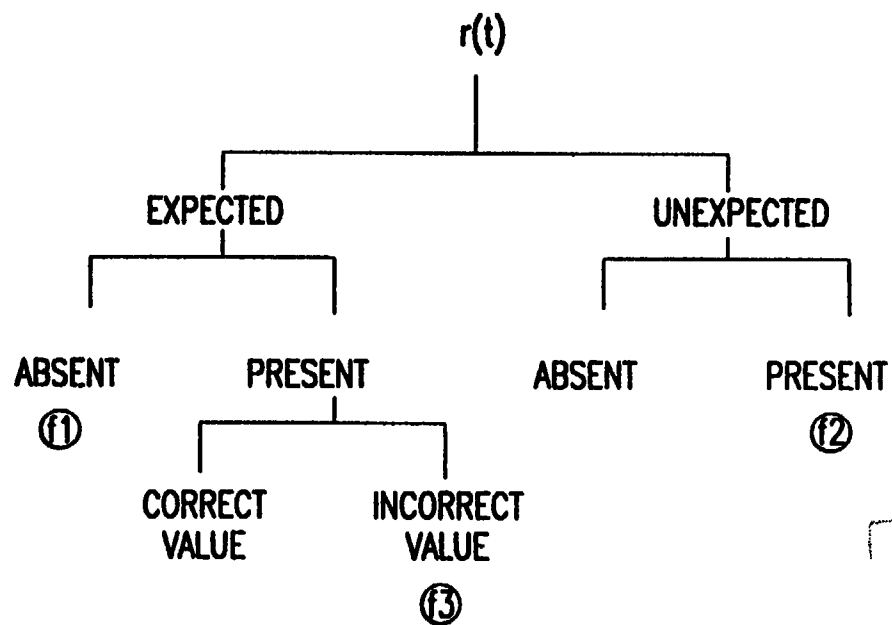
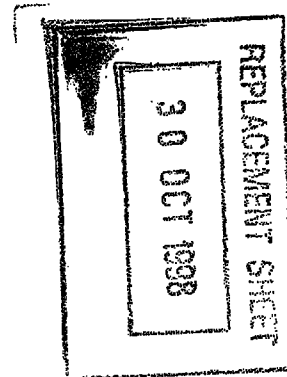
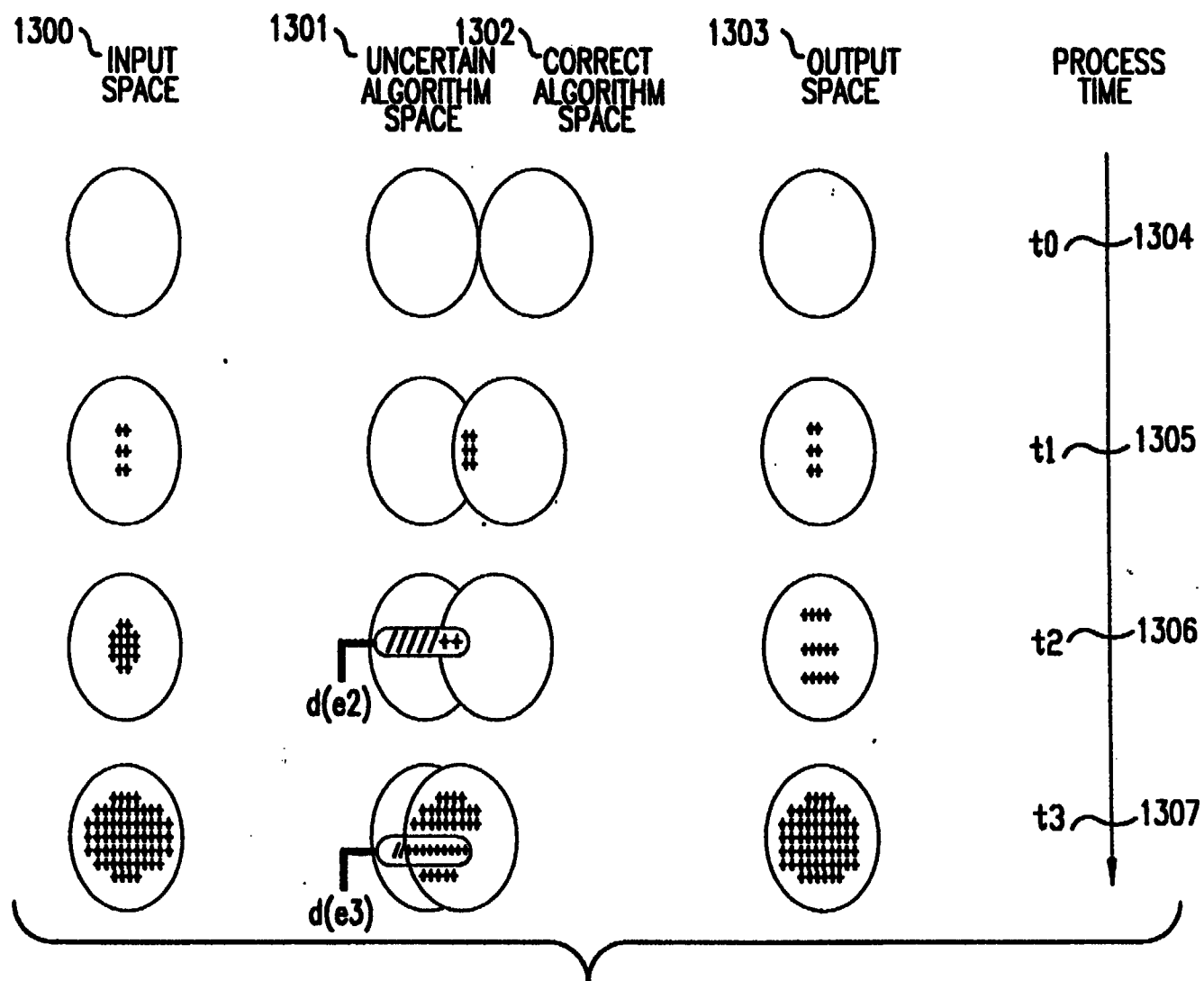


FIG. 12





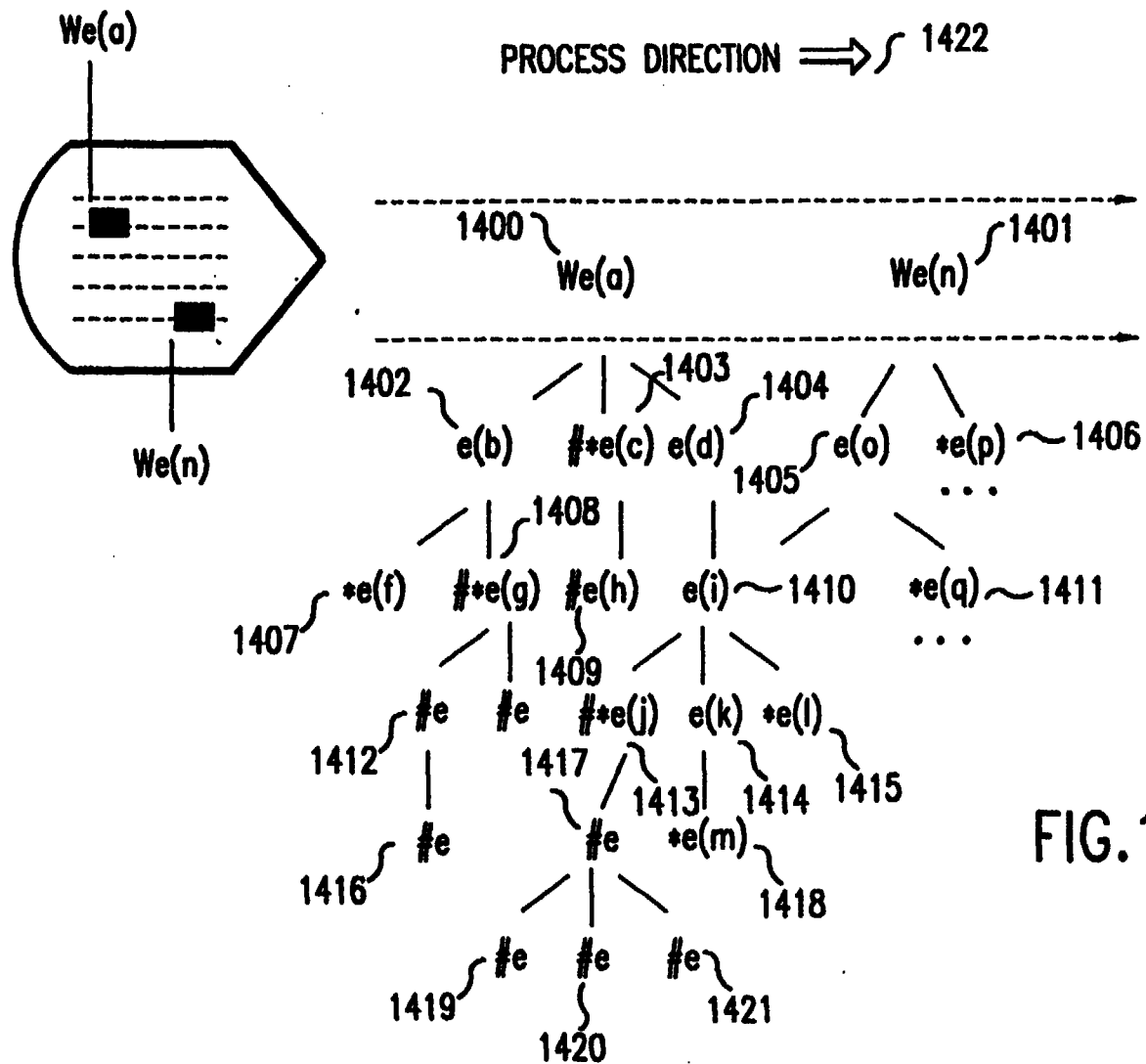
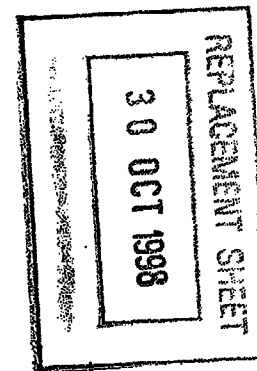


FIG.14



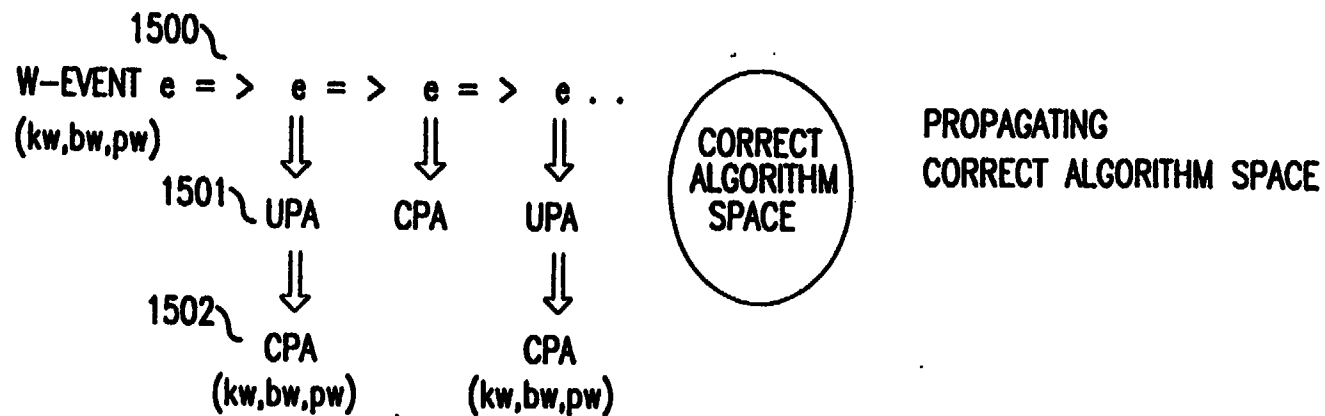


FIG.15

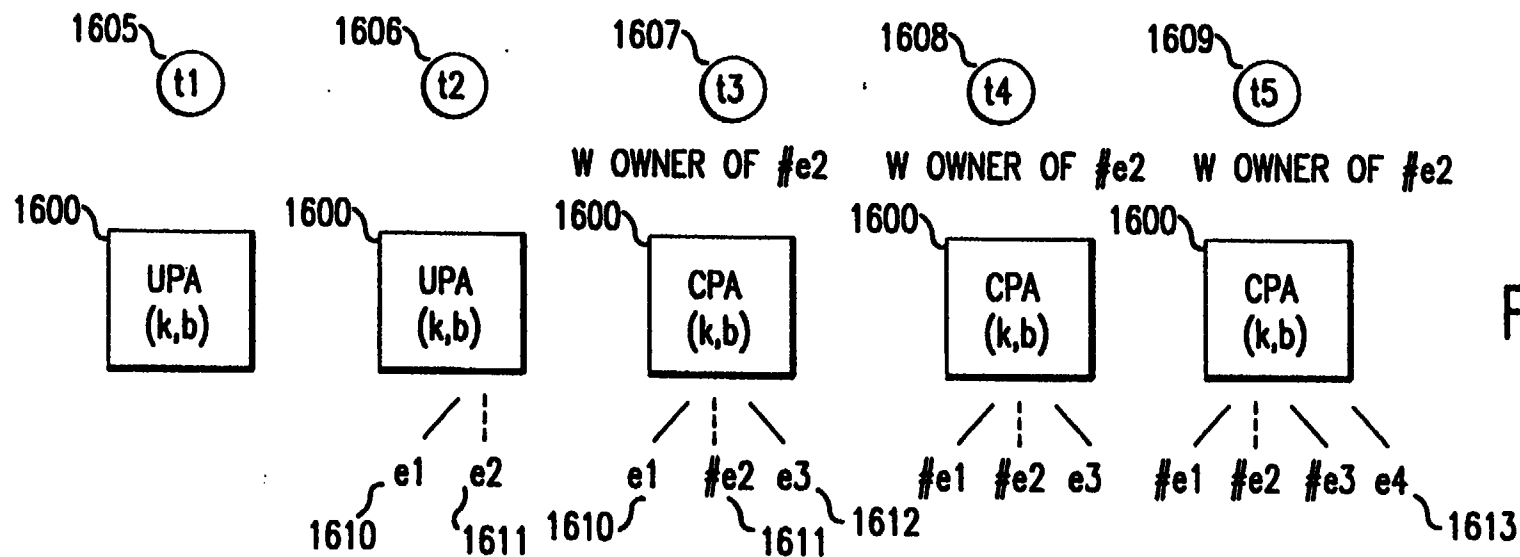
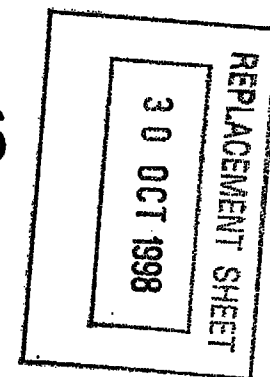


FIG.16



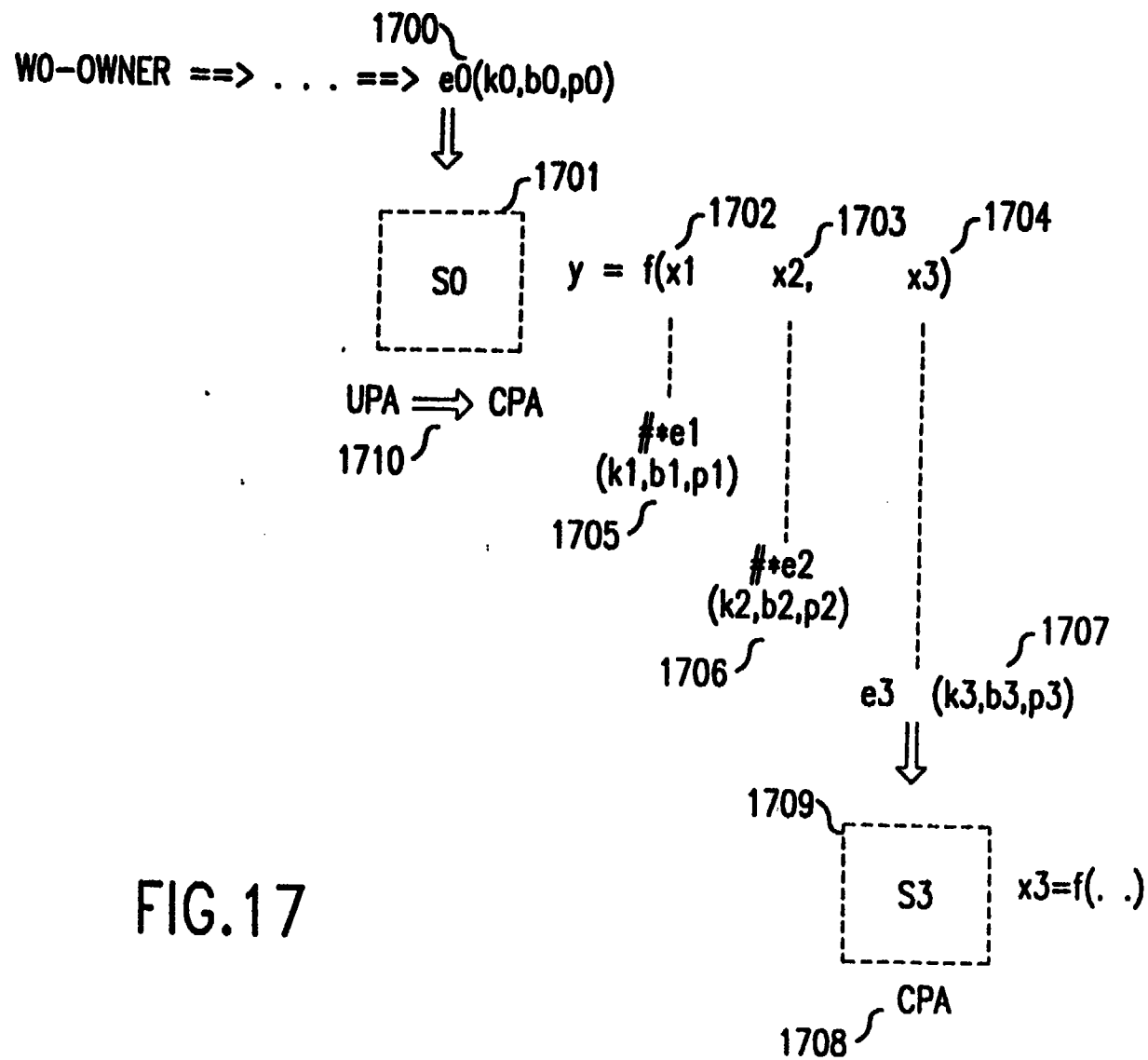


FIG.17

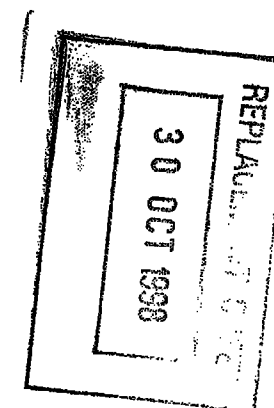


FIG. 18A

The diagram illustrates the control component of SUM object code, showing the flow of data and control between various modules and storage components. The main components and their interactions are as follows:

- CONTROL COMPONENT OF SUM OBJECT CODE**: This label points to the overall system architecture.
- SUM-MODEL (243)**: Receives input from the **SUM-REPOSITORY (246)** and sends data to the **SUM-ANALYTICAL ENGINE (244)** and the **SUM-DRIVER (269)**.
- SUM-REPOSITORY (246)**: A storage component that provides data to the **SUM-MODEL (243)** and the **TRANSLATE TO EXEC. COMPON. OF SUM O.C. (1807)**.
- SUM-ANALYTICAL ENGINE (244)**: Receives data from the **SUM-MODEL (243)** and the **SUM-REPOSITORY (246)**. It sends data to the **SUM-INTERFACE (263)** and the **SYSTEM EXECUTION HISTORY (253)**.
- SUM-INTERFACE (263)**: Acts as a bridge between the **SUM-ANALYTICAL ENGINE (244)** and the **SUM-DRIVER (269)**. It also interacts with the **SYSTEM INPUT HISTORY (252)** and the **INPUT MONITOR BUFFER (254)**.
- SUM-DRIVER (269)**: Receives data from the **SUM-MODEL (243)** and the **SUM-INTERFACE (263)**. It sends data to the **EXECUTE MACHINE CODE FRAGMENT (1806)** and the **SYSTEM EXECUTION HISTORY (253)**.
- EXECUTE MACHINE CODE FRAGMENT (1806)**: Receives data from the **SUM-DRIVER (269)** and the **TRANSLATE TO EXEC. COMPON. OF SUM O.C. (1807)**. It sends data to the **INCREMENTAL IMAGE EXTRACTOR/CONSTRUCTOR (266)** and the **SYSTEM I/O (258)**.
- TRANSLATE TO EXEC. COMPON. OF SUM O.C. (1807)**: Receives data from the **SUM-REPOSITORY (246)** and the **SUM-MODEL (243)**. It sends data to the **EXECUTE MACHINE CODE FRAGMENT (1806)** and the **MACHINE CODE FRAGMENTS (248)**.
- MACHINE CODE FRAGMENTS (248)**: A storage component that provides data to the **EXECUTE MACHINE CODE FRAGMENT (1806)**.
- INCREMENTAL IMAGE EXTRACTOR/CONSTRUCTOR (266)**: Receives data from the **EXECUTE MACHINE CODE FRAGMENT (1806)** and the **SYSTEM I/O (258)**. It sends data to the **SYSTEM RESULT HISTORY (251)** and the **PREVIOUS SCREEN IMAGE (255)**.
- SYSTEM I/O (258)**: A component that handles input and output, sending data to the **EXECUTE MACHINE CODE FRAGMENT (1806)** and the **INCREMENTAL IMAGE EXTRACTOR/CONSTRUCTOR (266)**.
- SYSTEM EXECUTION HISTORY (253)**: A storage component that receives data from the **SUM-ANALYTICAL ENGINE (244)** and the **SUM-DRIVER (269)**.
- SYSTEM INPUT HISTORY (252)**: A storage component that receives data from the **SUM-INTERFACE (263)**.
- INPUT MONITOR BUFFER (254)**: A storage component that receives data from the **SUM-INTERFACE (263)** and the **SYSTEM I/O (258)**.
- SYSTEM RESULT HISTORY (251)**: A storage component that receives data from the **INCREMENTAL IMAGE EXTRACTOR/CONSTRUCTOR (266)**.
- PREVIOUS SCREEN IMAGE (255)**: A storage component that receives data from the **INCREMENTAL IMAGE EXTRACTOR/CONSTRUCTOR (266)**.

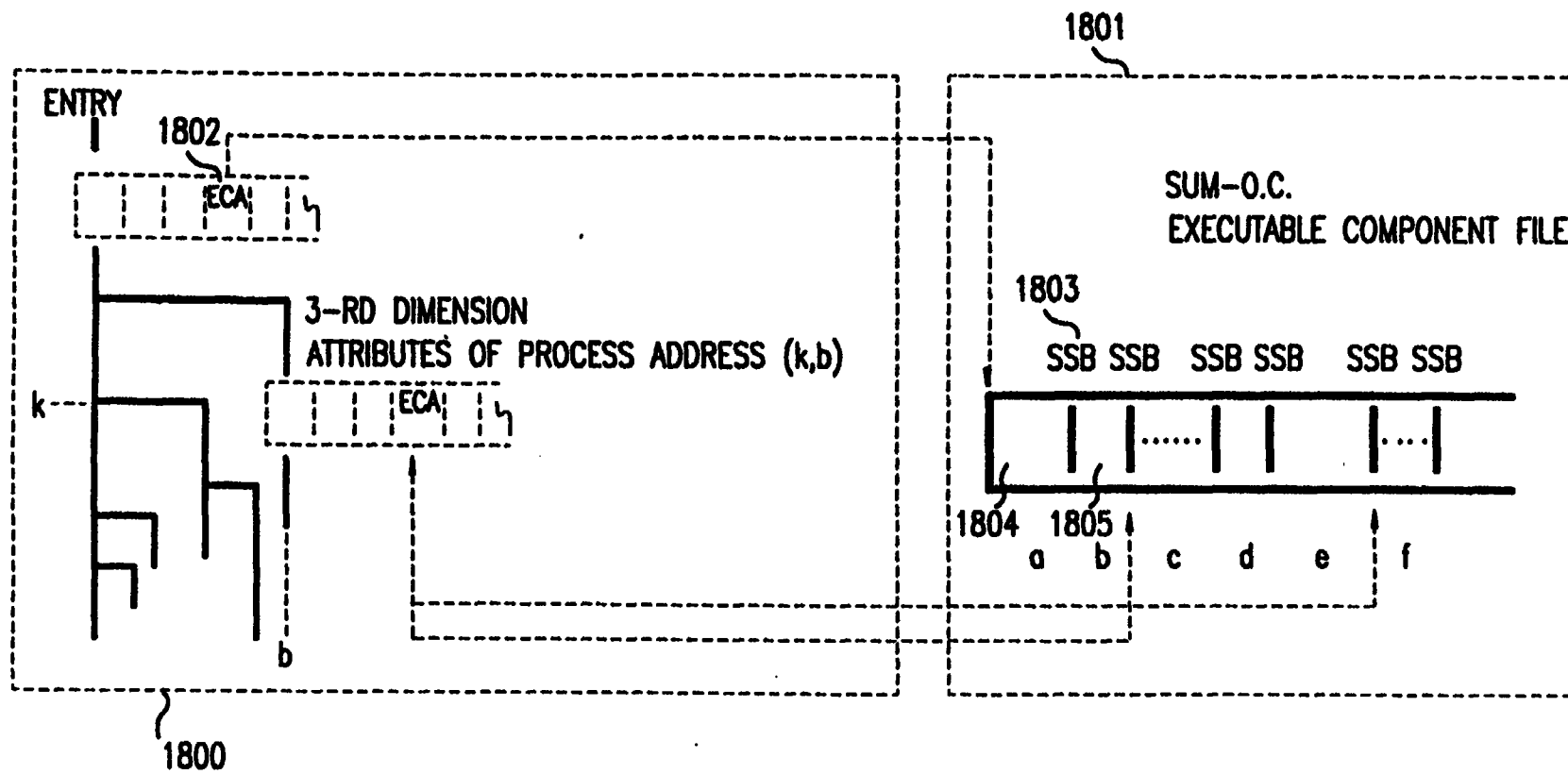


FIG. 18B

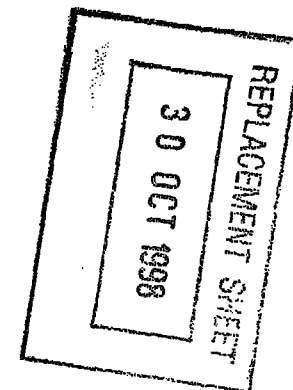
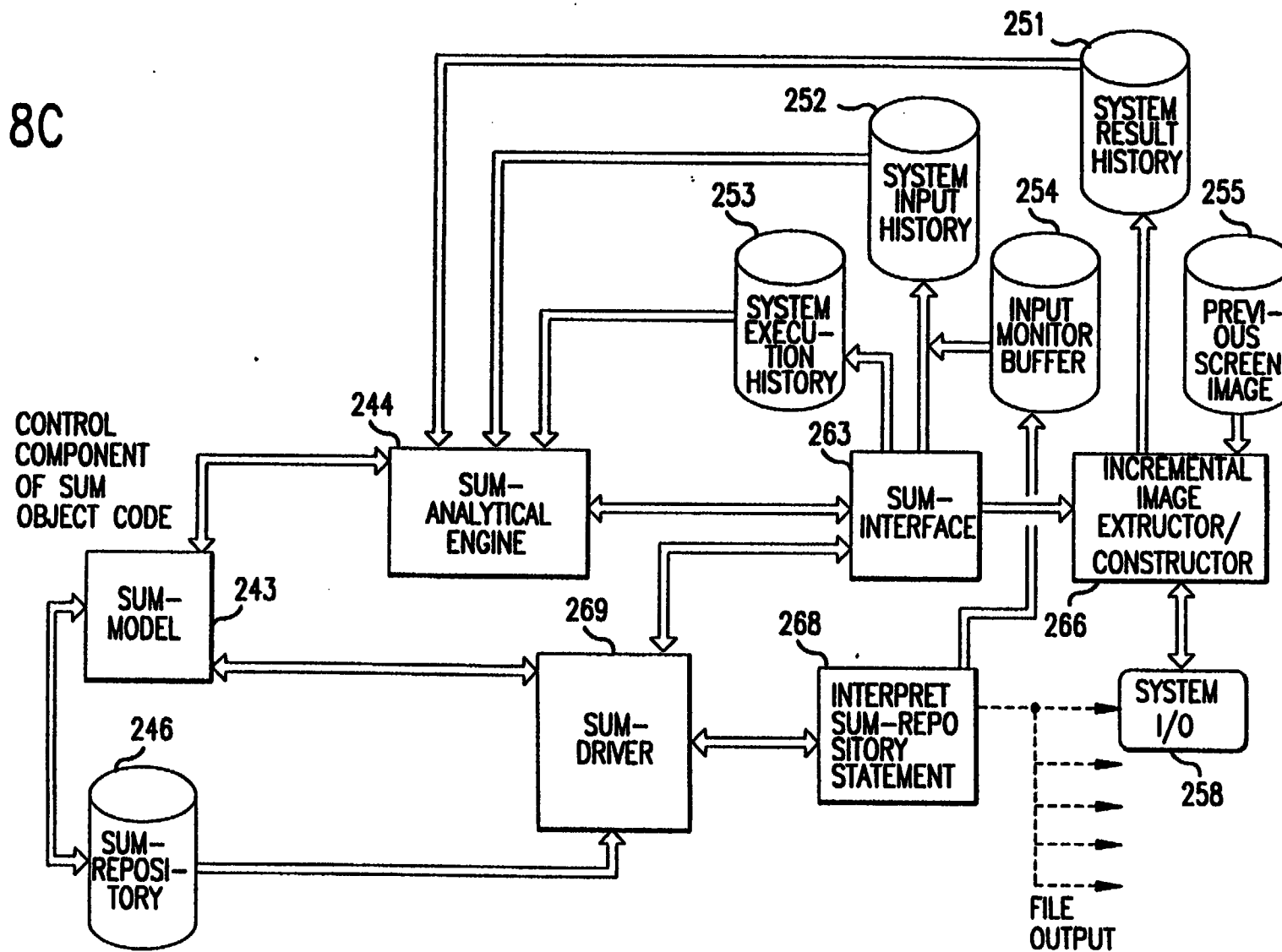


FIG.18C



30 OCT 1998

- 20 -

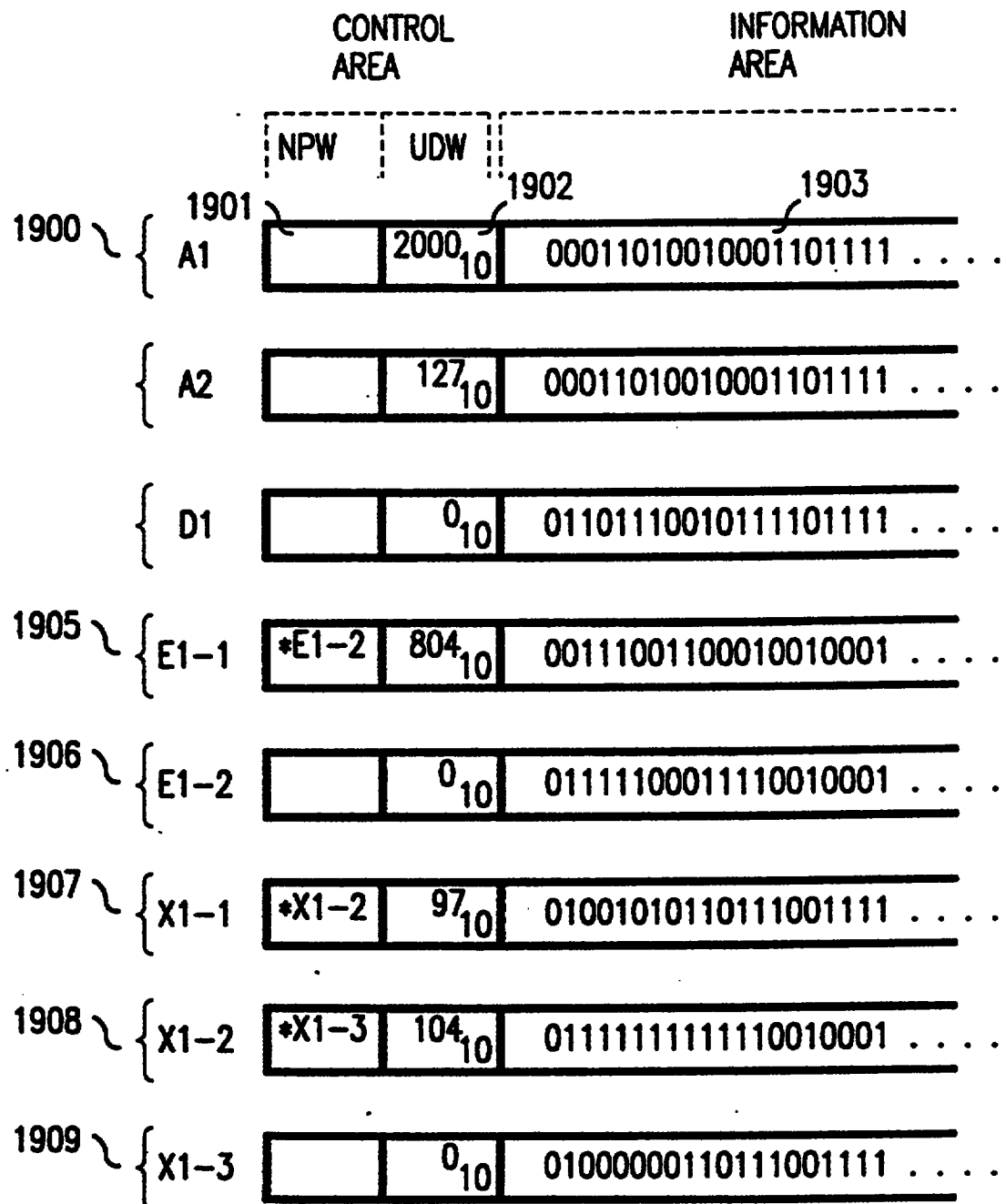


FIG. 19

REPLACEMENT SHEET

30 OCT 1998

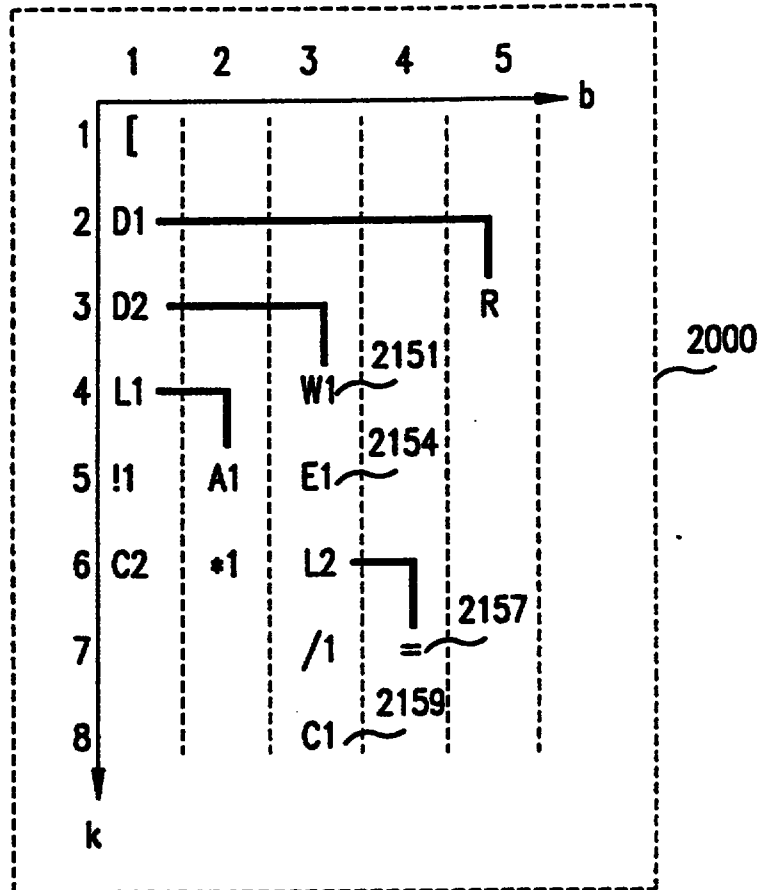


FIG.20A

```

INTEGER IN(3)
CHARACTER IA(3), IALF(10), CH
DATA IALF/'0','1','2','3','4','5','6','7','8','9'/
001  FORMAT(3X,'IN(I)=' ,I3,'DOESNOT MATCH INTEGER FROM 0 TO 9',
      C' TYPE "Y" TO CONTINUE')
002  FORMAT(A)
D1:   I=1,3
D2:   J=1,10
L1:   IF(IN(I).EQ.J-1)
A1:   IA(I)=IALF(J)
W1:   WRITE(*,001)IN(I)
E1:   READ(*,002)CH
L2:   IF(CH.NE.'Y'.AND.CH.NE.'y')
    
```

FIG.20B

2100

```

SUBROUTINE I2A(IN,IA)
INTEGER IN(3)
CHARACTER IA(3),IALF(10)
DATA IALF/'0','1','2','3','4','5','6','7','8','9'/
DO 10 I=1,3
  DO 55 J=1,10
    IF(IN(I).EQ.J-1)THEN
      IA(I)=IALF(J)
      GO TO 10
    ENDIF
55  CONTINUE
  WRITE(*,100)IN(I) ~2151
100  FORMAT(3X,'IN(I)=' ,I3,'DOES NOT MATCH INTEGER',
C    'FROM 0 TO 9 TYPE "Y" TO CONTINUE')
  READ(*,20)CH ~2154
20  FORMAT(A)
  IF(CH.NE.'Y'.AND.CH.NE.'y')~2157
10  CONTINUE ~2159
  RETURN ~2161
END

```

FIG.21A

2150

```

SUBROUTINE I2AXXX(IN,IA)
INTEGER IN (3)
CHARACTER IA(3),IALF(10)
DATA IALF/'0','1','2','3','4','5','6','7','8','9'/
001  FORMAT(3X,'IN(I)=' ,I3,'DOES NOT MATCH INTEGER',
C    'FROM 0 TO 9 TYPE "Y" TO CONTINUE')
002  FORMAT(A)
  DO 3001 I=1,3
    DO 3002 J=1,10
      IF(IN(I).EQ.J-1)THEN
        IA(I)=IALF(J)
        0  CALL SUMI(14,2)
        GO TO 23001
      ENDIF
      0  CALL SUMI (6,2)
    CONTINUE
    3002  WRITE(*,001)IN(I) ~2151
    0  CALL SUMI(20,2) ~2152
    0  CALL SUMI(21,2) ~2153
    READ(*,002)CH ~2154
    0  WRITE(M,002)CH ~2155
    IF(CH.NE.'Y'.AND.CH.NE.'y')THEN
      0  CALL SUMI(31,2) ~2156
      STOP ~2157
    ENDIF
    230010  CALL SUMI(24,2) ~2158
    3001  CONTINUE ~2159
    0CALL SUMI(35,2)
    RETURN
    END

```

FIG.21B

30 OCT 1998

- 23 -

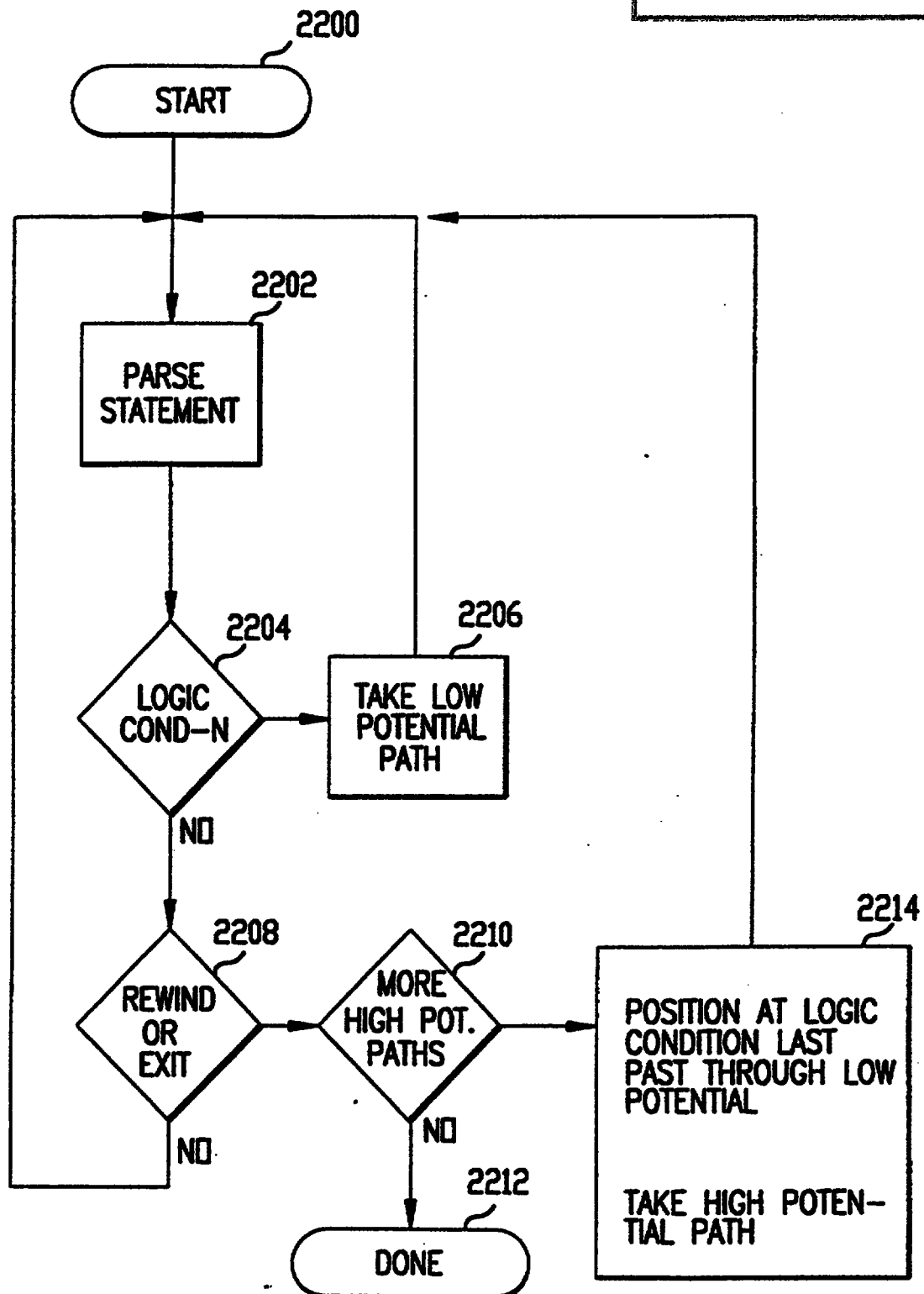


FIG.22

30 OCT 1998

- 24 -

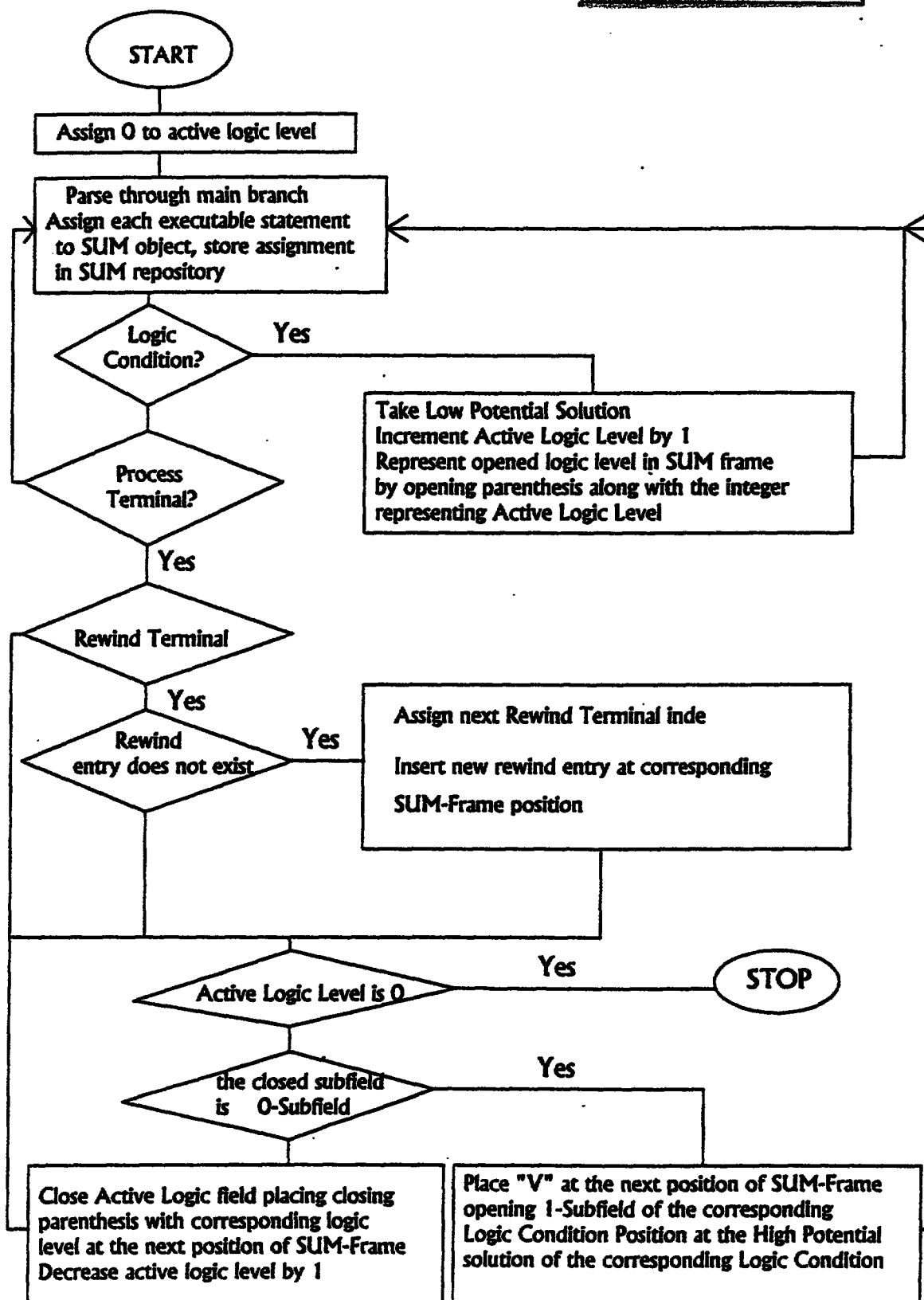


FIG 23

REPLACEMENT SHEET
30 OCT 1998

ANALYZER

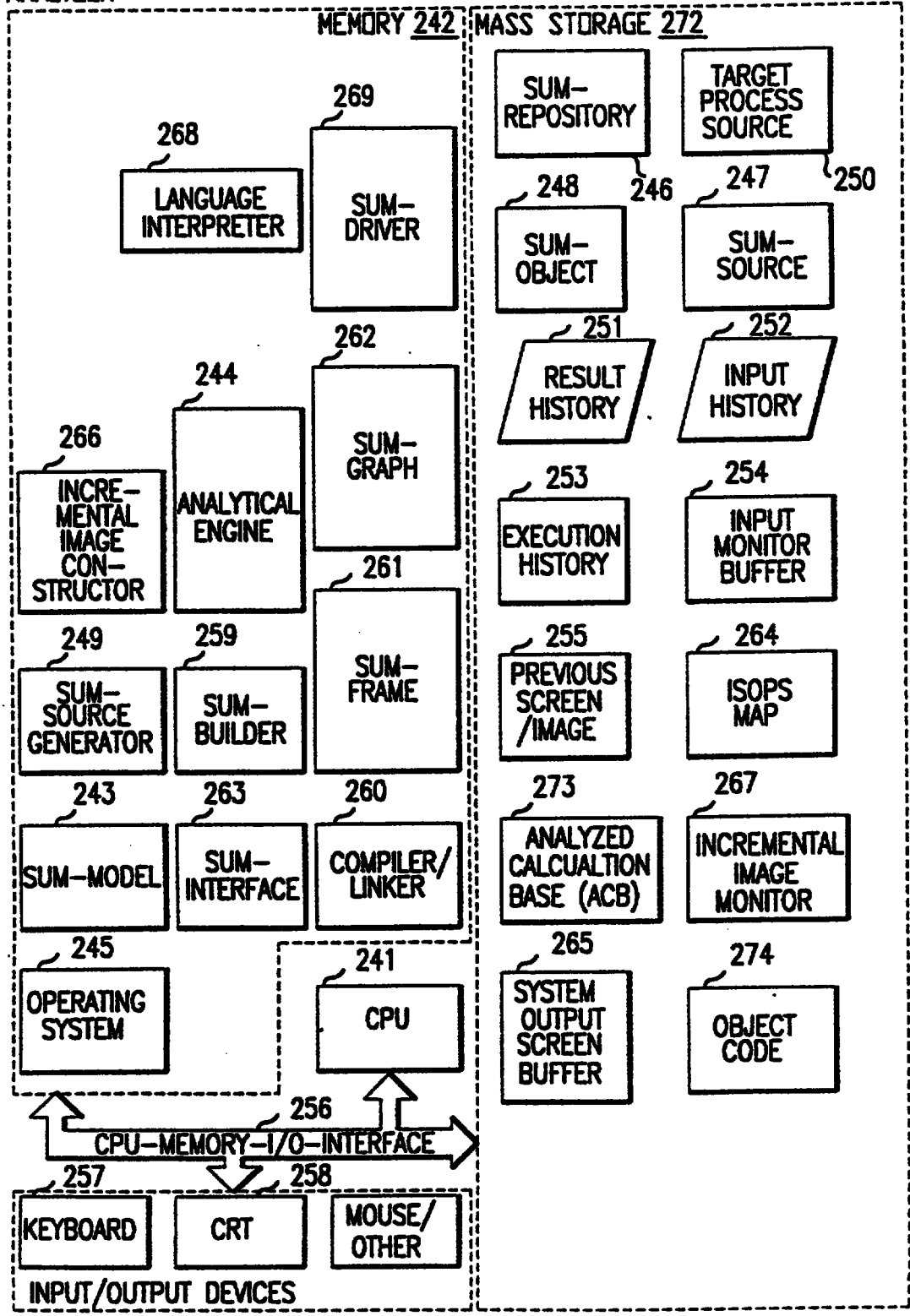
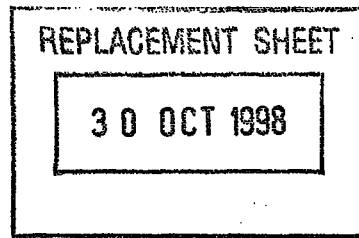


FIG.24



$W(k,b,p)$

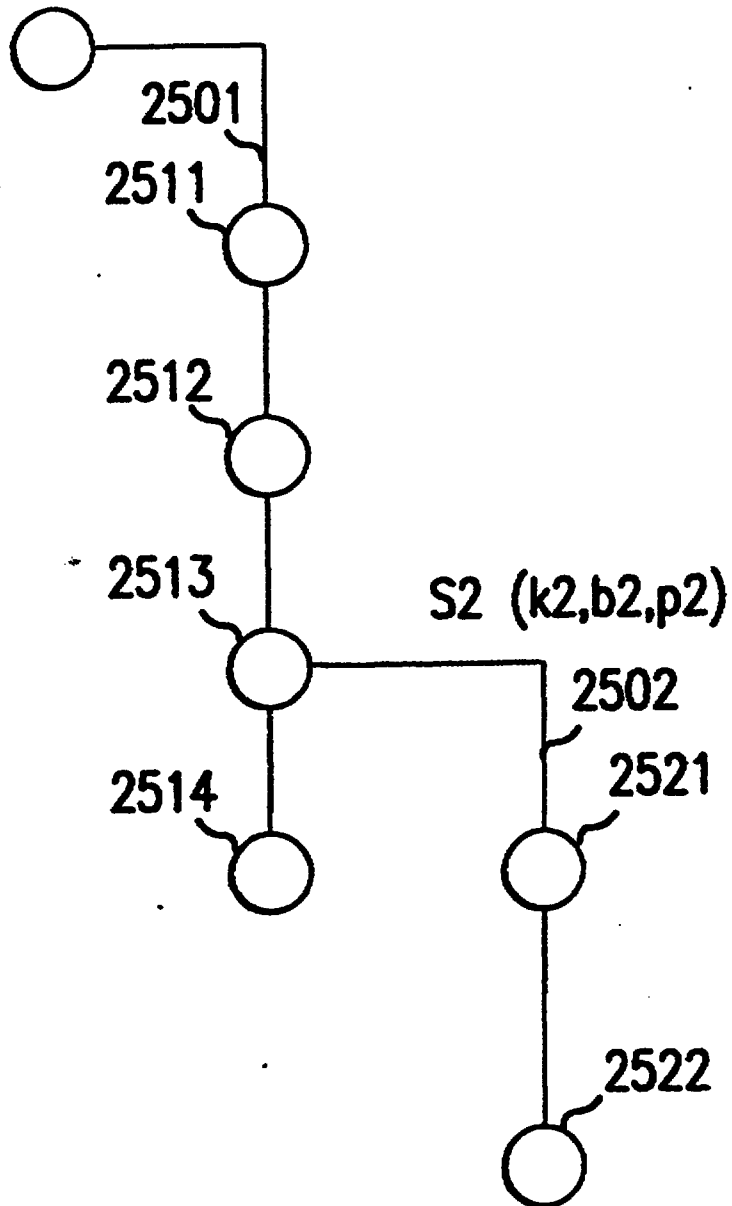
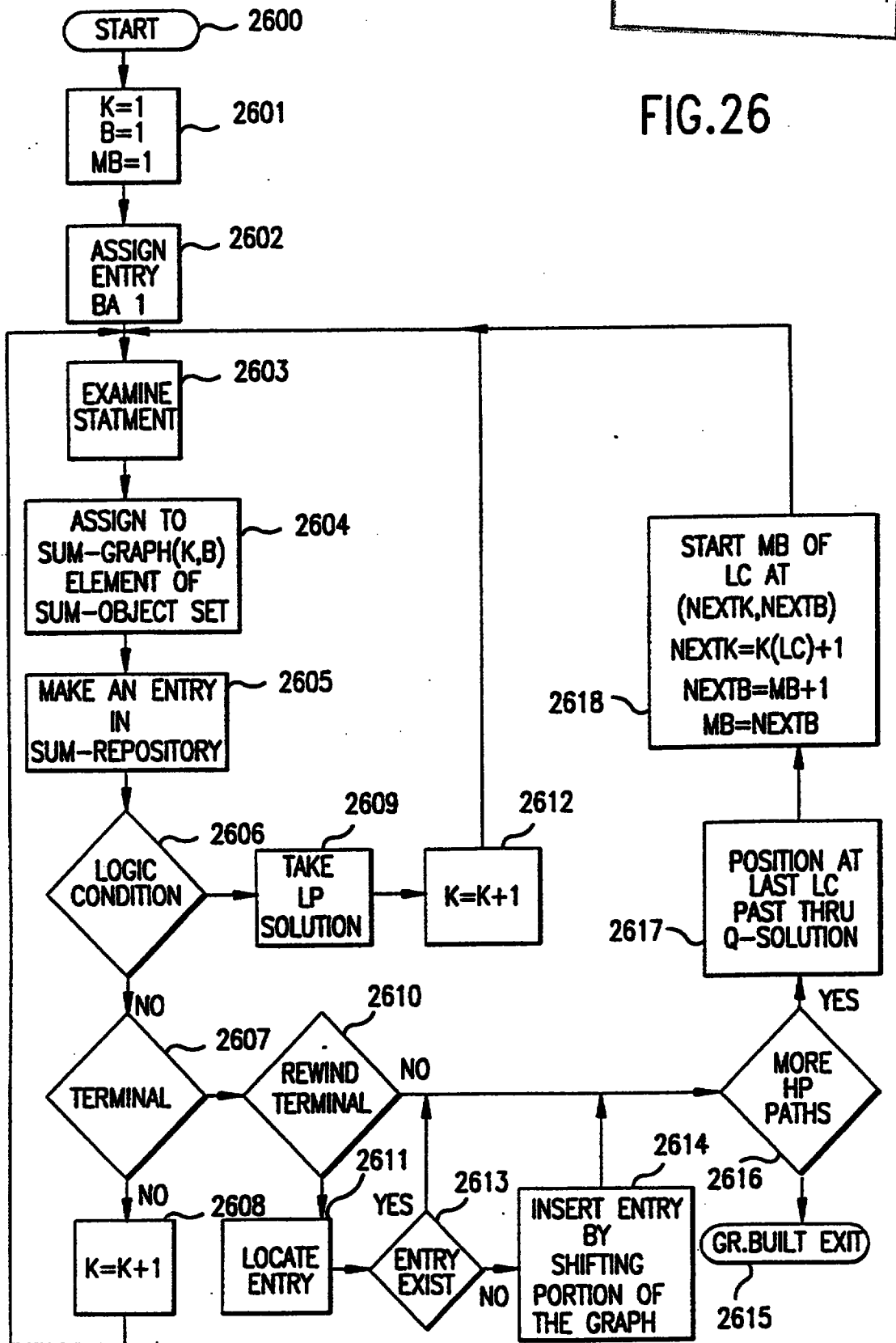


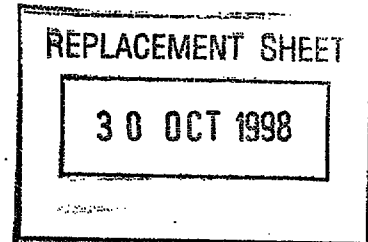
FIG.25

30 OCT 1998

- 27 -

FIG.26





ISOPS map sample format:

1.	2.	3.	4.	5.
W-	screen	screen	displ.	incremental
event-	position	area	in image	image
		changed	monitor	monitor
			file	file

_____ISOPS parameters file_____/_IIM file_/_/

{k,b,p,m}	{sl,sc}	{l1,c1,l2,c2}	DDDDDD	'oooooooooooooooooooo'
{k,b,p,m}	{sl,sc}	{l1,c1,l2,c2}	DDDDDD	'ooooooo'
{k,b,p,m}	{sl,sc}	{l1,c1,l2,c2}	DDDDDD	'oooooooooooooooooooo'
{k,b,p,m}	{sl,sc}	{l1,c1,l2,c2}	DDDDDD	'ooooooo'
{k,b,p,m}	{sl,sc}	{l1,c1,l2,c2}	DDDDDD	'oooooooooooo'

**FIGURE
27**