DISTRIBUTED INTELLIGENT ROBOTICS:

RESEARCH & DEVELOPMENT IN FAULT-TOLERANT CONTROL AND SIZE/POSITION IDENTIFICATION

A thesis presented in partial

fulfilment of the requirements for the degree of

MASTER OF ENGINEERING

in

COMPUTER SYSTEMS ENGINEERING

at Massey University, Turitea Campus,

Palmerston North,

New Zealand

KARTHIKEYA KRISHNA SUBRAMANIAM

2002

Abstract

This thesis presents research conducted on aspects of intelligent robotic systems. In the past two decades, robotics has become one of the most rapidly expanding and developing fields of science. Robotics can be considered as the science of using artificial intelligence in the physical world. Many areas of study exist in robotics. Among these, two fields that are of paramount importance in real world applications are fault tolerance, and sensory systems. Fault tolerance is necessary since a robot in the real world could encounter internal faults, and may also have to continue functioning under adverse conditions. Sensory mechanisms are essential since a robot will possess little intelligence if it does not have methods of acquiring information about its environment. Both these fields are researched in this thesis. In particular, emphasis is placed on distributed intelligent autonomous systems. Experiments and simulations have been conducted to investigate design for fault tolerance. A suitable platform was also chosen for an implementation of a visual system, as an example of a working sensory mechanism.

Acknowledgements

I would like to thank my supervisors Associate Professor Serge Demidenko, and Dr. Chris Messom for their help, patience, and guidance, and for the high standard of achievement they encouraged. I have been very fortunate to have as my supervisors two people who possess so great a knowledge of their subject areas, and who were always willing to spend the time to impart that knowledge. I would also like to thank them for their friendship.

I would like to thank my parents and my sister for always encouraging me to further my studies. I would also like to thank Vani for the countless hours she spent "keeping me company", while I worked.

Finally, I would like to thank the ASIA 2000 foundation for their continuing efforts in providing educational opportunities and experiences for students such as myself.

ERRATA

Distributed Intelligent Robotics:

Research and Development in Fault-Tolerant Control and Size/Position Identification

K.K. Subramaniam

Chapter 2.1.3.5, pg.23
Errata

On page 23, the notation $Q_{n,j}$ is used to describe a vertex, but the notation is not explained until page 24. $Q_{n,j}$ denotes the j^{th} vertex in the n^{th} section of a flow graph that has been divided into sections, in order to reduce complexity.

Chapter 2.2.4, pg.38
Errata

The following section is to be inserted at the end of page 38, to further explain the flow graph example:

"...erroneous transitions in dashed lines.

In considering the probabilities of erroneous transitions between states, it is important to recognize that the graph in figure 2-19 is an example of the erroneous transitions possible *only in the specific error situation stated.* Since the grey states represent unreachable states, the probabilities for erroneous transitions *from the preceding states* have rows that sum to one, since an erroneous transition must occur. This is coincidental in this case, and it is not necessary that the rows sum to one. The probabilities listed are the probabilities that an erroneous transition will occur in the specific example, and not the total probabilities for all possible erroneous transitions.

Figure 2-19..."

Chapter 2.3.2.2, pg.47
Errata

In Figure 2-23, a dashed arrow is used to show the erroneous transitions in the graph. Erroneous transitions are possible from vertices Q_1 to Q_1 and Q_1 to Q_3 (not Q_1 to Q_2, as indicated).

Chapter 2.3.2.2, pg.48
Errata

The third element of the G matrix is element $p_{1,3}$, and not $p_{3,1}$.

Chapter 2.3.2.3, pg.49
Errata

The references in the first paragraph are to the matrix M_{Qi}, and thus the first paragraph reads:

Here, M_{Qi} is the fault behaviour description of vertex Q_i. M is a $K \times N$ matrix, where K is the total number of faults modelled, and N is the total number of vertices in the flow-graph. Thus, element (2,1) of the matrix denotes the probability of an erroneous transition from Q_i to Q_1, in the presence of *fault type 2* – in a real implementation, this would be a particular type of fault.

Chapter 2.4.2, pg.54
Errata

Equation 2-23 written as:

$$\psi(\overline{x)^{-1}})p(x)=q(x)rem\ s(x)$$

is in fact:

$$\psi(x))\overline{p(x)}=q(x)rem\ s(x)$$

Chapter 2.4.3, pg.56, 57
Errata

Two figures are labelled as 'Figure 2-35' on pages 56 and 57. To correct this, the figure on page 57 has to be labelled as Figure 2-35b, in which case the last sentence on page 56 reads:

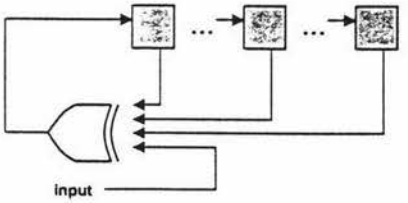
'In other words, the situation in figure 2-35b is possible.'

Also, the fourth sentence in Chapter 2.4.4 on page 57 must then read:

'Figure 2-36 below illustrates the insertion of an erroneous bit between the first and second bits of an 8-bit key.'

Chapter 2.4.4, pg.58
Errata

The shift register outputs are combined with an XOR gate, and thus the diagram is equivalent to:



Chapter 2.4.5, pg.63
Errata

The following statement is to be inserted after the first paragraph of page 63:

"...odd numbers of errors are detected.

Syndrome coding refers to an error checking method in which the error is detected by counting the occurrences of '1's and '0's in the resultant key. The number of occurrences can be compared with expected values, and there are many situations in which this checking scheme produces high coverage statistics.

When syndrome coding is used for..."

Contents

Figures

1 Distributed Intelligent Robotics

1.1 Introduction

Since the 1980s, intelligent robotics has been among the fastest developing and expanding areas of science and technology. Robotics can loosely be characterised as the area where artificial intelligence meets the real world [11].

As research in robotics progresses, increasing importance is being placed on the development of distributed systems [1,2,5]. In many applications, rather than using one robot, a group of autonomous intelligent units can be far more useful. The robots may be autonomous, in that they are all mobile individual units capable of performing tasks themselves. However, their advantage lies in their intelligence, in that they can be used to co-operate with each other, and achieve a set task together. In essence, they are a team of robots. The field of distributed intelligent robotics is so new, that it can be considered to be in its infancy,

compared to other fields of electronics. As such, it is ideal for research since its potential is virtually limitless, and its scope for application vast.

Distributed robotics can be divided into several target application areas: biologically inspired systems, communication, architectures, mapping/ exploration, object transport and manipulation, motion co-ordination, reconfigurable robots, and learning. Each of these fields will be described here briefly, but are dealt with in more detail in [3].

Biological inspirations for mobile robotics are often based on the behavioural patterns of social animals and insects. The most commonly used patterns are the control rules of societies of ants, bees, and birds. Multi-robot teams have been designed to flock, disperse, aggregate, forage, and follow trails. More recently, co-operation in higher animals such as wolves has led to developments in co-operative control, and more complex social behaviour (such as humans playing games) has been studied and implemented in domains such as robot soccer [6-8].

Communication within robot teams is essential, since the advantage of distributed systems lies in the ability of the robots to communicate and co-operate with each other. As such, this is a field that has drawn many researchers. More recent work in communication has focussed on representation of languages, and the correlation of these languages to real-world environments. In addition, research has also extended to fault-tolerance in multi-robot communications, in situations such as setting up and maintaining reliable networks for information exchange.

A great deal of research in robotics has focussed on the development of standardised architectures. The main issue in this facet of research is whether customised architectures should be used for each application, or whether it would be practicable to have a general all-encompassing architecture.

Mapping and exploration have long been ideal target applications for robotics, especially since the dawn of the space age. Although many robots have been sent into unfamiliar regions such as space and deep-sea sites, and used successfully, the question now arises whether there is any advantage in employing robot teams instead. The majority of research has also been in 2-D mapping, whereas real-world environments are now available where robots must be sent for exploration – perhaps the most significant of these is exploration of the surface of Mars.

Object transportation and manipulation usually involves moving objects either by pushing or carrying them, although the latter method is quite difficult since it requires the robots to grip and lift the objects. This field is quite challenging, as the robots need to move in a co-ordinated fashion in order to transport their object to the goal. An additional challenge is to perform these tasks on uneven surfaces, as most of the trials done so far have been on flat terrain.

Motion co-ordination has been a popular topic for research. Commonly studied domains include path-planning, traffic control, formation generation, and formation maintaining. Most of these topics have been researched with respect to 2-D surfaces. Additional fields of research are target searching and target tracking, both of which are fundamental issues in the functioning of systems such as the robot soccer competitions.

Reconfigurable robotics has advanced in leaps and bounds recently, with many successful implementations. The aim of reconfigurable robots is to achieve some advantage from shape, allowing the individual modules that make up the robot to re-connect themselves in the most useful structure for the current environment. Theoretically, these systems can be very robust, versatile, perform self-repair, and have redundancy so that the functions of failed modules can be taken on by other modules. A successful implementation of such a robot, called PolyBot, was developed at the Xerox Research Centre in California, and reported in [4].

Many researchers believe that there is potential for improved co-operation between robots, if the robots can possess autonomous learning skills. Some of the applications in this field include predator/prey, box pushing, foraging, multi-robot soccer, and co-operative target observation.

The scope for research in distributed robotics is vast, and there are already many applications where the use of robot teams would be invaluable. Since the inception of the field less than twenty years ago, considerable advancement has been made. All the facets of multi-robot systems described above possess great potential for research, and many questions still exist, such as the fundamental advantages of using multiple robots, how humans can most easily control multi-robot systems, and how to formulate design strategies for robots based on the environments they will be working in.

1.2 Scope of Research

Since the area of research in intelligent robotics is so vast and cannot be covered even partially within the frame of an M.Eng. degree, the work presented in this thesis was focussed on only particular elements of robotics. In particular, robot control, and fault tolerant design of the control units, was attended to. The theories proposed are intended to provide quantitative techniques of determining fault tolerance characteristics of control unit systems. Control units will often get a large amount of information from external sensory elements of the robot. One of the most common sensory mechanisms is a vision system, and as such, vision systems have also been investigated. A new algorithm for vision systems is considered, and a practical implementation is presented.

Previous research in methods of monitoring and recording information about the progress of a program in a control unit resulted in a number of techniques, which will be addressed later in this thesis. Some of these require additional hardware

within the monitored circuit, some require additional hardware outside the monitored circuit, while others rely on the incorporation of additional data into the instructions of the program. Signature analysis [9,10] is among commonly used methods, employing bit patterns assigned to instructions to keep track of the program progress.

The concepts from this method were used as the basis for the research in this thesis. It was intended that techniques be found whereby the structure of a circuit could be analysed, resulting in information about the reliability of the circuitry.

Circuits and their data streams can be subject to a variety of faults, ranging from simple single-bit flipping, to complex errors such as patterned errors, and appearance and disappearance of bits. Since there is a variety of error checking techniques, each technique may well provide fault coverages differently for the error types. It is useful to determine how efficient each technique is for each fault type, and to draw conclusions about the ideal error checking method for each fault type. Simulations were conducted to gain information about several error checking techniques, and the coverages they provided with a selection of error types.

In considering vision systems, there are many tasks that a vision system may be required to perform. Common tasks include classification of objects based on shape and/or colour, calculation of object positions, and so on. An application was chosen where implementation of a new vision system would be advantageous. Using a suitable algorithm, the code for the system was implemented, and integrated into the whole robot system.

1.3 Thesis Overview

The key achievements of the research reported in this thesis are:

- The formulation of new techniques for modelling errors occurring in control unit systems. Methods have been suggested for representing error occurrence information.
- An approach to quantitative analyses of the performance characteristics of the circuitry has been developed using the methods for representing error occurrence information. These methods are mainly statistical, and use concepts from reliability theory to estimate parameters necessary for determining circuit reliability.
- On the basis of the concepts from reliability theory, estimates were made about the possible modes of failure of the circuitry. In addition to the modes of failure, probabilistic results were also acquired indicating the locations for these errors.
- Different error checking techniques were tested on a selection of fault types. The results provide insight into the way the error checking techniques detect errors, and the peculiarities of each technique that make it ideal for particular error types, and error locations within data.
- Elements of a fault tolerant system were successfully implemented in an image processing algorithm for robot vision.
- An image processing subsystem for mobile distributed robotics was studied. New algorithms for colour/position detection have been implemented in software, and integrated into a fully operational robot system.

1.4 Chapter Overview

Chapter II:

This chapter details the research done previously on fault-tolerant design of control units. Attention is drawn to the research that is specifically related to, and

forms the basis of, the research in control unit design for fault tolerance. New techniques for fault-tolerant design are discussed, different error types are investigated, and methods are suggested for the representation of these error types.

Chapter III:

This chapter considers a robotic sensory system – a visual system. The existing robotic system is evaluated for its current performance, and the algorithms it uses to achieve its requirements. The information required from the visual system is also evaluated. A suitable algorithm is developed, implemented in code, and integrated into the whole robotic system successfully.

Chapter IV:

Discussions, conclusions, and references are provided in Chapter IV, summarising the achievements of the research.

Appendices:

Software and simulation code are presented in Appendices I, and III. Appendix II contains the simulation results, and articles published on the work in this thesis are given in Appendix IV.

The research and results from this thesis have been discussed and published at a seminar at Massey University, and two international conferences – ISIC 2001 in Singapore [12] and IMTC 2002 in Anchorage, U.S.A. [13]. Copies of the papers presented at the conferences are attached in Appendix IV.

The developed software for image processing was done with partial support, and under the auspices of, the ASIA 2002 programme, which supports a collaboration

between Massey University and the Singapore Polytechnic (grant number: ED6019; Fault Tolerant Distributed Intelligent Systems). The visual system software was presented to Singapore Polytechnic during a research visit in September 2001. The software was customised by working with the Singapore Polytechnic collaborators and incorporated into the fully operational robotic team competing in the Robocup (robot soccer) world robotics competition [14].

1.5 References

[1] **Bekey, G., Parker, L.E., & Barhen, J.**; *Distributed Autonomous Robotic Systems 4*; Springer-Verlag, 2000

[2] **Asama, H., Fukuda, T., Arai, & T., Endo, I.**; *Distributed Autonomous Robotic Systems*; Springer-Verlag, 1994

[3] **Parker, L.**; *Current State of the Art in Distributed Autonomous Mobile Robotics*; Distributed Autonomous Robotic Systems 4, Springer-Verlag, 2000

[4] **Yim, M, Zhang, Y, & Duff, D.**; *Modular Robots*; IEEE Spectrum, Feb. 2002

[5] **Proceedings: Intelligent Autonomous Systems (Conference) (6th: Giudecca Island, Italy)**; *Intelligent autonomous systems 6*; Amsterdam, Oxford: IOS Press, c2000

[6] http://www.teambots.org

[7] http://ci.etl.go.jp

[8] **Cass, S.**; *Robosoccer*; IEEE Spectrum, May 2001

[9] **Demidenko, S.N., Levin, E.M., and Lever, K.V.**; *New approach to synthesis of self-checking microprogrammed control units with specified fault-detection probabilities*; IEE Proceedings-E, Nov. 1991, Vol. 138, No. 6

[10] **Demidenko, S.N., Levine, E.M., and Piuri, V.**; *Synthesis of On-Line Testing Control Units: Flow Graph Coding/Monitoring Approach*; IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, IEEE Computer Society, Proceedings 2000

[11] http://www.frc.ri.cmu.edu/robotics-faq.html

[12] **Demidenko, S., & Subramaniam, K.**; *Correlating Failure Modes With Component Reliability In Synthesis Of Self-Checking Control Unit Circuits*; Proc. ISIC 2001, pp. 190-192

[13] **C.H. Messom, S. Demidenko, K. Subramaniam, & G. Sen Gupta**; *Size/Position Identification in Real-Time Image Processing using Run Length Encoding*; IEEE Instrumentation and Measurement, Technology Conference, Anchorage, AK, USA, 2002

[14] http://www.sp-edu.sg/schools/eee.news.html

2 Design of Fault-Tolerant Control Units

2.1 Control Units

2.1.1 Introduction

Control units usually form the backbone of an electronic system, and as such, often require careful scrutiny during their design and fabrication. Designing the control unit to be fault-free, or at least fault tolerant, can be a time consuming but necessary process. There are two main methods by which control algorithms may be designed. The first is by the use of finite state machines (FSMs). Finite state machines involve graphical development of the control paths, whereby the complete algorithm is represented diagramatically.

The second method is known as microprogramming. The design methodology in this case is based on a programming approach, whereby a set of control signals is

defined. At any point in time, it is necessary that some of these control signals be asserted. Thus, executing a microinstruction has the effect of asserting the subset of signals it represents.

2.1.1.1 Finite State Machines

A finite state machine consists of a set of states and rules that define how to change states. Each state may constitute one or more conditions, which decide the next state to progress to. In addition, each state may have one or more outputs, which will be asserted when the system is in that state. In the implementation of a finite state machine, all outputs are assumed to be deasserted, unless explicitly asserted in that state. A finite state machine is also designed so that each state is occupied by the system for one clock cycle. Figure 2-1 illustrates a high-level abstraction of a finite state machine [1].

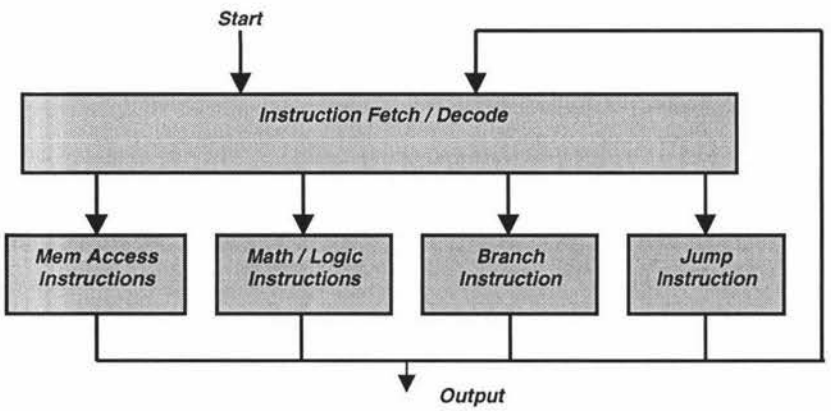


Figure 2-1: High-level view of finite state machine control.

In general, the instruction fetch / decode cycle requires two states, as shown in figure 2-2 below [1]:

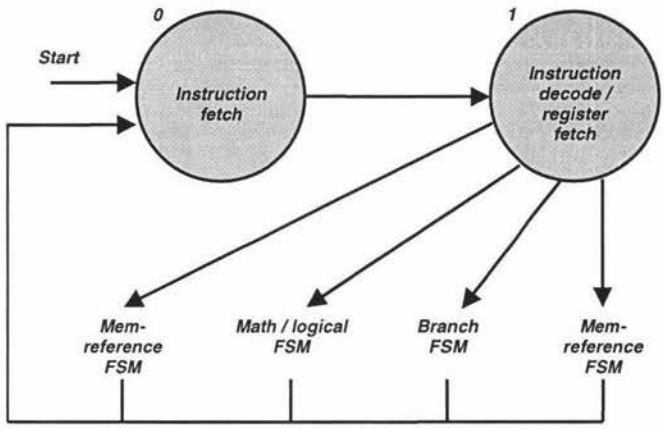


Figure 2-2: Instruction fetch and decode cycle.

The memory-reference instructions, typically involving storing data into memory from registers, or retrieving data from memory into registers, can be designed in a four-state machine. Figure 2-3 illustrates this design.

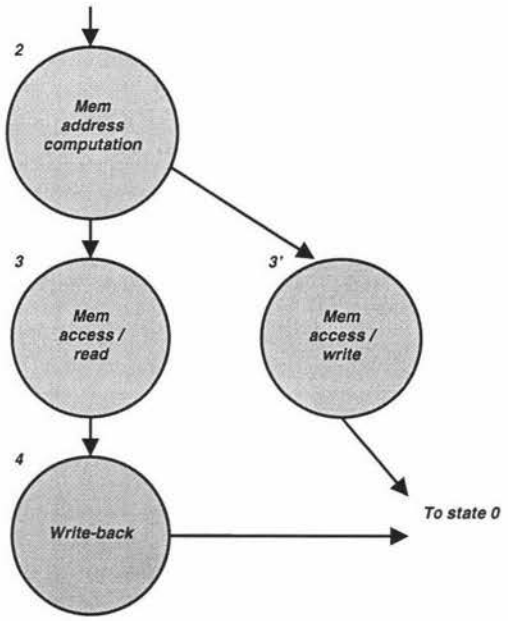


Figure 2-3: Memory reference instruction.

Arithmetic and logical operations can be implemented in a two-state finite state machine, as shown in figure 2-4.

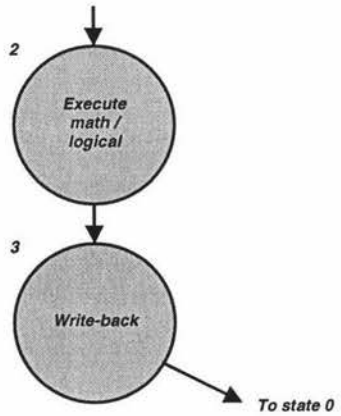


Figure 2-4: Arithmetic and logical instructions.

The remaining basic instructions are the 'conditional branch' and 'unconditional jump' instructions, both of which can be implemented in one instruction. The designs for these operations are illustrated in figures 2-5 and 2-6.

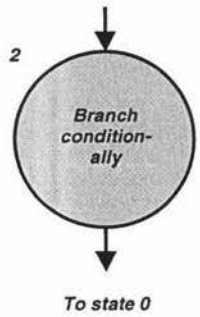


Figure 2-5: Conditional branch.

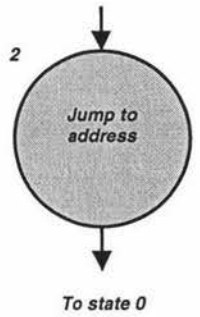


Figure 2-6: Unconditional jump.

The finite state machines illustrated above can be integrated to provide the building blocks for the functionality of a microcontroller. Once designed as a finite state machine, the system can then be implemented with a ROM.

2.1.1.2 Microprogramming

In systems which are extremely large, using finite state machines may become impractical. In such cases, the design methodology can be altered, and control signals to be asserted in a state can be considered an instruction that is executed by the datapath. Thus, executing an instruction has the effect of asserting the

control signals defined by that instruction. In representing the control signals as instructions, the sequence of the instructions can be defined by the use of a counter.

The microprogram acts as a symbolic representation of the control that will be translated by a program to control logic. The instructions are comprised of several fields, each of which represents a control line.

Once the system has been designed and the code generated, a microassembler is used to assemble the code and flag errors.

2.1.2 Design for Fault Tolerance

During program execution, the control subsystem (figure 2-7) generates instructions. These instructions are used by the processing subsystem in order to produce the required operations of the process. Therefore, it is imperative that the control subsystem not produce faulty output signals. Errors commonly occurring in control flow include simultaneous appearance of two or more instructions, disappearance of op-codes, erroneous jumps on logical conditions, substitution of correct instructions by erroneous ones [2]. In considering program flow, three structures are discussed – the follow structure, the subroutine structure, and the cycle structure. In order to continuously check the system for errors, external and internal methods of control-flow checking are available, whereby the content and sequence of control instructions is checked. Commonly implemented test regimes are 1) watchdog processors, which are useful for system-level observation; 2) monitors, which check control unit output; 3) concurrent self-test circuitry without reference signatures; and 4) self-test circuitry with reference signatures.

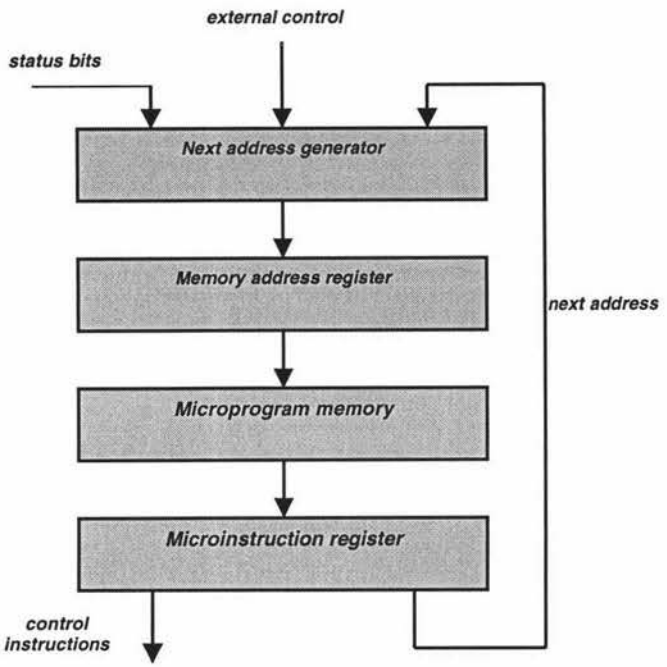


Figure 2-7: Microprogrammed control unit structure.

The alternative to integrating specialised circuitry into the controller itself is to monitor reliability-related parameters, such as current, temperature, radiation dose, and so on. The first three testing techniques are described briefly in section 2.1.2.1 – 2.1.2.3. Signature analysis is dealt with in section 2.1.2.4 in detail, since it forms the basis for the research in this thesis.

2.1.2.1 Monitoring Machines

An auxiliary monitoring machine operates in lock-step with the main finite state machine, so it is possible for the system to immediately detect any fault in either machine. Monitoring machines do not require the storage of any signatures, and allow for zero error-detection latency. Also, the monitoring machine is often not required to be as complex as the main machine, thus allowing the implementation to be simpler and lower in cost. Coverage is provided for delay faults and stuck-at faults [41], and the machine can be implemented independent of fault models, thus making the scheme useful for finite state machines that have already been built. Figure 2-8 illustrates the integration of a monitoring machine into the main system.

A major advantage provided by monitoring machines is that since the main machine and monitoring machine are designed differently, common mode failures are not likely to affect both machines identically [3,4].

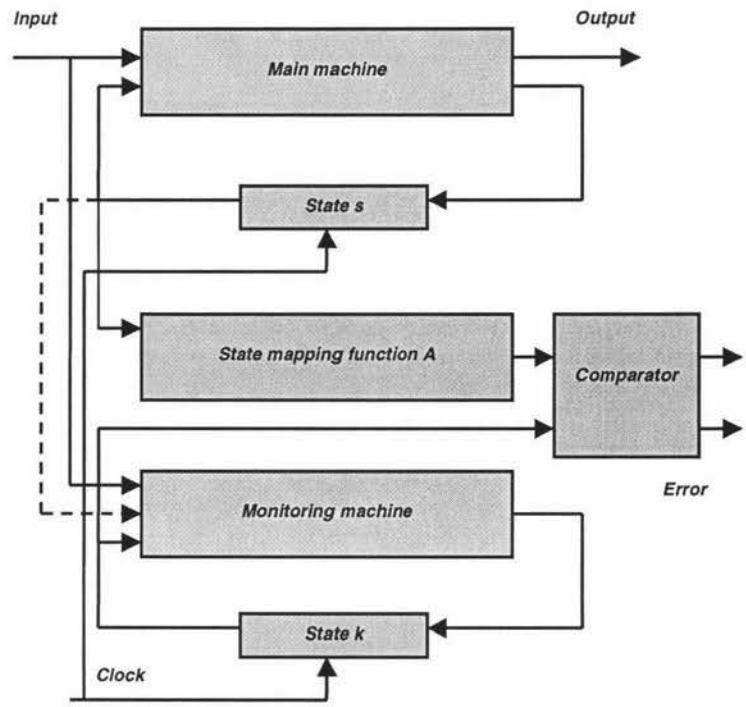


Figure 2-8: Incorporation of a monitoring machine into a system [3].

2.1.2.2 Watchdog Processors

Watchdog processors run concurrently with a main processor, checking its behaviour. Figure 2-9 illustrates the integration of a watchdog processor into a system.

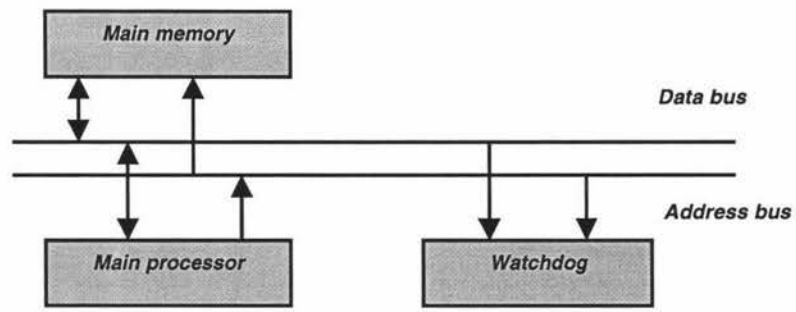


Figure 2-9: Use of a watchdog processor in a system [5].

The detection of errors using watchdog processors involves a 2-phase operation. Firstly, information is recorded about the process to be checked. The second step is to run the process, and gather data from the process that can be used for error detection. This information is then compared to the reference information. Information to be checked may include memory access behaviour, control flow, control signals, and reasonableness of results. Watchdog processors use circuitry that is independent of the circuitry being tested, and can therefore be implemented after the design phase, or in conjunction with existing test circuitry to increase system reliability.

2.1.2.3 Monitoring Techniques Without Reference Signatures

The main advantage of such techniques is in reducing the memory overhead associated with storing a signature at each vertex of the flow-graph [24]. A signature generation function is produced for every portion of sequential code, and successively applied to the op-code until the signature forms an m-out-of-n code [17], for a specified m and n (in an m-out-of-n code, m '1's must exist out of a codeword of length n). The location in memory that corresponds to the m-out-of-n code is tagged as a checkpoint. During the program's execution, the same signature generator is applied to the process, and if an m-out-of-n code does not appear at the tagged memory location, an error has occurred. To check at each tagged location whether the code is valid, a lookup table can be implemented in ROM, containing all the legal m-out-of-n codes [6, 7, 10].

Parity checking [7, 8, 27, 28] is another method which can be used, and will detect all single-bit errors. In fact, it will detect all errors of odd-numbered multiplicity. It is a highly compact code, since it uses only one bit for the error information. There are also more complex error coding methods based on parity checking [9].

A set of codes known as unordered codes [7, 8] can often be implemented effectively. In this technique, codes are generated whereby no two different code words, x and z, exist such that x covers z ($x \supseteq z$). X covers z denotes x having a 1 in every position that z has a 1. Thus, an error can be detected when two codes result from different seeds, but satisfy the property $x \supseteq z$. M-out-of-n codes and Berger codes [7] are examples of unordered codes.

The Berger code is a separable unordered code [7, 10]. The error coding part records the number of 0s in the data part. An alternative form of this code is for the error coding part to contain the complement of the number of 1s in the data part.

Arithmetic codes [7, 8, 27, 28] come in two forms: separable and non-separable. In separable arithmetic codes, the code words X' are obtained by using the data bits X to give a remainder by their modulo division of a base A ($X' = |X|_A$). $X' = A - |X|_A$ (inverse residue code) can also be used. In non-separable arithmetic codes the code word is equal to the product of the data bits and the base A ($X' = XA$).

The Dual-Rail Coding [8] method duplicates the information bits, and uses its complement as the error coding part. This is a very reliable coding technique, since any errors in the data bits or its complement can be detected. The disadvantage, however, is that duplicating all the data bits is expensive in memory.

Bose-Lin codes [10] were developed by Bose and Lin, and are optimal codes for detecting up to X unidirectional errors. The codes are systematic, and require a fixed number of error coding bits, independent of the number of data bits. Being systematic codes, the error bits are simply appended to the data bits, and no additional circuitry is needed to recover just the data bits.

In addition to the techniques mentioned above, many examples can be found of effective error coding techniques [16-18, 20-23, 25, 29, 30].

2.1.3 Process Monitoring Using Signature Analysis

2.1.3.1 Components of a Flow Graph

The algorithm governing a microprogrammed control unit can be described in several ways: by a set of instructions, by a matrix of transitions, or by a flow graph. It is most useful to use the flow graph method in introducing signature analysis concepts [26].

A flow graph consists of a set of vertices. The start vertex is denoted as Q_0, and the end vertices as Q_k, where k=1,2,3... [2]. There may be any number of intermediate vertices (states) between the start and end points. These vertices may be of several typical vertex types.

The first vertex type is the branch vertex, which is a logical operation. Depending on a logical condition, a branch is made to one of a number of possible vertices. The second vertex type is a convergent vertex. Here, two or more edges may make transitions to the same vertex. The third vertex type is simply a vertex that is part of a linear, serial section of the graph.

In designing control unit diagnostic features, a compromise is necessary between efficiency (fault coverage, latency, and accuracy of fault location), and simplicity of the diagnostic features [2]. In order to achieve the correct balance, the complexity of the flow graph can be reduced, by grouping vertices into sets, which will form the structural elements of the diagnostic system. The groups may consist of one vertex, pairs of sequential vertices, groups of sequential vertices, or even the flow graph as a whole [19]. The selection of fragments is an important phase in reducing the flow graph to an efficiently testable form. The graph reduction must also be done with respect to the testing method to be used on the

system. In order to partition the graph efficiently, one method is to use graph vertex colouring theory to form the labelled subgraphs.

2.1.3.2 Vertex Check Keys

A bit pattern, or key, can be assigned to every partition of the flow graph [2,13,15]. This is illustrated in figure 2-10.

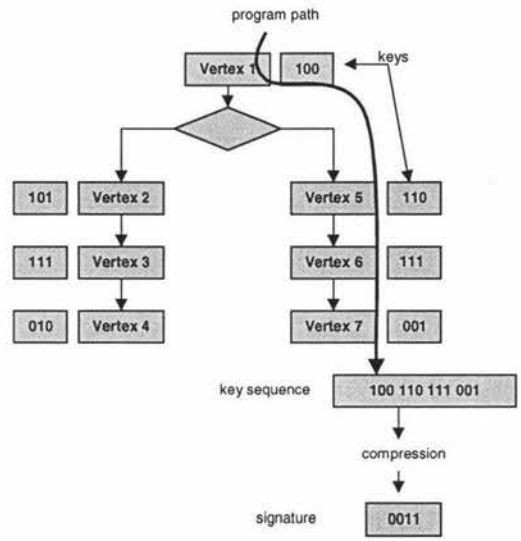


Figure 2-10: Keys assigned to each vertex are concatenated and compressed to form signature.

Now, the progress of the system can be monitored at any stage by comparing the key of the current graph segment with its equivalent pre-determined key corresponding to fault free functioning of the control unit. This is the essence of the control flow checking process. A number of alternatives are available for accommodating the reference key values. Firstly, they may be stored in the microprogram memory, in additional instruction fields created for the purpose. Alternately, they may be stored as part of the program itself, as special words. A third option is to have special memory units for the keys. Finally, they may not be stored in memory at all, and can instead be generated by special circuitry as they are needed by the program. The vertex keys may be defined by the diagnostics designer (assigned keys) at the time of implementation of the system. Alternately, they may be produced real-time based on signals within the reference

system (derived-keys) similar to the one being tested. They may also be derived from real signals in the special first execution of a real fault-free system. Assigned keys are either coded into the diagnostic features in advance, or they are generated in the course of program execution and monitoring. Derived-keys may have several sources: a microinstruction as a whole, a control field or fields, values of the signals in states of the control unit, or a microinstruction address.

Keys and vertices can be set up to have a one-to-one correspondence. If this situation exists, the keys are called valid keys. The sequence of flow graph vertices, which corresponds to the execution of a program, can be described by a series of valid keys. This provides a map of the path of a process through the whole system. In practice, memory constraints may not make it practicable to store whole sequences of keys. In this case, a new group of keys can be created to improve the reliability of the error detection in the process. These keys, unlike the first set of keys, contain information about the whole path of the process from the first vertex to the current vertex (or graph segment). To distinguish the two forms of keys, the first set of keys (which denote only the current state) are termed first-level keys (FLKs), and the second set (which contain information about the history of the process) are termed second-level keys (SLKs).

2.1.3.3 Generation of Vertex Check Keys

The compression of a binary sequence, such as a key, is termed space compression (SC), and this is done by a space compression block (SCB). An example of this is calculation of parity or Hamming code check bits. The mapping of a binary sequence onto a corresponding short word is termed time compression (TC), and the device that does this can be termed a time-compression block (TCB). Calculation of a signature using a feedback shift register is a typical example of this. In fact, SC may be used to obtain the values of first-level keys, while TC may be used to obtain the values of second-level keys. If time-division multiplexing is used, both processes may be done using the same circuitry.

An SLK is produced by compressing the corresponding sequence of FLKs. Different compression methods can be used for the SLK, allowing for varying degrees of reliability.

2.1.3.4 Using Check Keys to Monitor Control Flow

Either first-level or second-level keys can be used for fault monitoring. Each, however, provides its own advantages and disadvantages. Since each FLK is stored as a code for only its state, it is obvious that use of FLKs must be done in conjunction with step-by-step checking. Otherwise, it is impossible to insure that all single faulty transitions from one vertex to another (denoted $Q_i \rightarrow Q_j$) can be detected [11,12]. Using SLKs provides the advantage of reducing the number of checks. Since each key contains some information about the states before the current state, checking can be done in intervals. This may, however, result in multiple faulty transitions between checks going unnoticed.

2.1.3.5 Probabilistic Fault Coverage

The quality of flow-graph checking (the error coverage) is most effectively acquired by probabilistic methods. Several parameters are introduced here, which are used in the statistical estimates [11,12]. The probability of detecting a fault is denoted P_d. The occurrence of a faulty transition is denoted $Q_{n,j} \rightarrow Q_{m,p}$ where $Q_{n,j}$ and $Q_{m,p}$ are any two vertices between which the transition is not legal. Thus, $P_d(Q_{n,j} \rightarrow Q_{m,p})$ is the notation used to indicate that the faulty transition is detected by the diagnostic technique used. The value of this probability depends on the type and distribution of the keys in the flow graph, on the error detection method used, and the data compression technique used. This is discussed in detail in [11,12]. For each particular faulty transition $Q_{n,j} \rightarrow Q_{m,p}$, the value of P_d is binary, in that it is either zero if the error goes undetected, or one if the error is detected. There is also a probability that an error occurs when the flow graph is in

a vertex. This is denoted as $P_w(Q_{n,j})$, and is the probability that a faulty transition occurs from $Q_{n,j}$ to any other vertex in the flow graph. In general, the probability of all faulty transitions from a given vertex to all other vertices in the flow graph is denoted $P_w(Q_{n,j} \rightarrow Q_{m,p})$, where

$$\sum_m \sum_p P_w(Q_{n,j} \rightarrow Q_{m,p}) = 1$$

..(eq.2-1)

Then, we have

$$P_d(Q_{n,j}) = P_v(n) P_v(Q_{n,j}) \sum_m \sum_p P_w(Q_{n,j} \rightarrow Q_{n,p}) P_d(Q_{n,j} \rightarrow Q_{m,p})$$

..(eq.2-2)

where $P_v(n)$ is the probability that the transition will take place from within the *nth* section of the flow graph, and $P_v(Q_{n,j})$ is the probability that the fault has taken place precisely at the *jth* vertex on its occurrence in the nth section of the flow graph. It follows that the probability $P_d(n)$ of detecting single faulty transitions from the vertices of the *nth* section of a graph is:

$$P_d(n) = P_v(n) \sum_j \sum_m \sum_p P_v(Q_{n,j}) P_w(Q_{n,j} \rightarrow Q_{m,p}) P_d(Q_{n,j} \rightarrow Q_{m,p})$$

..(eq.2-3)

Single faulty transitions may be made within the *nth* section of the flow graph, or alternatively, to a vertex in another section of the graph. $P^i_d(n)$ specifies the probability of detecting faulty transitions inside a section:

$$P^i{}_d(n) = P_v(n) \sum_j \sum_p P_v(Q_{n,j}) P_w(Q_{n,j} \rightarrow Q_{n,p}) P_d(Q_{n,j} \rightarrow Q_{n,p})$$

..(eq.2-4)

where

$$\sum_p P_w(Q_{n,j} \rightarrow Q_{n,p}) = 1$$

..(eq.2-5)

Similarly, the probability of a faulty transition from the *nth* section of the flow graph to a vertex in another section is:

$$P^o{}_d(n) = P_v(n) \sum_j \sum_m \sum_p P_v(Q_{n,j}) P_w(Q_{n,j} \rightarrow Q_{m,p}) P_d(Q_{n,j} \rightarrow Q_{m,p})$$

..(eq.2-6)

where

$$\sum_{m}\sum_{p} P_w(Q_{n,j} \rightarrow Q_{m,p}) = 1 \quad and \quad m \neq n$$

..(eq.2-7)

According to the law of total probability, all the probability values are interrelated:

$$P_d(n) = P^i{}_w(n)P^i{}_d(n) + P^o{}_w(n)P^o{}_d(n)$$

..(eq.2-8)

where $P^i{}_w(n)$ and $P^o{}_w(n)$ are the probabilities that faulty transitions will take place from the vertices in the *nth* section to the other vertices inside the section, and outside it, respectively. The probability of detection of faulty transitions from separate vertices and sections to the entire flow graph is thus:

$$P_d = \sum_{n}\sum_{j}\sum_{m}\sum_{p} P_v(n)P_v(Q_{n,j})P_w(Q_{n,j} \rightarrow Q_{m,p})P_d(Q_{n,j} \rightarrow Q_{m,p})$$

..(eq.2-9)

where $P_v(n)$ is the probability that the transition will take place from a vertex within the *nth* section of the flow graph.

The following observations are taken from [11,12], and are useful in characterising and quantifying flow graph errors.

Generally, the occurrence of an error such as a faulty transition from the vertex Q_i to Q_j, can be a complex event, and can be broken down to be composed of a series of events:

i) occurrence of a fault in the vertex Q_i;

ii) occurrence of a transition from vertex Q_i to Q_j;

iii) possible detection of the faulty transition by error detection circuitry

The probability of an error occurring in the first place, in a vertex of the flow graph, depends on the reliability characteristics of the hardware used in the implementation of the control unit [14]. The flow graph structure, the ranges and variation of the input data signals, the distribution of these signals, and the probabilities of occurrence of the input signals are also all major factors in determining the error probabilities. The estimates of these probability parameters may be calculated theoretically by investigation of the hardware reliability characteristics, or empirically by experimentation. Here, the vector F (discussed in Section 2.2) can be used. The vector is of dimension N (where N is the number of vertices in the flow graph), and each element f_i is the probability of occurrence of an error in the Q_ith vertex. Each value f_i in the vector is in turn dependent on the probability that the process arrives in that vertex.

The probabilities of faulty transitions occurring between pairs of vertices can be expressed in the square matrix T, of dimensions N x N. Here, each element t_{ij} denotes the probability of a faulty transition occurring from vertex Q_i to Q_j, in the event of an error occurring during the execution of the instructions in Q_i.

A square matrix D, with dimensions N x N, is used to indicate the error detection capabilities of the diagnostic method used. If an error in the D matrix can be detected by the diagnostic methods used, the value of the corresponding entry in the D matrix is a one. Otherwise, a zero is entered to indicate that particular erroneous transition cannot be detected (no fault coverage). In matrix form, the equation governing the probability of detection of the errors by the diagnostic features is as follows:

$$P = F \times T * D \qquad \text{..(eq.2-10)}$$

where '*' indicates element-by-element multiplication, not normal matrix cross-products. The elements $p_{i,j}$ of the resultant matrix are the values of the probabilities of detecting faulty transitions between the corresponding Q_ith and Q_jth vertices of the flow graph. Example flow-graphs, and the calculations

required to obtain their fault coverages using all the parameters mentioned above are provided in [11,12].

2.1.4 Section Summary

Control units form one of the major components of any electronic system. In order to ensure that the output produced by the programs in the system is reliable, the performance of the control unit must be monitored for errors. Programs generally consist of sets of microinstructions, and in order to monitor the flow of a program, a monitoring technique must be used to keep track of the instructions executed by the system. In modelling the system, it is important to consider its hardware, as well as its software, which can be illustrated as a flow graph. Many techniques are available which allow monitoring of a control unit, and the technique of interest here is that of signature analysis. Signature analysis uses keys (bit sequences), which are appended to microinstructions. For each instruction that is executed, its key is appended to the existing sequence of keys. Ultimately, the key sequence is compressed using a feedback shift register (corresponding to a specified polynomial), and the result is the signature. The signature can be used to detect if an error occurred, by comparing it to a predetermined signature from an error-free execution of the system. This is the fault coverage, and can be determined mathematically using equations such as eq.2-1 and eq.2-2. The matrices described in eq.2-2 can be used to completely model the theoretical fault coverage, and the research presented in this thesis aims to find accurate methods of determining the parameters of these matrices based on the system's hardware and flow-graph characteristics.

2.2 Flow Graph and Hardware Reliability

2.2.1 Introduction

In section 2.1, it was stated that the reliability of a system could either be obtained by calculations based on the reliability characteristics of the implementation hardware, or by empirical methods whereby observations could be made of the system behaviour, and conclusions drawn about its reliability. In calculating the reliability of a system, it is important to recognise that the system can be divided into two segments – its software and hardware [1]. This enables the reliability calculations to be done from two perspectives. The hardware segment simply refers to the actual hardware in which the system has been implemented – the whole microcontrollers, as well as the individual components of the circuitry. The software refers to the flow graph structure [2, 11, 12, 31], which illustrates all the possible paths the process may take through each execution, as well as the probability that the process arrives in each vertex of the flow graph.

Calculating system reliability in essence means investigating equation 2-10, and finding methods to accurately estimate the parameters of the three matrices: F – for the fault probabilities of each vertex; T – for the erroneous transition probabilities; and D – for the detection matrix. Techniques for estimating the parameters of the F matrix are detailed in sections 2.2.3 to 2.2.5. Here, methods are presented whereby system flow graphs and hardware may be analysed to yield the desired parameters. Methods of representing fault types are discussed in Section 2.3, and demonstrate easy calculation of the fault probabilities; this is an investigation of the T matrix. The D matrix is dealt with in Section 2.4, where error detection capability is investigated.

2.2.2 System Parameters

In order to demonstrate the reliability calculations, it is first necessary to define certain parameters of the system being used in the examples. Theoretically, any flow graph structure and hardware may be used. In the case of the examples, however, simple systems have been chosen to explain the theory.

Each vertex of the flow graph has a time period during which the instructions in that state are executed. A vertex is as shown in figure 2-11 below.

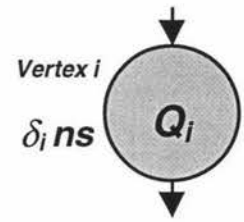


Figure 2-11: A flow graph vertex.

The vertex has state number, and this number will usually be denoted in binary, as is customary in finite-state machine design. The time period, δ_i, for the vertex is denoted in nanoseconds, and the process is regarded as being in this state for the time period.

A hardware element is as shown in figure 2-12.

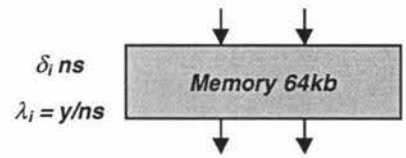


Figure 2-12: A hardware element.

The hardware element may be any component of the circuitry. It has a time period of δ_i nanoseconds, which is the time period required for an average operation to be performed in the component. For example, in a memory component this may be the time taken from receiving an address, to retrieving data from that address. The second major parameter of the hardware element is

λ_i. This is the component's failure rate, as defined by reliability theory, and is quoted in failures per nanosecond. Failure rates and reliability information for hardware components is often available in the manufacturer's specifications for the product.

2.2.3 General System Reliability

In reliability theory [32-35] electronic components are often modelled as possessing constant failure rates. The reliability of an element with constant failure rate can be calculated by:

$$R(t) = e^{-\lambda t}$$

..(eq.2-11)

where λ is the failure rate, and t is the time elapsed.

Now, in a real system, the hardware architecture will be of the form:

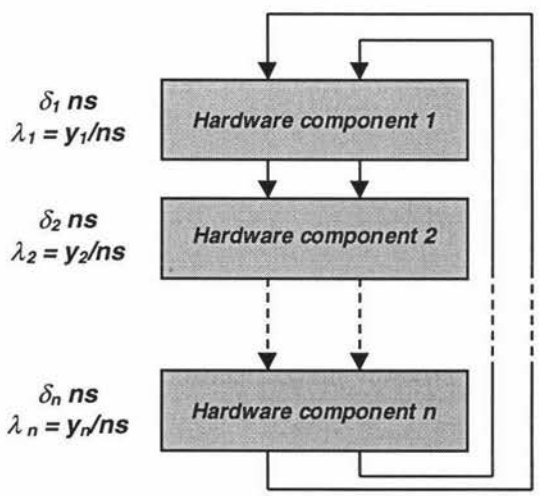


Figure 2-13: General model of a hardware implementation.

The processes will generally follow a routine, such as next address generation, register accesses, memory read operations, memory write operations, instruction fetch and decode operations, and so on. With components placed as such in series, the failure rate of the entire system, λ_s, is simply:

$$\lambda_s = \lambda_1 + \lambda_2 + ... + \lambda_n$$

..(eq.2-12)

Now, the individual reliabilities of the vertices of the graph can be calculated, bearing in mind that each vertex would take on average λ_s nanoseconds to execute. Thus, the reliability of a vertex is:

$$R_{Qi}(\sum_{k=1}^{i} \delta_k) = e^{-\lambda_s \sum_{k=1}^{i} \delta_k}$$

..(eq.2-13)

Here, Q_i is the vertex where the reliability is being calculated. The summation indicates that the reliability is dependent on the time period that the process has been running for. Therefore, the reliability reduces as the process progresses, and the reliability of the whole system when the process is in one of the last vertices is much lower than when the system is in one of the first vertices.

Finally, the mean-time-to-failure (MTTF) parameter θ_s can be calculated:

$$\theta_s = \frac{1}{\lambda_s}$$

..(eq.2-14)

The calculations can be illustrated with the following example system in figures 2-14 and 2-15:

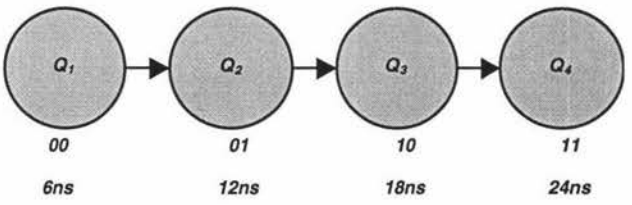


Figure 2-14: Example system flow graph.

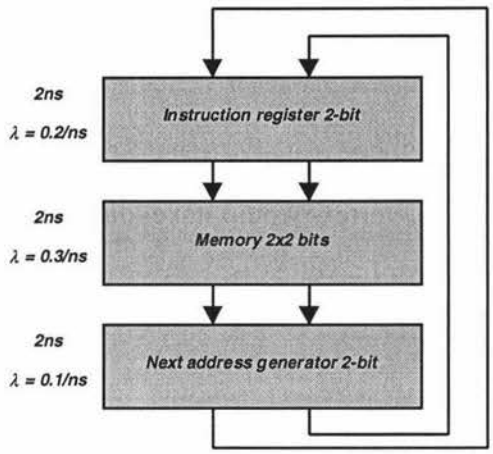


Figure 2-15: Example system hardware implementation.

The overall failure rate of the system λ_s, using eq.2-12, is:

$$\begin{aligned}\lambda_s &= \lambda_1 + \lambda_2 + \lambda_3 \\ &= 0.6/\text{ns}\end{aligned}$$

Thus, the reliabilities of each of the vertices can be calculated using eq.2-13:

Q_1: $R_{Q1}(6) = e^{(-0.6)(6)} = 0.0273$

Q_2: $R_{Q2}(12) = 0.0007$

Q_3: $R_{Q3}(18) = 2 \times 10^{-5}$

Q_4: $R_{Q4}(24) = 5 \times 10^{-7}$

The mean-time-to-failure (MTTF) of the system, θ_s, using eq.2-14:

$$\begin{aligned}\theta_s &= 1/\lambda_s \\ &= 1.67\text{ns}\end{aligned}$$

Interpreting this result, it is expected that there is a 63% chance that the system will fail within 1.67ns of operation.

2.2.4 Vertex Execution Probability

Once the reliability of each vertex can be calculated, the next important parameter to consider is the probability, for each vertex, that the process arrives in that vertex. Otherwise, the true reliability of a system cannot be known without ascertaining what contribution each component has to the overall functioning of the system. To record the probabilities, a vector is used:

$$A = \begin{bmatrix} P(Q_1) \\ P(Q_2) \\ \vdots \\ P(Q_n) \end{bmatrix}$$

These probabilities can be acquired by inspection of the flow graph. Branches and jump instructions are the main features to take into consideration during inspection, as it is at these points that paths separate, and thus probabilities split. At each branch or jump, it is important to know the probability of each of the paths being taken. A further consideration is at vertices where paths converge, as these vertices will be executed with the combined probabilities of the paths converging at them. So, for example, if only one vertex exists where the process can end, then it must have a probability of execution of 1, since the process will certainly arrive at this vertex sometime. Once again, the probability of failure of each vertex, or its unreliability, can be calculated by considering the time taken for the process to reach that vertex, and the reliabilities of the vertices before it i.e. the probability that the process reached that vertex before a fault occurs. If in practice it is discovered that the system to be evaluated contains components that do not exhibit constant failure rates, the reliability calculations will be more complex, and will require Weibull analysis [32-35].

The values obtained during the reliability analysis represent the reliabilities associated with the vertices, but the data necessary is in fact the probability of failure at those vertices – the unreliabilities. The unreliability of a component, F(t), is simply:

$$F(t) = 1 - R(t)$$

..(eq.2-15)

These values can be represented in the matrix b, which consists of the probability of failure of each vertex, when the process passes through that vertex:

$$B = \begin{bmatrix} p_f(Q_1) \\ p_f(Q_2) \\ \vdots \\ p_f(Q_n) \end{bmatrix}$$

In order to analyse the system for faults it is necessary to acquire information regarding the faulty transitions possible, and their probabilities. This is done with respect to each pair of vertices, and therefore the possibilities can be represented in a square matrix with all the transitions between each pair of vertices present:

$$G = \begin{bmatrix} Q_{1-1} & Q_{1-2} & \cdots & Q_{1-n} \\ Q_{2-1} & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ Q_{n-1} & \cdots & \cdots & Q_{n-n} \end{bmatrix}$$

Here, $Q_{1\text{-}1}$ denotes the erroneous transition from vertex 1 to itself, in the event an error occurs when the process is in vertex 1. Note that only erroneous transitions are recorded in the matrix, so the probabilities of legal transitions are set to zero.

Given matrices A, B, and G, it is possible to calculate the probability of a particular fault occurring in a given vertex, during execution of a process in the flow graph. Certain fault types, such as stuck-at faults, will result in erroneous transitions due to some states being rendered inaccessible. Such a fault will be reflected in the G matrix, as impossible state transitions will have zero probabilities. In general, however, the three matrices can be used thus:

$$p(fault\_type) = G \times (A * B)$$

..(eq.2-16)

In this case, p(fault_type) is the probability of a particular fault type occurring. The matrices would reflect the particular characteristics of the fault type under consideration. The '*' denotes element-by-element multiplication, and the 'x' denotes normal matrix multiplication.

As an example of the techniques suggested above, a system has been formulated (figure 2-16). The hardware of the system consists of a 2-bit instruction register, a 2x2 bit memory, and a 2-bit next-address generator. Each module of the hardware has been assigned an execution time of 2ns, and therefore a full cycle takes 6ns. Each of the modules of the system also has a failure rate associated with it.

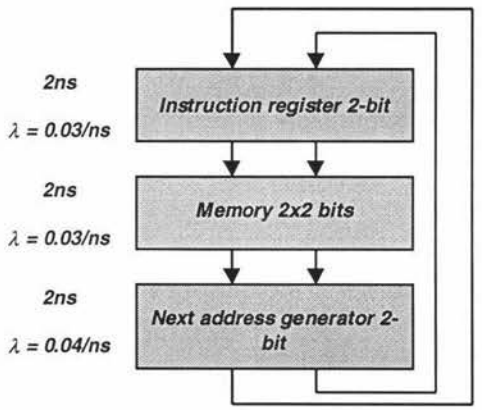


Figure 2-16: Example system hardware implementation.

The flow graph of the system consists of four vertices (figure 2-17), each of which takes a full CPU cycle to execute, and thus 6ns. The vertices have also been assigned 2-bit addresses. Both the paths from the branch condition are assumed to occur with equal probability. Since each of the vertices executes in 6ns, Q_1, Q_2, and Q_3 increment in time steps of 6ns, but Q_4 will be reached in 12ns in half of the executions, and in 24ns the rest of the time.

It is necessary to determine, for each state, the probability that the process will be in that state at any given time in the system's operation. To achieve this, the

system flow graph can be reduced to two parallel systems, bisected by the branch condition.

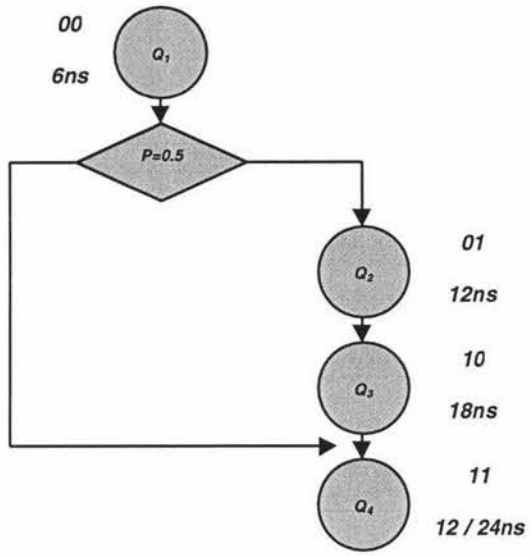


Figure 2-17: Example system flow graph.

Figure 2-18 illustrates the two parallel systems. The first sub-system consists of just two vertices - Q_1 and Q_4, taking a time of 12ns to execute. The second sub-system contains all four vertices, thus taking 24ns to execute.

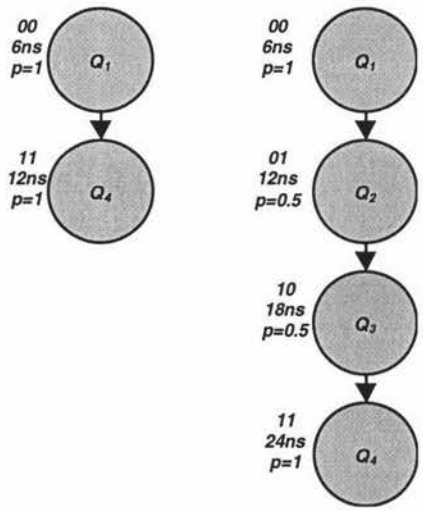


Figure 2-18: System decomposition into two parallel subsystems.

From this decomposition of the system, it can be seen that all the vertices do not contain the same probability of being executed. In other words, the probability of the process passes through a vertex varies from one vertex to another.

The 'p' values assigned to the vertices indicate the probability of the process passing through that vertex. Thus, vertices Q_1 and Q_4 have probabilities of 1, since the process will always pass through these vertices, regardless of which path is taken from the branch. Vertices Q_2 and Q_3, on the other hand, have probabilities of 0.5, since they will only be executed when the right branch is taken, and this occurs only half the time on average. Thus, the matrix A is:

$$A = \begin{bmatrix} 1 \\ 0.5 \\ 0.5 \\ 1 \end{bmatrix}$$

The failure rate of the entire system, λ_s, is the sum of the individual failure rates:

$$\begin{aligned} \lambda_s &= 0.03 + 0.03 + 0.04 \\ &= 0.1/\text{ns} \end{aligned}$$

Using the reliability formula, the reliabilities of the vertices can be calculated:

Q_1: $R_{Q1}(6) = e^{-0.1 \times 6} = 0.548$
Q_2: $R_{Q2}(12) = e^{-0.1 \times 12} = 0.301$
Q_3: $R_{Q3}(18) = e^{-0.1 \times 18} = 0.165$

Since the reliabilities are calculated relative to the time taken to reach each vertex, an ambiguity occurs at vertex Q_4, which may be reached in 12ns or 24ns. To compensate for this, the average reliability can be taken since each path to vertex Q_4 has equal probability (if the branch did not have equal probabilities for the paths, a weighted average would be needed to compensate for the most common path to the vertex). Thus,

Q_4: $R_{Q4}(12) = e^{-0.1 \times 12} = 0.301$
$R_{Q4}(24) = e^{-0.1 \times 24} = 0.091$

and

$$R_{Q4avg} = (0.301+0.091) / 2 = 0.196$$

The failure probabilities are then:

Q_1:	$F_{Q1}(t) = 0.452$
Q_2:	$F_{Q2}(t) = 0.699$
Q_3:	$F_{Q3}(t) = 0.835$
Q_4:	$F_{Q4}(t) = 0.804$

These are put into the B matrix:

$$B = \begin{bmatrix} 0.452 \\ 0.699 \\ 0.835 \\ 0.804 \end{bmatrix}$$

Finally, the erroneous transition probabilities are shown in the G matrix.

$$G = \begin{bmatrix} Q_{1-1} & Q_{1-2} & Q_{1-3} & Q_{1-4} \\ Q_{2-1} & Q_{2-2} & Q_{2-3} & Q_{2-4} \\ Q_{3-1} & Q_{3-2} & Q_{3-3} & Q_{3-4} \\ Q_{4-1} & Q_{4-2} & Q_{4-3} & Q_{4-4} \end{bmatrix}$$

$$= \begin{bmatrix} 1/2 & 0 & 1/2 & 0 \\ 1/3 & 1/3 & 0 & 1/3 \\ 1/3 & 1/3 & 1/3 & 0 \\ 1/4 & 1/4 & 1/4 & 1/4 \end{bmatrix}$$

The G matrix here shows all the erroneous transitions possible. If, now, an error is introduced into the system, the parameters will change. The graph in figure 2-19 illustrates the use of the matrices for a stuck-at zero fault in the least significant bit (LSB) of the hardware's next-address generator. In the example, matrices A and B remain unchanged, while matrix G is altered to reflect the changes in the flow graph paths. Figure 2-19 below shows the flow graph, with the inaccessible transitions in grey, and the erroneous transitions in dashed lines.

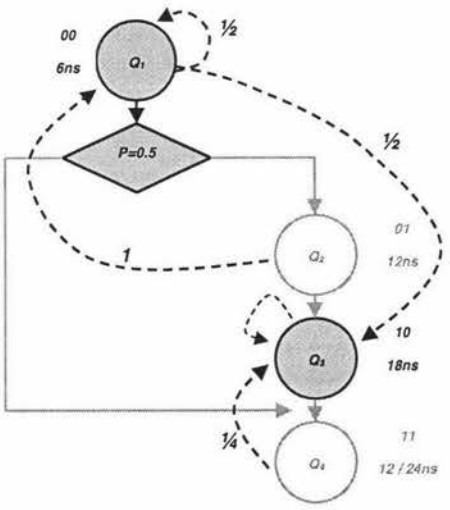


Figure 2-19: Flow graph transitions with fault.

The G matrix for this flow graph is given below, with the probabilities for the transitions to inaccessible states removed:

$$G = \begin{bmatrix} ½ & 0 & ½ & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & ¼ & 0 \end{bmatrix}$$

The fault probabilities now are:

$$p(stuck\_at\_zero\_fault) = G \times (A * B)$$

$$= \begin{bmatrix} ½ & 0 & ½ & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & ¼ & 0 \end{bmatrix} \times \left\{ \begin{bmatrix} 1 \\ ½ \\ ½ \\ 1 \end{bmatrix} * \begin{bmatrix} 0.452 \\ 0.699 \\ 0.835 \\ 0.804 \end{bmatrix} \right\}$$

$$= \begin{bmatrix} 0.4348 \\ 0.4520 \\ 0.4175 \\ 0.1044 \end{bmatrix}$$

The value of 0.4348, in the first row of the vector, implies that there is a probability of 0.4348 of the system failing at Q_1, in the presence of a stuck-at zero fault in the least significant bit of the next address generator. Fault probabilities can also be calculated for other faults occurring in the system. For each fault, the flow of the process will be affected uniquely, and this change will be reflected in the G matrix.

2.2.5 Hardware Reliability

Faults may occur in different parts of the hardware. In order to investigate these faults it is convenient to consider the system on a vertex-by-vertex basis. Each vertex can be divided into its constituent memory access routines, thus breaking each instruction up into the individual hardware elements used. In addition, it is also necessary to recognise that each hardware element has its own reliability characteristics. If a vertex requires a certain time period to execute, and uses a set of hardware components in its execution, the vertex can be deconstructed as shown in table 2-1:

Process time period (ns)	Hardware element	Failure rate (/ns)
$0 - t_1$	Component 1	y_1
$t_1 - t_2$	Component 2	y_2
:	:	:
$t_{n-1} - t_n$	Component n	y_n

Table 2-1: Constituent hardware accesses for a vertex

Once deconstructed, the fault probabilities of each hardware element can be found by summing the corresponding accesses by all the vertices to each hardware element.

2.2.5.1 Hardware Component Usage

In the execution of each vertex, hardware elements are used in turn. In order to investigate their reliability characteristics, however, it is important to recognise that they are still considered to be in use between operations, as illustrated by figure 2-20. Each hardware component is used by a number of vertices, denoted μ_1, μ_2, and so on.

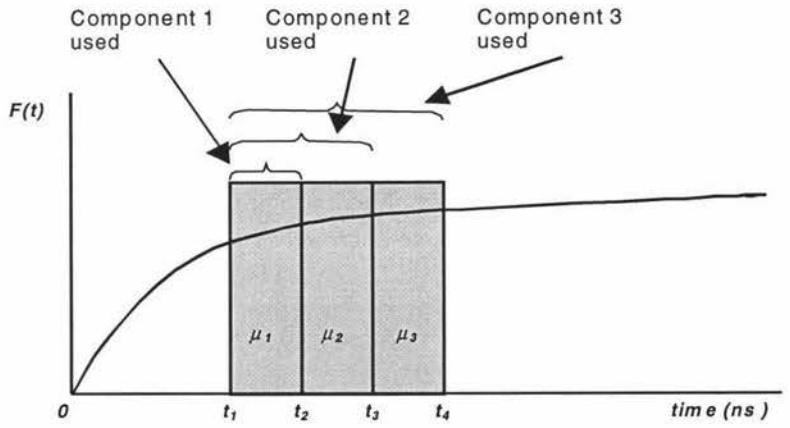


Figure 2-20: Hardware component usage by a vertex.

Now, to calculate the failure probabilities of the individual hardware units, each unit can be considered in turn. Firstly, if component w is used until time t_r, then the probability of failure of component w, using reliability theory, can be calculated as follows:

$$F_w(t) = \int_0^{t_r} e^{-\lambda_w t} dt$$

..(eq.2-17)

The failure probabilities for processes μ_2 and μ_3 can be derived in a similar manner.

2.2.5.2 Total Probabilities for Component Usage

To calculate the usage that each hardware element undergoes, and its probability of failure, the probability of each vertex being executed needs to be taken into consideration. For a single vertex r a vector can be used to represent the failure probabilities of each of the hardware elements used:

$$F_{Q_r} = \begin{bmatrix} Q_{r \to c1} \\ Q_{r \to c2} \\ \vdots \\ Q_{r \to cK} \end{bmatrix}$$

Here, K represents the total number of hardware elements in the system. The subscript *c1* denotes hardware component 1, and $Q_{r\to c2}$ denotes vertex *r* using hardware component 2. If this is true for each vertex, then the total failure probabilities for all the hardware elements can be derived from summing the individual failure probabilities acquired from each transition:

$$\begin{bmatrix} F_{c1}(t) \\ F_{c2}(t) \\ \vdots \\ F_{cK}(t) \end{bmatrix} = P_{Q_1}\begin{bmatrix} Q_{1\to c1} \\ Q_{1\to c2} \\ \vdots \\ Q_{1\to cK} \end{bmatrix} + P_{Q_2}\begin{bmatrix} Q_{2\to c1} \\ Q_{2\to c2} \\ \vdots \\ Q_{2\to cK} \end{bmatrix} + \cdots + P_{Q_n}\begin{bmatrix} Q_{N\to c1} \\ Q_{N\to c2} \\ \vdots \\ Q_{N\to cK} \end{bmatrix}$$

..(eq.2-18)

Thus,

$$\begin{bmatrix} F_{c1}(t) \\ F_{c2}(t) \\ \vdots \\ F_{cK}(t) \end{bmatrix} = \sum_{m=1}^{N} P_{Q_m}\begin{bmatrix} Q_{m\to c1} \\ Q_{m\to c2} \\ \vdots \\ Q_{m\to cK} \end{bmatrix}$$

..(eq.2-19)

The parameter N is the total number of vertices in the system.

An alternate representation of this result is to assign one *K x N* matrix (the Q matrix) that contains all the data regarding vertices and their associated hardware elements. This matrix multiplied by the P vector of the state probabilities yields the desired result:

$$\begin{bmatrix} F_{c1}(t) \\ F_{c2}(t) \\ \vdots \\ F_{cK}(t) \end{bmatrix} = \begin{bmatrix} Q_{1\to c1} & Q_{2\to c1} & \cdots & Q_{N\to c1} \\ Q_{1\to c2} & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ Q_{1\to cK} & \cdots & \cdots & Q_{N\to cK} \end{bmatrix} \times \begin{bmatrix} P_{Q_1} \\ P_{Q_2} \\ \vdots \\ P_{Q_N} \end{bmatrix}$$

..(eq.2-20)

The example in figure 2-21 illustrates the concepts discussed.

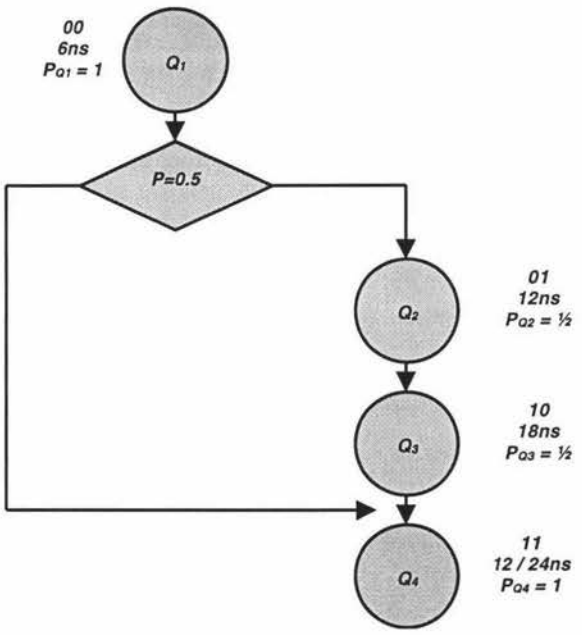


Figure 2-21: Example flow graph.

Let the branch have equal probability of branching to either path – 0.5. The hardware architecture is illustrated in figure 2-22. Each component of the architecture has been assigned a failure rate and a time taken for an operation to be completed in that unit.

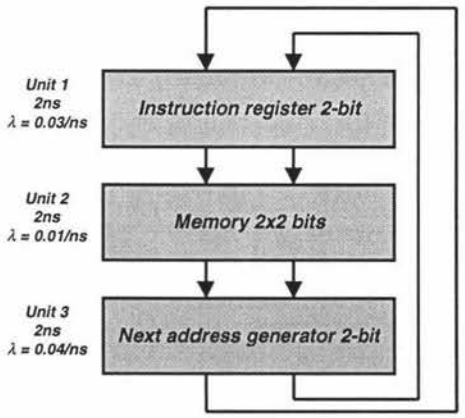


Figure 2-22: Example hardware architecture.

For example, the instruction register (component 1) takes 2ns to access the next memory address, and has a failure rate of 0.03 failures/ns. In addition, each vertex is assigned a probability, P_{Qn} (n=1, 2, …, N), which is the probability that the process will pass through that vertex during execution.

The table of time slices for each vertex is given in table 2-2:

Vertex	Time (ns)	Hardware	Failure Rate (/ns)	F(t)
Q_1				
	2	Instr reg	0.03	0.0582
	4	Memory	0.01	0.0392
	6	Next addr gen	0.04	0.2133
Q_2				
	8	Instr reg	0.03	0.2133
	10	Memory	0.01	0.0952
	12	Next addr gen	0.04	0.3812
Q_3				
	14	Instr reg	0.03	0.3430
	16	Memory	0.01	0.1479
	18	Next addr gen	0.04	0.5132
Q_4				
	8	Instr reg	0.03	0.2133
	10	Memory	0.01	0.0952
	12	Next addr gen	0.04	0.3812
	20	Instr reg	0.03	0.4512
	22	Memory	0.01	0.1975
	24	Next addr gen	0.04	0.6171

Table 2-2: Example process times.

Now, using eq. 2-20, the results are:

$$\begin{bmatrix} 0.6686 \\ 0.3071 \\ 0.1597 \end{bmatrix} = \begin{bmatrix} 0.0582 & 0.2133 & 0.3430 & 0.6645 \\ 0.0392 & 0.0952 & 0.1479 & 0.2927 \\ 0.2133 & 0.3812 & 0.5132 & 0.9983 \end{bmatrix} \times \begin{bmatrix} 1 \\ 0.5 \\ 0.5 \\ 0.5 \end{bmatrix}$$

The value 0.6686, for example, indicates that if a fault occurred during the execution of a process, there is a probability of 0.6686 that it happened in the instruction register. It is also necessary to recognise that the final column of the Q matrix is the sum of the two separate vertices leading to vertex Q_4, and the last element of the P matrix is therefore ½ rather than 1, even though the probability of reaching vertex Q_4 is 1.

2.2.6 Section Summary

The F, T, and D matrices introduced in equation 2-20 can be used to determine the system reliability characteristics, by taking into account the system flow graph, the system hardware, and the error detection capabilities of the checking mechanism used.

Reliability theory is used as the basis for estimating the parameters that affect the reliability of each individual vertex in a system flow graph. Furthermore, the vertex reliabilities are correlated to the underlying hardware of the system. New sets of parameters are introduced, which allow the representation of fault occurrences, and interactions between vertices. In addition, techniques are suggested that allow the decomposition and simplification of flow graphs and hardware devices to allow for easier calculation of the reliability parameters.

Once the reliabilities of vertices are identified, the next phase in determining the overall reliability of the system is to identify the possible erroneous transitions that may occur from each vertex. These erroneous transitions are also dependent on the type of error occurring in the vertex. These probabilities are represented in the T matrix, and Section 2.3 investigates parameter estimation for the matrix.

2.3 Representation of Fault Types

2.3.1 Introduction

This chapter investigates techniques for estimating the parameters of the T matrix, as defined by equation 2-10. Once the fault probabilities have been calculated (the F matrix), the next step in calculating system reliability is to determine the probabilities for each transition. This information is stored in the square T matrix, containing the probabilities for all illegal transitions – between any pair of vertices. Since the values in the T matrix reflect the probabilities of erroneous transitions, there are many different possible T matrices – dependent on the type of fault occurring. Thus, the T matrix gives the probabilities of erroneous transitions *of a particular error type*.

2.3.2 Representing Different Fault Types

2.3.2.1 Fault Types

In order to demonstrate how the fault characteristics of a system can be modelled, stuck-at faults are used as an example here. In reality, a variety of fault types could take place. They include single-bit and multiple-bit fault patterns, bridging faults, dynamic faults, and so on. Each hardware component of the control unit can described by its unique set of fault characteristics, such as the types of faults that occur, and the frequency with which each fault occurs. These characteristics could either be determined experimentally on real scaled-down systems, or by simulation. In addition, the occurrence of these faults would also be related to the failure probability at each vertex, which would be determined by the reliability of the underlying hardware.

In the previous chapter, the G matrix was used to indicate transition probabilities between vertices. The same notation can be used for specific faults as well. In

the case of individual fault types, many probabilities in the matrix will be zero. This is due to the fact that some transitions will not be possible, once the fault has rendered certain states inaccessible. In addition to the G matrix, the A vector is necessary. It contains the probability for each vertex that the process passes through it during execution. Finally, the vector B is also needed, which contains the failure probabilities of the process at each vertex. These probabilities are derived from the reliabilities of the individual hardware elements used by the vertex.

2.3.2.2 Fault-Oriented Representation

As an example, let us consider the flow graph illustrated in figure 2-21. A stuck-at zero fault in the least-significant address bit could occur in four situations: at any of the vertices. Figures 2-23 to 2-26 illustrate the transitions that could occur in the presence of such a fault, given that the fault occurs in a particular vertex. For example, figure 2-23 illustrates that, given the fault occurs while the process is in vertex Q_1, the only transitions possible are both erroneous – to vertices Q_1 and Q_3.

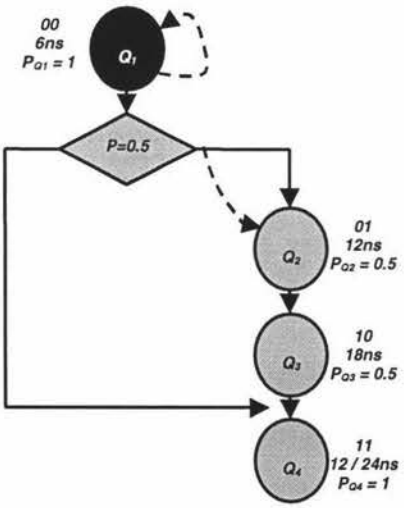


Figure 2-23: Fault at vertex Q_1.

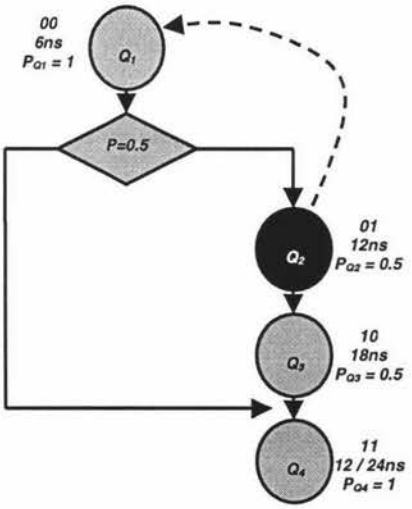


Figure 2-24: Fault at vertex Q_2.

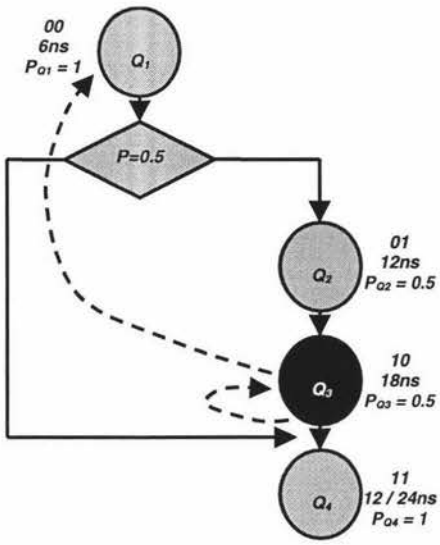


Figure 2-25: Fault at vertex Q_3.

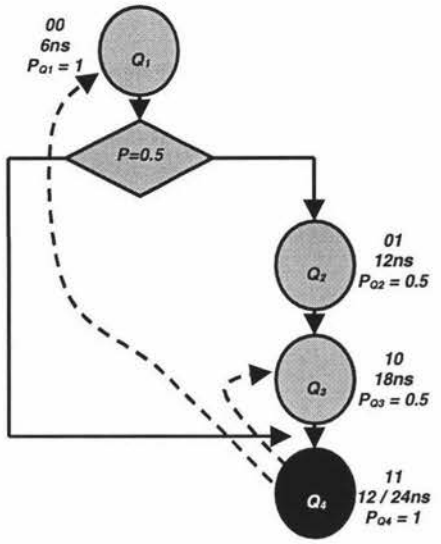


Figure 2-26: Fault at vertex Q_4.

In figures 2-23 to 2-26, the erroneous transitions are marked by the dashed lines. The erroneous transitions can be represented in the G matrix as:

$$G = \begin{bmatrix} p_{1-1} & 0 & p_{3-1} & 0 \\ p_{2-1} & 0 & 0 & 0 \\ p_{3-1} & 0 & p_{3-3} & 0 \\ p_{4-1} & 0 & p_{4-3} & 0 \end{bmatrix}$$

Using the A, B, and G matrices it is possible to acquire a square matrix, F, consisting of all the erroneous transition probabilities. Each element in F denotes the probability of an erroneous transition during a run of the algorithm, in the event that a stuck-at fault occurs. Since the matrix represents all the transitions of the system, for only stuck-at faults, the method can be called fault-oriented.

2.3.2.3 Vertex-Oriented Representation

An alternative to the fault-oriented representation is the vertex-oriented representation. In this case, the G matrix is replaced by the M matrix. In this matrix, information is provided for only one vertex. The M matrix is constructed as illustrated below:

$$M_{Q_i} = \begin{bmatrix} p_{fault1-Q1} & p_{fault1-Q2} & \cdots & p_{fault1-QN} \\ p_{fault2-Q1} & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ p_{faultK-Q1} & \cdots & \cdots & p_{faultk-QN} \end{bmatrix}$$

Here, M_{Qj} is the fault behaviour description of vertex Q_j. M is a $K \times N$ matrix, where K is the total number of faults modelled, and N is the total number of vertices in the flow-graph. Thus, element (2,1) of the matrix denotes the probability of an erroneous transition from Q_j to Q_1, in the presence of *fault type 2* – in a real implementation, this would be a particular type of fault.

Figures 2-27 to 2-30 show the possible transitions from Q_1, in the presence of stuck-at zero and stuck-at one faults, for the least and most significant bits:

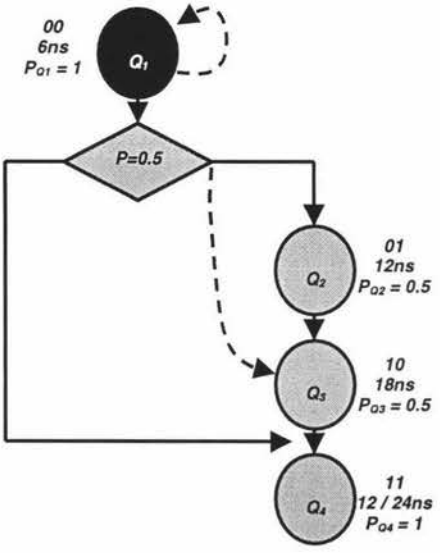


Figure 2-27: LSB stuck-at zero.

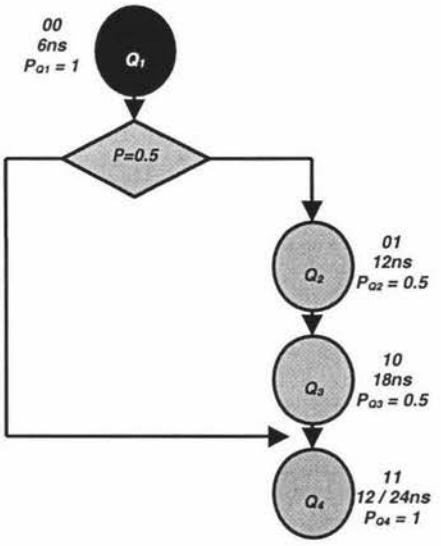


Figure 2-28: LSB stuck-at one.

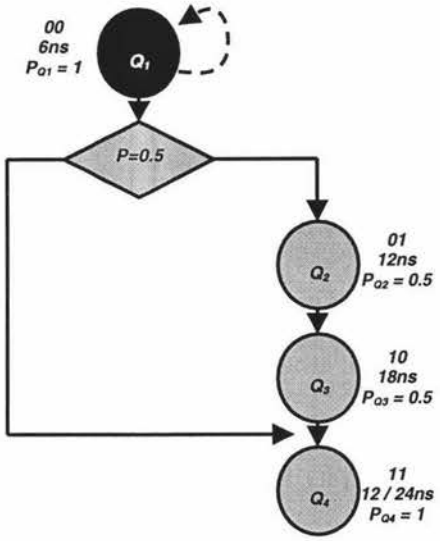


Figure 2-29: MSB stuck-at zero.

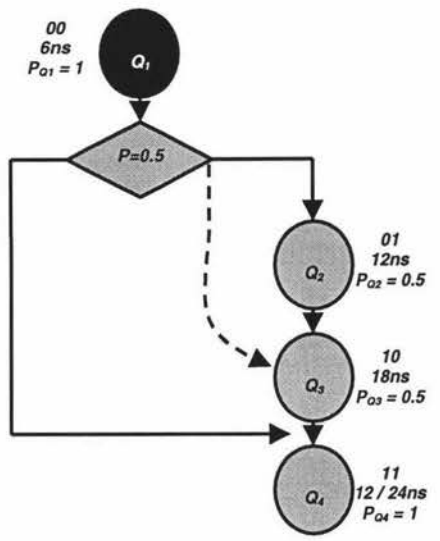


Figure 2-30: MSB stuck-at one.

The transitions illustrated in figures 2-27 to 2-30 can be represented in the M matrix as follows:

$$M_{Q_1} = \begin{bmatrix} p_{LSB\Rightarrow 0-1} & 0 & p_{LSB\Rightarrow 0-3} & 0 \\ 0 & p_{LSB\Rightarrow 1-2} & 0 & 0 \\ p_{MSB\Rightarrow 0-1} & 0 & 0 & 0 \\ 0 & 0 & p_{MSB\Rightarrow 1-3} & 0 \end{bmatrix}$$

Using the M matrix, it is possible to acquire the probabilities of each possible faulty transition from Q_1, in the event of any of the fault types occurring:

$$P_{(fault-type)Q_1} = M_{Q_1} \times p_{Q_1} \times F_{Q_1}(t),$$

where p_{Q1} is the probability of the process passing through Q_1, and $F_{Q1}(t)$ is the failure probability of vertex Q1. In general,

$$P_{(fault-type)Q_i} = M_{Q_i} \times p_{Q_i} \times F_{Q_i}(t)$$

..(eq.2-21)

This notation is vertex-oriented, since it represents all the possible transitions for all the fault types, pertaining to one vertex.

2.3.3 Section Summary

This chapter focused on the transition matrix (T matrix) from equation 2-10. The T matrix is used to describe the behaviour of the flow graph with respect to erroneous transitions.

Examples are given showing the effects of various faults on the execution path through the flow graph, and suggestions made regarding how to represent different fault types. The values in the T matrix change with each error type. In addition, the values can also change depending on the location and time of the error in the execution of the process. For each error that occurs, erroneous transitions are possible to illegal states – states that should not be jumped to in error-free execution. Here, too, the state that the error occurs in and the time of its occurrence determine which illegal states the process could possibly jump to.

Identifying the full set of possible erroneous transitions a) for each error type; and b) from errors in each vertex; are necessary steps in forming a complete model of the behaviour of the system in the presence of errors.

Section 2.4 examines the final element of the system characterisation – the error detection capabilities, by investigating the performance of several error coding techniques in the presence of errors.

2.4 Fault Coverage

2.4.1 Introduction

In section 2.1 the major parameters required to determine system reliability were introduced, in eq.2-10. The three matrices involved were the F, T, and D matrices. Two of these, the F and T matrices, were dealt with in the previous chapter which described techniques to acquire the system fault probabilities. In this chapter the final parameter, the D matrix, is investigated. This matrix contains the error coverage data, and typically contains ones and zeros which denote 'error coverage' and 'no error coverage' respectively. Error detection [27,28,36] is discussed first, with results from experiments on different error detection techniques presented in Appendix 2.

This series of experiments examines the self-checking technique based on signature analysis [2, 11, 12, 31]. Each step in the process is assigned a key (a bit pattern), and keys are concatenated as the process executes. After a certain number of steps a key sequence is formed, and this key sequence is fed into a feedback shift register, and a signature is produced [2,11]. This signature can be compared to a predetermined signature that is known to be correct. If the signatures are different, an error has certainly occurred. If the signatures are the same, either no error has occurred, or an error has occurred with its effect being masked. The percentage of errors that can be detected is known as the fault coverage. In the simulations, several error detection methods are compared to signature analysis.

Overall fault coverages can be calculated for key sequences by introducing errors into the key sequence, and using an error detection method (such as signature compression) to check whether the error is detected, or its effect masked. This can be done with respect to the whole key sequence, but to scrutinise the effects

of the error detection, the fault coverage can be analysed with respect to each individual bit of the key. In signature analysis, there are four major parameters that influence the fault coverage – the length of the compression polynomial, the configuration of the polynomial, the number of bits in error, and the positions of the erroneous bits in the key sequence.

The length and type of the compression polynomial generally determines the fault coverages acquired. As stated earlier, fault coverage is also influenced by the number and positions of the erroneous bits.

2.4.2 The Signature Compression Process

In general, the signature compression process can be modelled as follows:

Figure 2-31: Signature generation process.

The erroneous key can be represented as the superposition of two keys - the error bit stream, and the original sequence (figure 2-32). Here, the term key corresponds to the sequence of the individual check keys acquired from the section of the flow graph being checked.

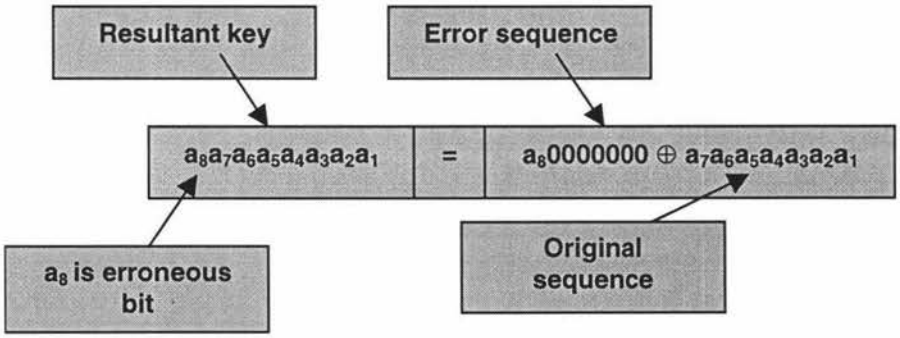


Figure 2-32: Representation of keys as a superposition of multiple bit sequences – error is in the MSB position of the original key.

The signatures of the error bit stream and the original sequence may be added in a modulo-2 manner to produce the signature of the resultant key, if the error sequence produces a signature that is '0' (figure 2-33).

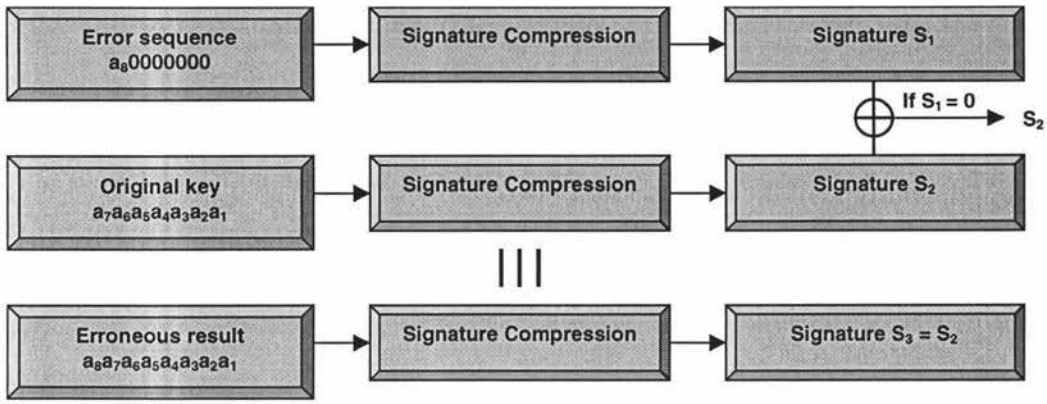


Figure 2-33: Adding signatures of two streams to produce the signature of the third stream.

The underlying theory of signature analysis is closely related to that of M-sequences, in that it is based on the use of primitive, irreducible generating polynomials [36]. Mathematically, the concept is based on polynomial division over GF(2) (a finite field comprises of a finite number of elements. A field of q elements is called a Galois Field and is denoted by GF(q)) [36]. The dividend is the check key sequence produced by the control unit being tested, and it can be represented as a polynomial p(x) of degree k-1, where k is the stream length. For example, sequence 10011 has the polynomial form $p(x)=x^4 \oplus x \oplus 1$. The divisor is a primitive irreducible polynomial $\psi(x)$, and division results in a quotient q(x), and remainder s(x) which are related by:

$$p(x) = q(x)\psi(x) \oplus s(x)$$

..(eq.2-22)

The remainder s(x), which is a polynomial of the same or lower degree than $\psi(x)$, is the signature. In long division form, the signature is obtained thus:

$$\psi^{-1}(x)\overline{)p(x)} = q(x) rem\ s(x)$$

..(eq.2-23)

2.4.3 Bit Errors at the Beginning of the Key

When errors occur in the keys, the positions of the erroneous bits influence the coverage provided by means of signature analysis. Also, higher-order signature compression polynomials generally give more extensive fault coverages.

Fault coverage provided by signature analysis has been studied and presented in numerous publications. The types of errors considered in these publications are usually those where bit values are changed within key sequences, while the key length remains unchanged. In the case of control units, however, the key length can also be affected. In this section, the term 'error' relates not only to bit-altering errors, but also to errors that change the length of the key sequence.

For single-bit enlargement and shortening errors, we can first consider the situation where the first bit is in error (first bit deleted, or erroneous bit inserted in the first position of the bit stream). This can be represented as shown in figure 2-34:

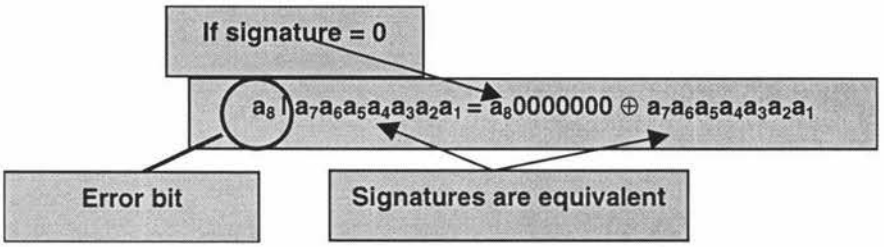


Figure 2-34: Single-bit error at the beginning of the key.

In this case, the erroneous bit may take on one of two values, '0' or '1'. In the experiments, it is assumed that both states occur with equal probability. The erroneous bit will also be the first bit fed into the shift register, which is initialised to zeros before the compression process. If the bit is a zero, it will simply be shifted in without changing the contents of the shift register. Thus, in all the cases when the bit error is a '0', the error will be masked. If the error is '1' instead, the contents of the shift register would have been changed. Thus, in this particular error type, half the errors are masked, and the coverage for the first bit is 50%. A

2-bit error may now be considered. If, for example, the erroneous bits are at the beginning of the stream, then the error can be classified as follows:

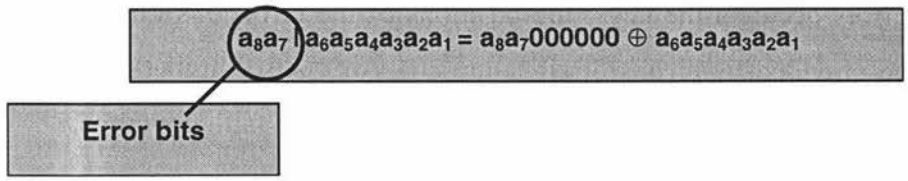


Figure 2-35: 2-bit error at the beginning of the key.

In this case, the error can be considered as a pair; the possible values for the error are: '00', '01', '10', and '11'. As in the case of the single-bit error, if zeros are fed into the shift register at the beginning of the compression process, the contents of the register remains unchanged. Thus, in the case of the first alternative, '00', the error will not affect the resultant signature after the compression process. In the other three alternatives, the presence of a '1' will change the contents of the shift register. Since '00' occurs 25% of the time, 25% of the errors in this position will go unnoticed. Thus, the fault coverage for this error type is 75%.

In considering 3-bit errors, the error can be classified as a superposition of four bit streams. In this case, the error may have one of eight values: '000', '001', '010', '011', '100', '101', '110', and '111'. The value of '000' is the only error that can be masked, and since this combination is 1 out of 8 equally probable combinations, it occurs 12.5% of the time. Thus, the fault coverage is 87.5% for this error type. The remaining bits provide a higher fault coverage. An additional consideration has to be taken into account in the case of 3-bit errors. For 1- and 2-bit errors the signatures of the separate error bit streams are always unique, when they are combined to form the error signature. Therefore, it is always possible to detect errors if the individual signatures are recognised as products of erroneous bit streams. In the case of 3-bit errors, however, the superposition of the individual signatures may produce the signature of the original correct bit stream. In other words, the situation in figure 2-36 is possible.

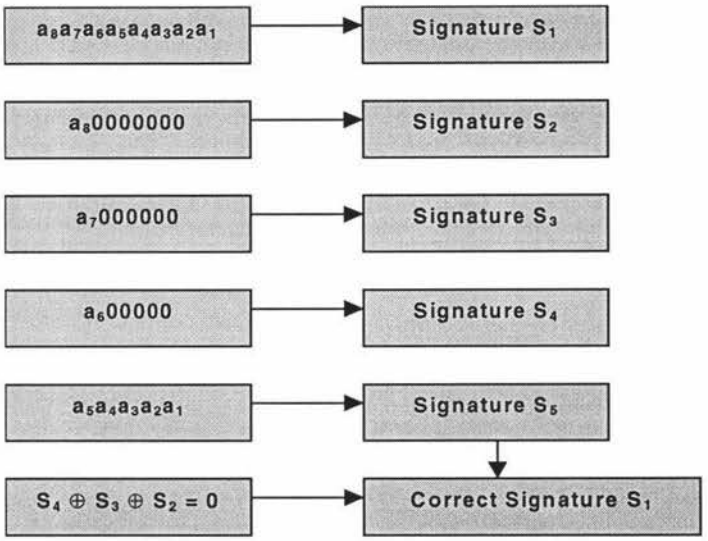


Figure 2-35: Error signatures combining to give correct signature.

2.4.4 Bit Errors – The General Case

The examples given previously dealt with errors where the key was lengthened by the addition of error bits onto the beginning of the key. It is, of course, possible for bits to be inserted anywhere in the key. The fault coverages acquired in these cases, however, can theoretically be determined using the same logic. Figure 2-37 below illustrates the insertion of an erroneous bit between the first and second bits of an 8-bit key. Once again, the keys can be treated as the combination of multiple sequences.

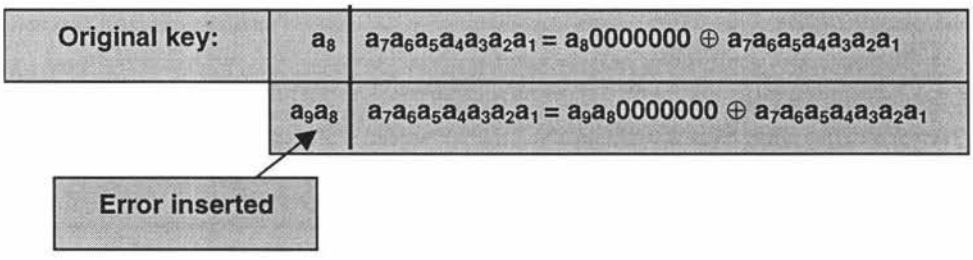


Figure 2-36: Error inserted between bits in key.

The original 8-bit key has now become a 9-bit sequence. Note, also, that the MSB of the original key a_8 is denoted as a_9 in the erroneous sequence. In this case, if the MSB were a '1' (50% probability), then the error will be detected

regardless of what the error bit is, since the '1' will first be shifted into the shift register, and the compression process would have begun. On the other hand, the MSB could be a '0'. In this case, if the error bit is a '1', the error will be detected, but not if the error bit is a '0'. Therefore, the only circumstance under which the error is masked is when both the MSB and the error bit are '0'. In this situation both bits are shifted into the register without changing its contents, and the compression begins only after the error bit has passed.

As another example, let us consider a 2-bit insertion into an 8-bit key, where the error bits are inserted between the 4^{th} and 5^{th} bits of the key. The resultant sequence is now 10 bits long. As in the previous example, the error will not be detected if all of the first four bits of the key are '0', and the error bits are also '0'. In addition, the error can also go undetected if the configuration of the error bits and the bits of the key allow masking of the error. For the error to be masked, the signature analyser must eventually produce the same output it would have produced, had the error not occurred. In effect, the presence of the error is then overlooked. In the cases of 1- and 2-bit errors, however, it is not possible for the shift register to return to its original state. In the case of a PRBS (pseudo-random binary sequence) [36] generator with a non-zero input (such as a signature analyser), the changing states of the shift register can be represented as a state machine (figure 2-38). The state machine could theoretically start in any of its states, as long as it has the correct seed (initial conditions) to start in that state – all the registers containing zero to begin with is a convenient initial state.

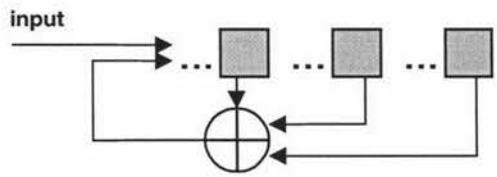


Figure 2-37: Example shift register with input.

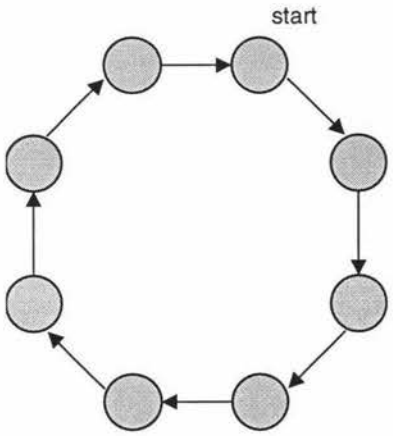


Figure 2-38: Set of states for the shift register in figure 2-37.

Figures 2-39 and 2-40 illustrate the concept of shift registers changing states, with an error present. The state that the shift register should be in without errors, and the state that it jumps to because of the error, can both be represented as separate shift registers with their own seeds (figure 2-41). The combined outputs of the two registers is the resultant sequence.

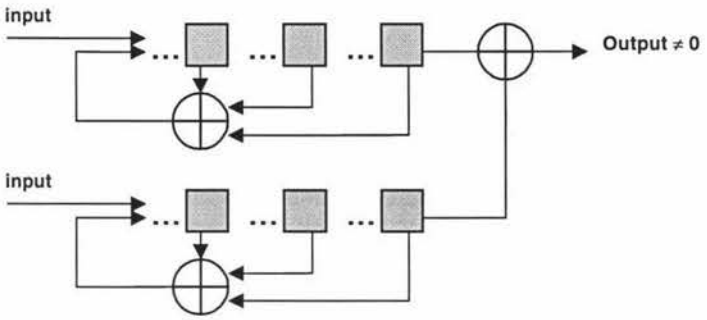


Figure 2-39: Erroneous sequence represented as the combined outputs of separate registers.

The altered path of the state transitions can be illustrated as shown below, with the erroneous states in black, and the erroneous transitions in dashed lines.

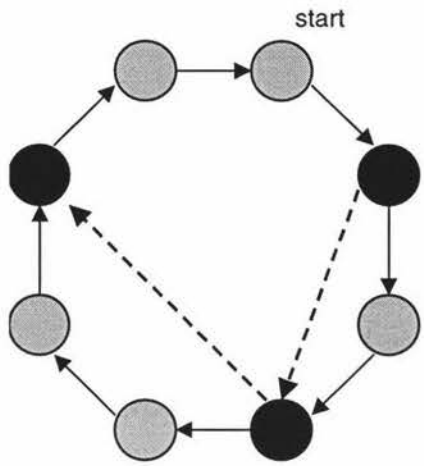


Figure 2-40: Erroneous state transitions due to errors.

In the case of 3-bit errors, a similar problem arises with introducing errors into the key, as with errors before the MSB. There are permutations in which three error bits may mask the overall effect of the erroneous key, by falsely producing the same signature as the original key. Due to this phenomenon, many 3-bit errors will be masked in the signature verification process. The figures below illustrate this concept:

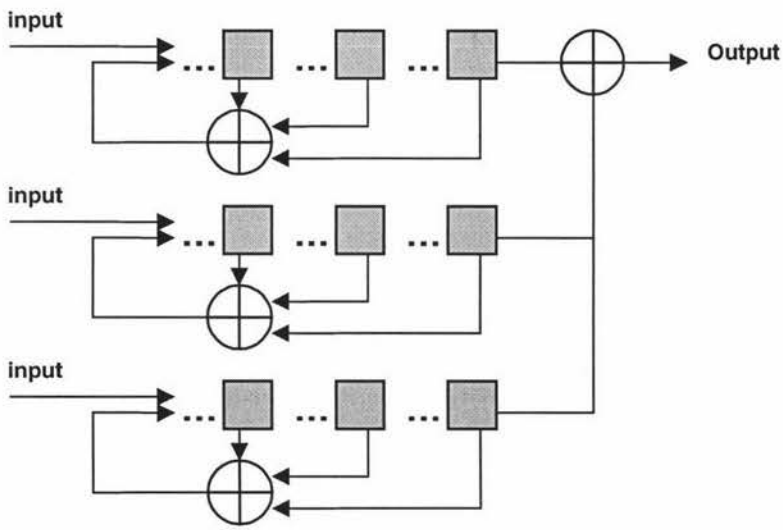


Figure 2-41: 3-bit error represented as the sum of separate sequences.

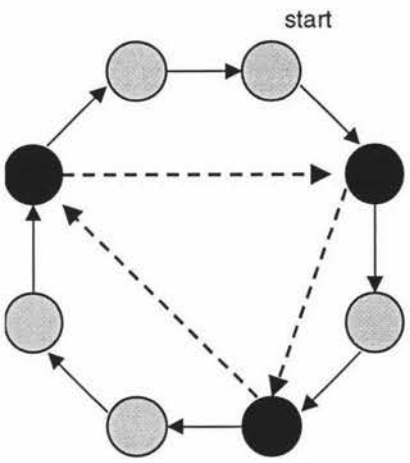


Figure 2-42: Errors are masked due to register returning to its original state.

Numerous techniques are available, whereby keys can be checked for errors they may have incurred. The simulations [37-39] in this chapter explore the fault coverages achievable using some of the more common error checking mechanisms. In particular, the simulations highlight the differences between the error checking mechanisms, and the technique of special interest - signature analysis.

The simulations were written in MATLAB [40]. Code was created for generating keys, and in order to gain conclusive results all possible combinations of keys were generated for the selected key length (so, for 8-bit keys $2^8 = 256$ keys were generated). For each key, errors were introduced in all the possible combinations for the number of errors inserted (so, for 8-bit keys and 2-bit errors, each of the 256 keys was subjected to ${}^8C_2$ error combinations). Modules were also written to incorporate various error coding techniques into the erroneous sequences generated, and their correct counterparts. The MATLAB code for the simulations is provided in Appendix 1.

For key shortening, the simulations were run using 8-bit and 12-bit keys. 1, 2, and 3 bits were deleted from the 8-bit keys, and 1 to 5 bits for the 12-bit keys. With 8-bit keys, for example, there were a total of 2^8 = 256 possible keys. All

the possible key combinations were generated, and the fault coverages recorded for each position of the bit deletion.

For 1-bit deletion, there were simply 8 positions where the bits could be deleted. For 2-bit deletions, the bits deleted were 1 and 2, 2 and 3, 3 and 4, and so on. Similarly, with 3-bit deletions, the bits deleted were 1, 2, and 3; 2, 3, and 4; and so on.

The simulation was performed for 8-bit keys, using 4-bit compression. The keys were lengthened from 1-3 bits. The graphs provided show the overall coverages for each bit position, and individual contributions from each error combination are also provided for 1- and 2-bit lengthening. Nine positions are used to insert errors into, as illustrated below:

Combination 1: xyz $a_8a_7a_6a_5a_4a_3a_2a_1$

Combination 2: a_8 xyz $a_7a_6a_5a_4a_3a_2a_1$

:

:

Combination 9: $a_8a_7a_6a_5a_4a_3a_2a_1$ xyz

2.4.5 Discussion of Simulation Results

In the case of parity checking, the coverage obtained is uniform across all bit positions. In addition, the coverage is also uniform regardless of the number of bits deleted. This is due to the fact that, essentially, any set of deletions is symmetrical in nature – in one bit deletions either a '0' or a '1' may be deleted with equal probability, and thus 50% of the errors are detected; for three bit errors, two '1's may be deleted and one '0', or vice versa – both occur with equal probability. When no '1's or two '1's are deleted, the error goes undetected, whereas if one or three '1's are deleted, the error is detected. Thus, regardless of the deletion pattern, 50% of deletions are always detected.

In the case of lengthening, the coverage obtained is uniform regardless of bit position, once again. The coverage is also uniform regardless of the number of bits inserted. As with shortening, this is because '1's and '0's occur with equal probability. Only odd numbers of errors are detected.

When syndrome coding is used for sequence compression, a direct correlation can be made between the fault coverage and the number of bits deleted. For one bit deletion (for example, '0's and '1's are deleted with equal probability) a bit is counted as missing, 50% of the time. In the case of 2-bit deletions, three out of four error cases are detectable, since they contain '1's, and only the '00' case cannot be detected. Thus, 75% of the faults are detected. The position of the error in the key is not significant here, since a missing '1' can be counted anywhere in the key, and therefore fault coverages are uniform for each set of deletions. For example, in the case of 3-bit insertions, seven out of eight error cases are detectable, since they contain '1's, and only the '000' case cannot be detected. Thus, 87.5% (⅞) of the faults are detected. Shortening and lengthening produce similar results.

In the case of transition counting for shortened keys, both, the positions of the bits deleted, and the number of bits deleted, are important in achieving the fault coverage required.

For example, 1-bit deletion of the first bit yields a coverage of 50%. This is reasonable, since the first bit always has a 50% chance of being the same as the second bit. These cases are when they are both '0', or both '1'. In both these cases, there is no transition between the first and second bit, and so this makes no contribution to the transition detection process. Thus, deletion of the first bit makes no difference, and produces an undetected error.

If deletion of the second bit is now considered, it can be seen that the situation is more complex than that of the first bit. In this case, there are bits on either side of the deleted bit. Thus, the interaction of the deleted bits with both its neighbours must be considered. Since three bits are involved, the problem can be solved by considering how many bit patterns there can be (eight in this case), and determining which of these bit patterns can produce undetectable errors. Table 2-3 below illustrates this procedure:

Bit Position			Transition Detector Result	
1	2(Deleted Bit)	3	Before Deletion	After Deletion
0	0	0	0	0
0	0	1	1	1
0	1	0	2	0(detected)
0	1	1	1	1
1	0	0	1	1
1	0	1	2	0(detected)
1	1	0	1	1
1	1	1	0	0

Table 2-3: 1-Bit deletion of 2nd bit (general case is any non-end bit, i.e. any bits other than MSB or LSB).

From the table it can be seen that the combinations '101' and '010' are detected – these are two out of eight combinations, and thus 25%.

For the deletion of two bits, there are sixteen possible combinations, of which eight yield detectable error patterns. This is due to the fact that the two bits being deleted must be taken into consideration, as well as the two bits on either side of the deleted bits – four bits and sixteen combinations in total. 50% of these errors are detectable in the non-end bits of the key. In the case of the MSB and LSB, three bits are taken into consideration. In these cases, two of the three bits are going to be deleted, so the result of the transition detection must originally have been zero. There are only two cases where the result of transition detection is zero, before and after deletion – when all three bits are the same: '000' and '111'. These are the undetectable cases, and account for two out of eight combinations. Thus, the fault coverage is for six out of eight cases – 75%.

Similarly, with lengthened keys, both the position of insertion of the bit, and the number of bits inserted are important, and influence the fault coverage.

For example, in a 1-bit insertion, inserting before the first bit yields a coverage of 50%. This is logical, since '0' and '1' will be inserted, and the first bit of the key has to be one of these two possibilities. In the case that the inserted bit is the same as the first bit, there is no transition between the error bit and the first bit, and so this makes no contribution to the transition detection process. Thus, insertion before the first bit makes no difference, and produces an undetected error.

If insertion between the first and the second bit is now considered, it can be seen that the situation is more complex than that of the insertion before the first bit. In this case, there are bits on either side of the inserted bit, and so the interaction of the error bit with both its neighbours must be considered. Since three bits are involved, the problem can be solved by considering how many bit patterns there can be (eight in this case), and determining which of these bit patterns can produce undetectable errors. The table below illustrates this procedure:

Bit Position			*Transition Detector Result*	
1	*2(Inserted Bit)*	*3*	*Before Insertion*	*After Insertion*
0	*0*	*0*	*0*	*0*
0	*0*	*1*	*1*	*1*
0	*1*	*0*	*0*	*2(detected)*
0	*1*	*1*	*1*	*1*
1	*0*	*0*	*1*	*1*
1	*0*	*1*	*0*	*2(detected)*
1	*1*	*0*	*1*	*1*
1	*1*	*1*	*0*	*0*

Table 2-4: 1-Bit insertion between 1st and 2nd bits (general case is anywhere within the key, i.e. anywhere other than the beginning or the end of the key).

From the table it can be seen that the combinations '101' and '010' are detected – these are two out of eight combinations, thus producing a 25% coverage.

For the insertion of two bits inside the key, there are sixteen possible combinations, of which eight yield detectable error patterns. This is due to the fact that the two bits being inserted must be taken into consideration, as well as the two bits on either side of the error bits – four bits and sixteen combinations in total. 50% of these errors are detectable in the non-end bits of the key. In the case of attaching error bits to the beginning or end of the key, three bits are taken into consideration. In these cases, two of the three bits are going to be added as error bits, so the result of the transition detection must be zero. There are only two cases where the result of transition detection are zero before and after insertion – when all three bits are the same: '000' and '111'. These are the undetectable cases, and account for two out of eight combinations. Thus, the fault coverage is for six out of eight cases – 75%.

2.4.6 Fault Coverage – Comparative Analysis

Comparing the various error detection techniques with exactly the same keys and errors provides a reliable basis on which to draw conclusions about their relative advantages and disadvantages.

In the case of parity checking, the uniform fault coverage across all bit positions provides a useful technique to be employed if the error probabilities are also considered to be uniform. In general, however, the coverage tends to be lower than with the other techniques, given the same parameters. The advantage gained in using parity checking is that, provided the fault coverage achievable is adequate, little space is needed to store the error checking information. Thus, if memory is severely limited, parity checking may be the only feasible option.

Syndrome coding provides the same advantage as parity checking - uniform fault coverage. In this case, however, the fault coverage increases with the number of erroneous bits – with parity checking the fault coverage remained constant. Syndrome coding is therefore useful if the number of errors occurring in a key is

of significance. Since the coverage remains constant across the whole key, error probabilities for each bit position can still be ignored.

Transition counting provides the advantage of uniform fault coverage across all bit positions in general. The fault coverage also tends to increase with the number of errors introduced into the key. Thus, this is also a useful technique if the number of errors introduced is a significant parameter. An additional advantage to transition counting is its tendency to be more sensitive to errors occurring at the beginning or end of the bit stream. Considerably higher fault coverages are achieved at the ends of the bit stream. Thus, this technique is useful if an error type is present which tends to strike at the beginning or end of a series of bits.

In comparison to the techniques described above, signature analysis provides the advantage of increasing fault coverage as the bit stream progresses. In addition, fault coverage also increases with the number of error bits in the key. Further parameters that can be varied in signature analysis are the degree and configuration of the compression polynomial used. The higher the degree of the polynomial, the higher the fault coverage achieved. The degree of the compression polynomial also dictates the amount of memory used for storing the error detection information.

The comparisons above are stated as generalisations, in that they reflect the overall fault coverages from all keys. This is so, since all possible combinations of the keys were generated during the simulations. In practice, it is possible to further improve fault coverage by considering the keys assigned to each vertex of the process flow graph. These keys could be assigned in such a way as to aid in the detection of errors, and minimise masking of errors. Figures 2-43, 2-44, and 2-45 are examples of key assignments to vertices of a flow graph. The graph is a simple one, but is nevertheless useful in illustrating the concept of key assigning.

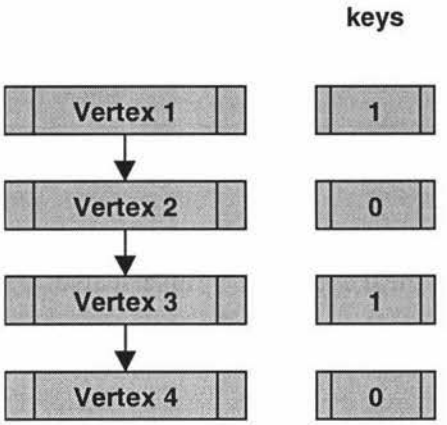


Figure 2-43: Flow graph with alternating 1-bit keys.

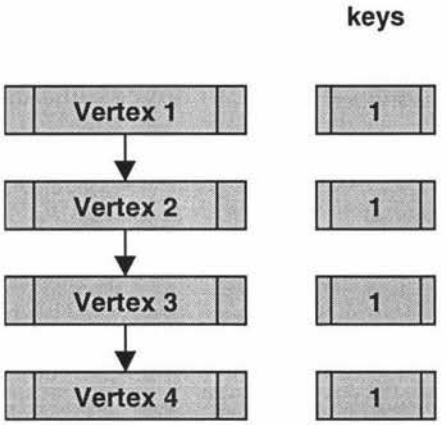


Figure 2-44: Flow graph with identical 1-bit keys.

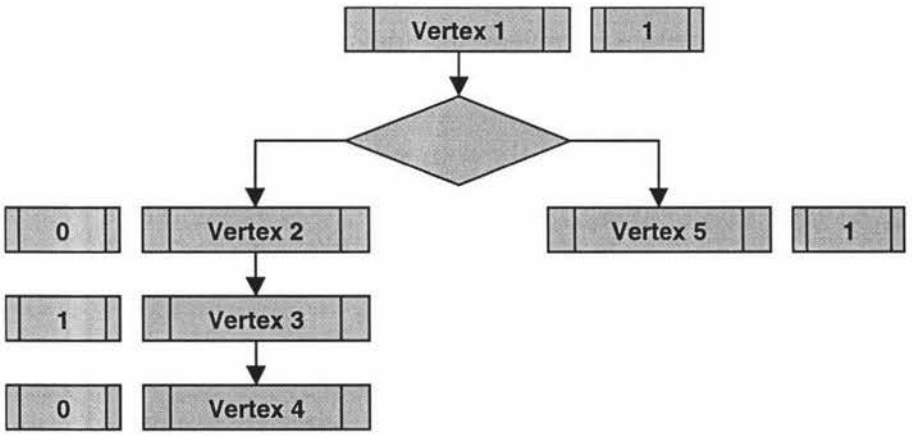


Figure 2-45: A more complex graph – key assignment is not trivial.

In the case of figure 2-43, the alternating '1's and '0's provide favourable conditions for error checking using some of the techniques. Parity checking would still yield 50% coverages, since deletion or insertion of '0's would not be detected. Deletion or insertion of two '1's would also go undetected. Syndrome coding, too, would only provide an 'average' fault coverage for 1-bit errors. Deletion or insertion of any number of '0's would be completely unnoticed.

The key assignments in figure 2-43 are advantageous if transition counting or signature analysis can be used. Deletion of any of the bits would immediately result in the loss of two transitions per deletion. Similarly, insertions would also generally result in additional transitions which would indicate errors. Signature analysis would certainly be useful in this case, as most errors would be detected. In addition, signature analysis would also be useful because there is a probability for any number of bit errors to be detected.

Parity coding could certainly be used for the graph in figure 2-44, as any odd-number of bit errors would be detected. Syndrome coding would provide excellent coverage, and would be an ideal choice in this case. The presence of only '1's guarantees that addition or removal of any number of bits would immediately be detected. This technique would also be useful if the exact path of the process is not important. If the result from the syndrome coding can simply be checked to belong to a number of acceptable path lengths, the method could certainly prove useful.

The key assignments in figure 2-44 are completely unfavourable to transition counting. Any number of deletions would not be picked up, since there would still be no transitions. Even insertion of any number of '1's would be undetected. The only situation in which transition counting would be useful would be in the addition of '0's.

As in the key assignments in figure 2-43, the key arrangements in figure 2-44 could be used with signature analysis. The technique would certainly provide higher fault coverage than parity checking and transition counting. For this particular case, however, syndrome coding allows the best fault coverage. In addition, it would be a simpler and faster technique to implement than signature analysis.

Figure 2-45 illustrates a general, more realistic graph. In such a case, two courses of action are possible. Firstly, if the system has already been completely implemented, signature analysis provides the best chance of achieving a high fault coverage. Even in the case of a large and complex system, signature analysis at least provides the advantage that its complexity can be increased by using higher degree polynomials. This would allow better fault coverage by reducing aliasing. Without knowledge of the system's structure, it would be difficult to determine whether the other techniques would provide good fault coverages.

The alternative is to have in-depth knowledge of the system during its implementation. If this is possible, key assignment could be done in such a way as to provide maximum fault coverage, and be favourable to the error detection method to be used. Thus, the error detection technique could be selected with prior knowledge of the system. This is the most favourable scenario, as it allows maximum flexibility in incorporating self-test technology in the system.

Elements of the discussed fault detection techniques were implemented in the image processing program of the mobile robot that will be discussed in Chapter 3.

2.4.7 Section Summary

Once the vertex fault probabilities and transition probabilities have been calculated, the final step in designing self-checking is to implement an appropriate error detection technique. Many issues have to be considered in selecting the

error coding technique – the memory available, the processing speed available, the flexibility of the system (i.e. how much can the existing code be changed), and so on. In order to select the most efficient error coding method, knowledge of the expected error types is very useful.

In order to gain further insight into the error coverages provided by different error coding techniques, several of them have been tested with an error type that cannot always easily be detected – shortening and lengthening of the key sequences. The key sequences were corrupted with all the possible error combinations, and the same corrupted sequences used for each error coding technique. This allows an accurate comparison of the methods.

2.5 References

[1] **Patterson & Hennessy**; *Computer Organization and Design 2nd Ed.*; Morgan Kaufman Publishers 1998

[2] **Demidenko, S.N., Levin, E.M., and Lever, K.V.**; *Concurrent self-checking for microprogrammed control units: an analytical survey*; IEE Proceedings-E, Nov. 1991, Vol.138, No.6

[3] **Parekhji, R.A., Venkatesh, G., and Sherlekar, S.D.**; *Concurrent Error Detection Using Monitoring Machines*; IEEE Design and Test of Computers, 1995

[4] **Parekhji, R.A., Venkatesh, G., and Sherlekar, S.D.**; *Monitoring Machine Based Synthesis Technique for Concurrent Error Detection in Finite State Machines*; Journal of Electronic Testing: Theory and Applications (JETTA), Vol. 8, No. 2, April 1996

[5] **Mahmood, A., and McCluskey, E.J.**; *Concurrent Error Detection Using Watchdog Processors - A Survey*; IEEE Transactions on Computers, Feb.1988, Vol.37, No.2

[6] **Upadhyaya, S.J., and Ramamurthy, B.**; *Concurrent Process Monitoring with No Reference Signatures*; IEEE Transactions on Computers, April 1994, Vol.43, No.4

[7] **Nicolaidis, M, and Anghel, L.**; *Concurrent Checking for VLSI*; Microelectronics Engineering, No. 49, pp. 139-156, 1999

[8] **Nicolaidis, M., and Zorian, Y.**; *On-Line Testing for VLSI-A Compendium of Approaches*; Journal of Electronic Testing: Theory and Applications (JETTA), Vol. 12, No. 1 / 2, Feb/April 1998

[9] **Stiffler, J.J.**; *On-Line Fault Monitoring*; Journal of Electronic Testing: Theory and Applications (JETTA), Vol. 12, No. 1 / 2, Feb/April 1998

[10] **Das, D., and Touba, N.A.**; *Synthesis of Circuits with Low-Cost Concurrent Error Detection Based on Bose-Lin Codes*; Journal of

Electronic Testing: Theory and Applications (JETTA), Vol. 15, No. 1 / 2, Aug/Oct 1999

[11] **Demidenko, S.N., Levin, E.M., and Lever, K.V.**; *New approach to synthesis of self-checking microprogrammed control units with specified fault-detection probabilities*; IEE Proceedings-E, Nov. 1991, Vol. 138, No. 6

[12] **Demidenko, S.N., Levine, E.M., and Piuri, V.**; *Synthesis of On-Line Testing Control Units: Flow Graph Coding/Monitoring Approach*; IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, IEEE Computer Society, Proceedings 2000

[13] **Ling, Y., Mi, J., & Lin, X.**; *A Variational Calculus Approach to Optimal Checkpoint Placement*; IEEE Transactions on Computers, July 2001, Vol. 50, No. 7

[14] **Piuri, V., Antola, A., Ferrandi, F., & Sami, M.**; *Semiconcurrent Error Detection in Data Paths*; Vol. 50, No. 5, May 2001, IEEE Transactions on Computing

[15] **Tsuda, N.**; *Fault Tolerant Processor Arrays Using Additional Bypass Linking Allocated by Graph-Node Coloring*; Vol. 49, No.5, May 2000, IEEE Transactions on Computers

[16] **Nicolaidis, M.**; *On-line testing for VLSI: state of the art and trends*; Integration: the VLSI journal 26, 1998

[17] **Shen, L., & Su, S.**; *A Functional Testing Method for Microprocessors*; IEEE Transactions on Computers, Vol. 37, No. 10, October 1988

[18] **Edirisooriya, G., & Robinson, J.**; *Aliasing Properties of Circular MISRs*; Journal of Electronic Testing: Theory and Applications (JETTA), Vol. 4, No. 2, May 1993

[19] **Cooray, N., & Czeck, E.**; *Guaranteed Fault Detection Sequences for Single Transition Faults in Finite State Machine Models Using Concurrent Fault Simulation*; Journal of Electronic Testing: Theory and Applications (JETTA), Vol. 8, No. 3, June 1996

[20] **Nicolaidis, M.**; *Efficient UBIST Implementation for Microprocessor Sequencing Parts*; Journal of Electronic Testing: Theory and Applications (JETTA), Vol. 6, No. 3, June 1995

[21] **Hellebrand, S., Wunderlich, H., & Hertwig, A.**; *Mixed-Mode BIST Using Embedded Processors*; Journal of Electronic Testing: Theory and Applications (JETTA), Vol. 12, No. 1 / 2, Feb./Apr. 1998

[22] **Mukherjee, N., Karri, R., & Chakraborty, T.**; *Built-In Self Test: A Complete Test Solution for Telecommunication Systems*; IEEE Communications Magazine, June 1999

[23] **Nicolici, N., Al-Hashimi, B., Brown, A., & Williams, A.**; *BIST Hardware Synthesis for RTL Data Paths Based on Test Compatibility Classes*; IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems; Vol. 19, No. 11, Nov. 2000

[24] **Gupta, S., Pradhan, D.**; *Utilization of On-Line (Concurrent) Checkers during Built-In Self-Test and Vice Versa*; IEEE Transactions on Computers, Vol. 45, No. 1, Jan. 1996

[25] **Agrawal, P.**; *Fault Tolerance in Multiprocessor Systems Without Dedicated Redundancy*; IEEE Transactions on Computers, Vol. 37, No. 3, Mar. 1988

[26] **Damiani, M., Olivo, P., Favalli, M., & Riccò, B.**; *An Analytical Model for the Aliasing Probability in Signature Analysis Testing*; IEEE Transactions on Computer-Aided Design, Vol. 8, No. 11, Nov. 1989

[27] **Spragins, J.**; *Telecommunications: Protocols and Design*; Addison Wesley, 1991

[28] **Glover, I., & Grant, P.**; *Digital Communications*; Prentice Hall Europe 1998

[29] **Bezerra, E., Vargas, F., & Gough, M.**; *Improving Reconfigurable Systems Reliability by Combining Periodical Test and Redundancy Techniques: A Case Study*; Journal of Electronic Testing: Theory and Applications (JETTA), Vol. 17, No. 2, Apr. 2001

[30] **Talkhan, E., Ahmed, A., & Salama, A.**; *Microprocessor Functional Testing Techniques*; IEEE Transactions on Computer-Aided Design, Vol. 8, No.3, Mar. 1989

[31] **Demidenko, S., & Subramaniam, K.**; *Correlating Failure Modes With Component Reliability In Synthesis Of Self-Checking Control Unit Circuits*; Proc. ISIC 2001, pp. 190-192

[32] **Smith, D.**; *Reliability, maintainability and risk 5th ed.*; Boston : Butterworth Heinemann, 1997

[33] **O'Connor, P.**; *Practical reliability engineering*; Chichester ; New York : Wiley, c1985

[34] **Beasley, M.**; *Reliability for engineers*; Macmillan Education, 1991

[35] **Green, A., & Bourne, A.**; *Reliability technology*; London, New York, Wiley-Interscience, 1972

[36] **Demidenko, S.N., and Yarmolik, V.N.**; *Generation and Application of Pseudorandom Sequences for Random Testing*; John Wiley & Sons Ltd., 1988

[37] **Banks, J.**; *Handbook of simulation*; New York : Wiley, c1998

[38] **Bossel, H.**; *Modeling and simulation*; Wellesley, MA : A.K. Peters ; Wiesbaden, Germany : Vieweg, c1994

[39] **Ross, S.**; *Simulation*; San Diego : Academic Press, c1997

[40] **Biran, A., & Breiner, M.**; *MATLAB 5 for engineers*; Harlow, England ; Reading, Mass. : Addison-Wesley, 1999

[41] **Bushnell, M., & Agrawal, V.**; *Essentials of Electronic Testing*; Kluwer Academic Publishers, U.S.A., 2000

3 Robotic Visual Systems

3.1 Introduction

This chapter investigates robotic visual systems, which are among the most common forms of sensory mechanisms used in robots. Section 3.2 describes the basic concepts of visual systems. Section 3.3 then selects a specific application for robot vision, and describes the currently used visual system. A new, more efficient method is also considered, and its implementation described. Once implemented in code, the system was actually integrated into the whole robot system successfully, and demonstrated a marked improvement in performance.

3.2 Robotic Visual System Concepts

Real-time visual systems are necessary in applications where control of the robot is dependent on visual information. This, in fact, translates to a large proportion of implementations [15].

Older systems often used an initially captured image to study the environment. They formed Cartesian co-ordinates from this image to plot a course through the terrain. Real-time image processing is, of course, an advantage for such applications, and obviously a necessity for dynamic environments. Dynamic environments may consists of terrain with other moving objects, situations where an object is being tracked while it is moving, cases where the planned path of motion continuously changes, and so on.

There are two basic approaches to vision-based information acquisition and control [12,13]: Image-Based Visual Servo (IBVS), and Position-Based Visual Servo (PBVS). In IBVS, errors are mapped directly from the image, and sent to actuator commands that act to correct the apparent error in position. In PBVS, software is used to reconstruct the robot's environment, and paths for motion are mapped from this reconstruction.

3.2.1 Position-Based Visual Servo

In PBVS, features are extracted from the captured image, and used to create a partial 3-D reconstruction of the Euclidean environment, or of the motion of a tracked object in the environment. An error is then computed in the Euclidean space, and this error information is sent to the control system which acts to correct the error. Thus, the main challenge in this technique is tracking objects, and accurately estimating their Cartesian co-ordinates [16].

3.2.2 Image-Based Visual Servo

In IBVS, positions are calculated by simple comparisons. If the position of the tracked object in the current image matches the desired position of the object, then the object must be in the correct position. All co-ordinates and velocities are measured relative to the camera position. This approach is, obviously, suited to

situations where motion of the tracked object is largely confined to one plane, and where any skewing of the image is either minor or can be compensated for. Hybrid methods also exist where IBVS is used to control certain degrees of freedom, while the remaining degrees of freedom are controlled by other methods.

3.2.3 Cameras

The output of a camera is the 2-D image of an environment, taken from the perspective of the visual system. Several configurations are possible for camera positioning: 'eye in hand', where cameras are mounted on the robot, 'end point, open loop', where the camera is fixed and set up to observe the target, and 'end point, closed loop', where the camera is fixed and observes and target *and* robot.

The majority of cameras used by researchers conform to RS170 or PAL video standards, with sample rates of 30Hz or 25Hz, respectively. The advent of digital cameras, however, has introduced a variety of equipment that can operate at a range of resolutions and frame rates. Digital cameras usually have the digitizer and memory built into the camera, so that images can be stored on the camera, and accesses can be made to specific pixel regions.

3.2.4 Feature Extraction, Object Location, and Tracking

Feature extraction involves the system software finding measurable values for visual features in the captured image. Many types of features may be extracted, such as point features, the distance between two points in the image and the orientation of the line connecting the two points, perceived edge length, the relative areas of objects in the image, the centre of gravity or higher order moments of surfaces, and the parameters of shapes in the image.

Object location is often done in conjunction with feature extraction. Useful features often occupy a relatively small proportion of the image, and thus locating the useful objects is usually a major task in itself. Location of the objects, in turn, will be based on recognition of known features of the objects.

Once objects have been recognised, they may often have to be tracked [14]. Many applications rely on the ability of the visual system (camera and software) to keep track of moving objects. Depending on the relative speed of the moving object, of course, this can prove to be quite a memory-intensive operation. In such cases, the speed of the computer in processing the images is of paramount importance. This relates to three issues in particular: the amount of memory available, the speed of the processor, and the efficiency of the algorithm used for processing the images. Due to financial constraints, the first two parameters may often be fixed, or relatively inflexible. Thus, constant research into producing more efficient algorithms is the only way for improvement.

In order to do a detailed study of robotic visual systems, an application was selected which encompasses a large proportion of the elements of visual systems discussed above. The robot soccer competitions use robots that are autonomous units, working in conjunction with a central system that plans game strategies. The image processing requirements of the system are very intensive, and thus provide an ideal challenge for research and improvement. In the game, cameras are mounted above the playing field, and the captured images are passed onto the control unit.

3.3 Applications of Robotic Visual Systems

3.3.1 Robot Development

The Robocup robotics competitions [1,2] were started as a platform for testing practical implementations of new developments related to robotics – this includes

a variety of fields, such as electronics, signal processing, communications, image processing, and so on. The competition is designed as a game of soccer, played on a mini soccer field, using robots as the players. Several divisions are also available, for robots of different sizes, as well as a simulation league for proposed systems. The requirements of each division can also be quite different; for example, in the larger robots each mobile unit carries its own camera for navigation, while the smaller robots have a single camera that overlooks the entire playing field, and controls the robot movements from a central computer. The robocup tournaments are held annually, in different locations.

Robot design and development can be separated into several parts, such as the game-playing strategy, control units, robot navigation, and the vision system. As a joint project, Massey University and the Singapore Polytechnic have undertaken the task of improving their existing robots belonging to the small robots division. The vision system was targeted. In the division, all the robots are equipped with communications equipment. A central computer controls the robots by transmitting information to them regarding the location of the ball, where the robot must go next, where its opponents are, and where its team mates are. The robots also operate using a common vision system, which views the playing field from above, and transmits the real-time images of the game to the central computer. The computer then makes decisions on its game strategy based on the positions of all the players and the ball.

3.3.2 Conventional Vision System Implementation

The old vision system employed an incremental tracking system [11], whereby a window around each object is created, and once the location of an object has been determined, its movements are tracked only within the window. This technique is used mainly due to the savings it offers in processing time, as the whole image of the playing field does not need to be analysed for each iteration of the object recognition algorithm. Instead, only the window for each object needs to be

analysed. This results in a very small proportion of the field being analysed for each captured image, and has the downfall of sometimes losing an object that strays outside its window. This method was nevertheless necessary in order to make the image processing algorithm function at the speed required to process the captured images in real-time.

Once an image has been captured, the objects in it are classified in order to determine whether they are opponents, friendly players, or the ball. Classifying the objects into their respective categories is done by recognising coloured patterns on the upper faces of the robots. This is the second major process in the overall image processing algorithm, following the first phase of object location. When an image of the playing field is captured, each coloured pattern in the image is not composed of one solid colour, but is instead composed of a set of colours falling into a region in the colour space around the colour classification of the object (figure 3-1).

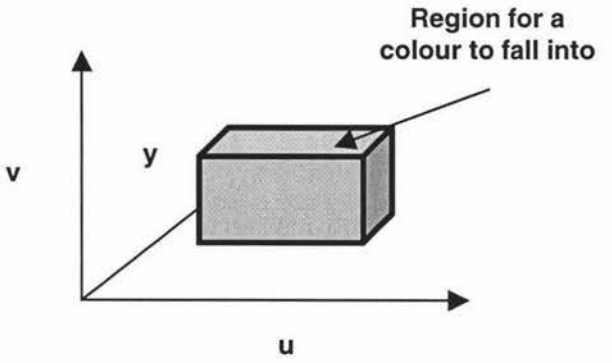


Figure 3-1: Y-U-V colour boundaries of a recognised game colour.

To represent the region in the colour space, a lookup table is used. When a colour pixel is encountered, the lookup table is searched to find the colour categorisation for that particular colour. This is a very time-consuming process, as a large lookup table is needed for the whole spectrum of colours that may be encountered in the image, while the search is executed for each pixel. A lookup for a pixel's colour would entail checking whether the pixel falls within all the boundaries

defined for a colour. For example, a lookup for the colour red would be something like:

```
if(Y<RedYMax && Y>RedYMin && U<RedUMax && U>RedUMin
&& V<RedVMax && V>RedVMin) ColourClassification = Red;
```

An additional issue is the memory utilisation of the process, for hardware implementations.

3.3.3 Improved Algorithm

The newly implemented algorithm incorporates all the functions required for the image analysis process. The main idea of the technique is presented in [3]. The image is scanned one row at a time, with each run of pixels of the same colour category being run-length encoded. For each pixel, an alternative method is used to check which colour category a pixel falls into. Instead of using one large lookup table for the colour classifications, three relatively small tables are used for the separate Y, U, and V values, providing a large memory saving. The colours used in the game are assigned numbers corresponding to powers of two, for example – black – 0, red – 1, blue – 2, green – 4, orange – 8, and so on. When a pixel is encountered, it is possible that its YUV value falls into a region of the colour space that corresponds to more than one of the colour categories of the game. Using logical OR and AND operations the colour category of the newly acquired pixel can be determined. For example, if the colour orange is defined as:

Y={0,8,8,8,8,8,8,8,8,8}, U={0,0,0,0,0,0,0,8,8,8}, V={0,0,0,0,0,0,0,8,8,8}

and the colour blue as:

Y={0,2,2,2,2,2,2,2,2,2}, U={2,2,2,0,0,0,0,0,0,0}, V={0,0,0,2,2,2,0,0,0,0}

The different lookup Y, U, and V tables for each colour can be combined using the bitwise 'OR' operator to form a single lookup table for each Y, U, and V parameter. So, the combined table for Y is:

Y={0,8,8,8,8,8,8,8,8,8} OR Y={0,2,2,2,2,2,2,2,2,2}
=Y{0,10,10,10,10,10,10,10,10,10}

Now, a newly acquired pixel with YUV values of (1,8,9) can be evaluated by the expression Y[1] AND U[8] AND V[9]. This results in an '8' when performed with the orange colour definition, but '0' for the blue, indicating that the pixel falls into the 'orange' category [4]. During the run-length encoding, if a pixel falls into the same colour category as the preceding pixels, it is added to the same run-length encoded structure.

After the first pass through the image, each row of pixels of the same colour category will be stored in a run-length encoded structure. If two lines of pixels of the same colour are connected on adjacent rows of the image, they are considered to be parts of the same object. Forming these objects is the final stage of the algorithm. The image is scanned again, line by line. This time, each run-length encoded sequence of pixels is associated with surrounding pixels of the same colour, thus forming the actual objects in the image (figure 3-2).

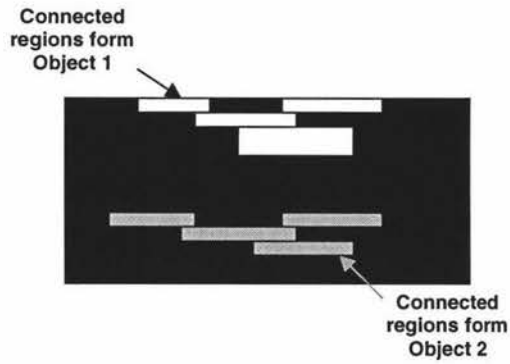


Figure 3-2: Forming objects from connected regions.

The flow chart in figure 3-3 illustrates the functioning of the algorithm.

3.3.4 Implementation

3.3.4.1 Data Types

The existing software for image processing in the robot, as well as all the other components of the robot, were written using C++ [5-10], and therefore the new algorithm was implemented in the same language. The program code begins by defining the numbers assigned to each of the significant colours in the game:

```
#define COLOUR1_REF 1
#define COLOUR2_REF 2
#define BALL_REF 4
#define TEAM_REF 8
#define OPPONENT_REF 16
#define BACKGROUND_REF 0
```

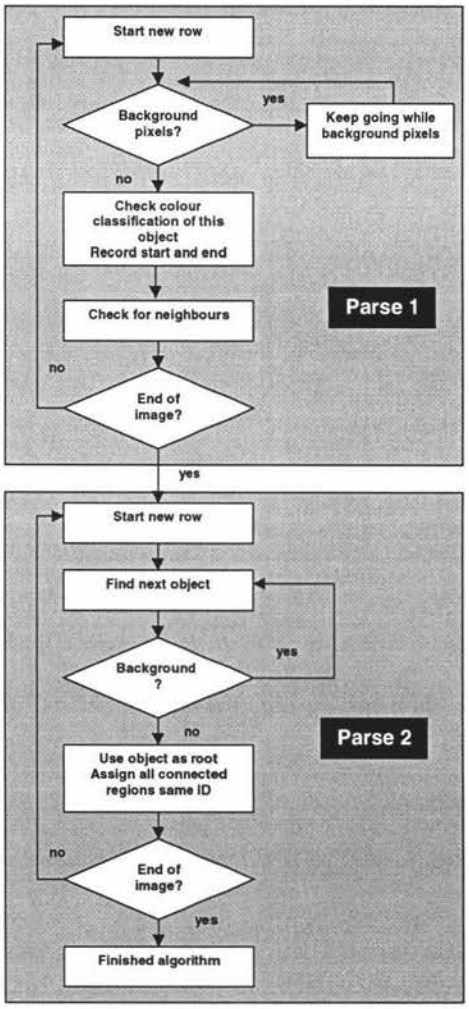


Figure 3-3: Algorithm description flow chart.

Next, the data structures used are defined. Two structures are used to hold all the data generated during the image analysis process. The RLE_element structure holds the run-length encoding information for each sequence of pixels falling into the same colour category. iElement_ID is the number assigned to the sequence of pixels in the first run through the image. Each sequence of pixels gets a new iElement_ID. iObject_ID is the number assigned to each object, after the second run through the image. At this stage, all connected regions of the same colours would have been identified as being the same object. iStart_Pos, iEnd_Pos, and iRow store information about the location of the run of pixels. iColour_Ref records which colour category the run of pixels falls into. iParent_Child_Exists is significant for identifying whole objects, in their construction from individual lines of pixels.

```
struct RLE_element{
     int iElement_ID;//element id
     int iObject_ID;//classified object
     int iStart_Pos, iEnd_Pos;
     int iColourRef;//colour reference of pixel
     int iParent_Child_Exists;//-1 if no parent, child
     int iRow;
};
```

For every sequence of pixels other than the playing field, regions of the same colour in the line above or below the current sequence of pixels are recorded as neighbours, in a Neighbour structure. This indicates that a line of pixels is connected to similar pixels around it (figure 3-4).

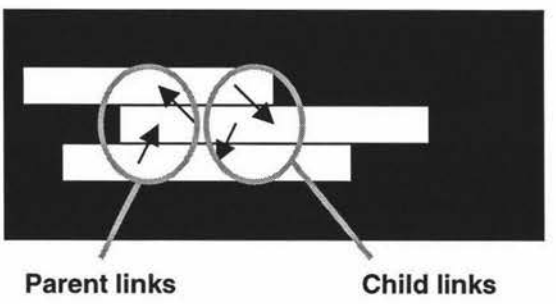


Figure 3-4: 'Neighbour' links between adjacent rows.

To incorporate this, within the Neighbour structure, iParents[ ] and iChildren[ ] store the iElement numbers of the current RLE_Element's neighbours. Both the structures are stored as arrays. RLE_Elements are simply indexed consecutively, while Neighbour structures are indexed by the iElement_ID of the RLE_Element they belong to.

```
struct Neighbour{
      int iParent_Count, iChild_Count; //-1 if none exists
      int iParents[20];//array holds RLE_IDs of parents
      int iChildren[20];//holds RLE_IDs of chilren
      int iCurrent_Child, iCurrent_Parent; //used in object
      //location
      //to identify how many parents and children
      //have been visited in traversal
};
```

Finally, the YUV boundaries for all the game colours are defined.

```
BYTE ballcolYmin=10, ballcolYmax=50;
BYTE ballcolUmin=60, ballcolUmax=80;
BYTE ballcolVmin=60, ballcolVmax=80;

BYTE colour1Ymin=80, colour1Ymax=100;
BYTE colour1Umin=90, colour1Umax=110;
BYTE colour1Vmin=90, colour1Vmax=100;

BYTE colour2Ymin=150, colour2Ymax=200;
BYTE colour2Umin=90, colour2Umax=140;
BYTE colour2Vmin=110, colour2Vmax=120;

BYTE teamcolYmin=120, teamcolYmax=150;
BYTE teamcolUmin=110, teamcolUmax=150;
BYTE teamcolVmin=125, teamcolVmax=150;

BYTE oppocolYmin=100, oppocolYmax=200;
BYTE oppocolUmin=155, oppocolUmax=180;
BYTE oppocolVmin=80, oppocolVmax=110;
```

These define the boundaries for the Y, U, and V values of the recognised colours of the game. For practical use, the values will have to be calibrated so that any pixel that belongs to an object that is part of the game is identified under the correct category.

3.3.4.2 Run-Length Encoding

There are essentially three main sets of functions in the program. The first is the process of run-length encoding. Secondly, after run-length encoding a sequence of pixels, its parents and children must be found. The last task is the object location. The code for the run-length encoding is given below. The scanning continues as a loop, finishing at the last line of the image. Within each line, a loop continues for every run of pixels that is tested to be the same colour. When any colour is encountered, the start location of that colour is recorded in the next available structure of the array, and it is assigned the next available iElement number. The next pixel is tested for its colour, and if it is the same, it is added to the same structure, and the end location of the run-length encoding is incremented. This is continued until a different colour is encountered.

```
void RLE_image(){
      int row_counter, col_counter;
      int pixel_in_image;
      iCurrent_ID = 1;
      BYTE * pImage=iImage_Matrix;
      BYTE pY,pU,pV;

      //main loop ends at last row of image matrix
      for(row_counter=0;row_counter<IMAGE_HEIGHT;row_counter++){
      RowStart[row_counter]=RLE_index + 1; //since it will be
      //incremented later

      for(col_counter=0;col_counter<IMAGE_WIDTH;col_counter++){

      //RLE until not background colour
      /*
      while(iImage_Matrix[(row_counter*IMAGE_WIDTH)+col_counter]=
=BACKGROUND_REF && col_counter<IMAGE_WIDTH){
      //do nothing except increment counter
      col_counter++;
      }
      */

      //ARRAY APPOACH
      ///*
      //These references to iImage_Matrix will be more efficient
      //with a pointer or a counter
      // so that the ((row_counter*IMAGE_WIDTH)+col_counter)*3 is
      //not needed for each pixel element

      //check what class current pixel falls into
      //BITWISE and of LUT values
```

```
pixel_in_image =
      LUT_Y[iImage_Matrix[((row_counter*IMAGE_WIDTH)+col_counter)
*3]]&
      LUT_U[iImage_Matrix[((row_counter*IMAGE_WIDTH)+col_counter)
*3+sizeof(BYTE)]] &
      LUT_V[iImage_Matrix[((row_counter*IMAGE_WIDTH)+col_counter)
*3+2*sizeof(BYTE)]];

      RLE_index++;


      RLE[RLE_index].iStart_Pos=col_counter;//(row_counter*IMAGE_
WIDTH)+

      RLE[RLE_index].iEnd_Pos=col_counter;//(row_counter*IMAGE_WI
DTH)+
      //REMOVED SINCE ROW AND COLUMN WILL CORRECT POSITION
                  RLE[RLE_index].iColourRef=pixel_in_image;

                  if(pixel_in_image!=BACKGROUND_REF){
                        RLE[RLE_index].iElement_ID=iCurrent_ID;
                  } else {
                        RLE[RLE_index].iElement_ID=0;
                  }
                  RLE[RLE_index].iRow=row_counter;
                  col_counter++;

                  //continue while same pixel class
pixel_in_image =
      LUT_Y[iImage_Matrix[((row_counter*IMAGE_WIDTH)+col_counter)
*3]]&
      LUT_U[iImage_Matrix[((row_counter*IMAGE_WIDTH)+col_counter)
*3+sizeof(BYTE)]] &
      LUT_V[iImage_Matrix[((row_counter*IMAGE_WIDTH)+col_counter)
*3+2*sizeof(BYTE)]];

      while((pixel_in_image==RLE[RLE_index].iColourRef)
                  //compare to check if same pixel class
      && (col_counter<IMAGE_WIDTH)){

      RLE[RLE_index].iEnd_Pos=col_counter;
      //(row_counter*IMAGE_WIDTH)
      col_counter++;
pixel_in_image =
LUT_Y[iImage_Matrix[((row_counter*IMAGE_WIDTH)+col_counter)*3]]&
LUT_U[iImage_Matrix[((row_counter*IMAGE_WIDTH)+col_counter)*3+siz
eof(BYTE)]] &
LUT_V[iImage_Matrix[((row_counter*IMAGE_WIDTH)+col_counter)*3+2*s
izeof(BYTE)]];
                  }
col_counter--;//since it will be incremented by the for loop

      pImage-=3;  //move the pointer back as well
                  //iCurrent_ID++;

      if(RLE[RLE_index].iColourRef != BACKGROUND_REF) {
                        iCurrent_ID++;
```

```
                        parentChildLink();
                    }

        /*----------end of Child-Parent Portion----------*/


            }//end col_counter while loop

        }//end row_counter while loop
}
```

3.3.4.3 Neighbour Testing

Once each set of pixels has been run-length encoded, the parentChildLink( ) function is called to update the RLE_Element structures with respect to new connections that may have been formed of similarly coloured pixels. This is performed for all game colours except the playing field itself, as it is not required for the object identification. The function simply uses the current RLE_Element, and checks to see if there were any RLE_Elements with the same colour directly above. If so, the current RLE_Element's iElement number is recorded as a child in the Children[ ] array of the RLE_Element above, and the RLE_Element above is recorded as a parent of the current RLE_Element. The code for this process is shown below:

```
void parentChildLink(){

        /*----------end of RLE portion----------*/
        //check for parent-child relationship by:
        //checking pixel directly above
        //pixel to left should already have been RLEed
        //SKIP the lines where iColourRef == BACKGROUND_REF (i.e.
//0)

        if(RLE[RLE_index].iRow!=0){//only link for rows that are
//not the first row
        for(int i=RowStart[RLE[RLE_index].iRow1];i<RLE_index;i++){
//CHECK ALL RLE //ITEMS, later add a data structure that keeps
track of the start //of each row, so that we do not have to go
through irrelevant //items

//if(RLE[i].iRow==RLE[RLE_index].iRow-1){//Look at previous row
//only

        if(RLE[RLE_index].iColourRef == RLE[i].iColourRef){//Check
//for same colour
```

```
      if(RLE[i].iEnd_Pos>=RLE[RLE_index].iStart_Pos&&RLE[i].iStar
t_Pos<=RLE[RLE_index].iEnd_Pos){

      RLE[RLE_index].iParent_Child_Exists=1;

      RLE[i].iParent_Child_Exists=1;

      //update this object's parent array
      int NeighbourID=RLE[RLE_index].iElement_ID;
      int N_P_ID=RLE[i].iElement_ID;

      Neighbours[NeighbourID].iParents[Neighbours[NeighbourID].iP
arent_Count++] = i;

      //update parent's child array

      Neighbours[N_P_ID].iChildren[Neighbours[N_P_ID].iChild_Coun
t++] = RLE_index;

              }
            }
        //}
        }//end parent-child for loop
      }
```

3.3.4.4 Object Location

Once the whole image has been run-length encoded, and the parent-child links have been established, the image is traversed again to locate the player-identifying patterns. This is done in the object_location( ) and traverse_graph( ) functions. The object_location( ) function loops through all the RLE_Elements which correspond to non-background pixels. For each RLE_Element, the traverse_graph( ) function is performed. This function uses the structure passed to it as the root of a tree. It then assigns a new iObject_ID to the structure. Then, using recursive calls it performs a depth-first search through all its parents and children. For each new parent or child, it assigns the same iObject_ID, thus identifying all connected RLE_Elements as the same object. Once the entire graph has been traversed, the function returns to the object_location( ) function, and the next RLE_Element is checked. If it already possesses an iObject_ID, it is concluded that this element has been identified as part of an object. The code for the object location functions is given below:

```
void object_location(){
      iCurrent_ID=1;

      //loop to start graph traversal using each RLEed element as
      //root
      for(int j=0;j<=RLE_index;j++){
            if(RLE[j].iColourRef!=BACKGROUND_REF
                  &&RLE[j].iObject_ID==0) {
                  traverse_graph(j);iCurrent_ID++;
      }

  }
}

//perform depth-first traversal over all vertices
void traverse_graph(int j){
      int iCurrent_Vertex=j, i;

      //stop and RETURN at this point if current vertex has (no
//parents or children) OR

      //(all parents and children have already been visited)
      if(RLE[j].iParent_Child_Exists==0
      //ParentChild doesn't exist has value 0

      ||RLE[j].iObject_ID!=0)
      //Check to see if node already visited
      {
            RLE[j].iObject_ID=iCurrent_ID;
            return;
      }
      else{
      //go to next parent if parent list not exhausted
      //go to next child if child list not exhausted
      //call traverse_graph() and send as parameter the vertex
//number of the
      //next vertex
      RLE[j].iObject_ID=iCurrent_ID;
      for(i=0; i<Neighbours[RLE[j].iElement_ID].iParent_Count;
i++)
            {
      traverse_graph(Neighbours[RLE[j].iElement_ID].iParents[i]);
            }

      for(i=0;i<Neighbours[RLE[j].iElement_ID].iChild_Count; i++)
            {

      traverse_graph(Neighbours[RLE[j].iElement_ID].iChildren[i])
;
            }


      }
}
```

3.4 Chapter Summary

The robots used in the Robocup competitions were chosen as an implementation platform for the improved image processing algorithm. This is because these robots incorporate a wide range of technologies, including wireless communications, image processing, robotics, and mechanics.

The image processing algorithm was targeted for improvement, in the light of a newly published technique for object identification by colour. Identification of objects by their colour is the main function of the image processing algorithm, as all the objects in the game are identifiable only by their colour. Categorising a pixel's colour was a time consuming process in the original algorithm, as the only way to achieve this was to compare the captured pixel's colour against all the colours in a table. The new algorithm provided a faster method of achieving this, by a series of boolean comparisons.

The implementation of the technique incorporates identifying pixel colours, and recognition of the objects that they belong to. The method has been tested using artificial images, as well as using captured game images. A significant time saving is gained over the original image analysis algorithm, and the new system is currently undergoing final modifications to be integrated into the entire robot control system.

It is, theoretically, possible for a further improvement in the performance of the algorithm, if an alternative implementation can be found for the traverse_graph( ) function. Since the function is currently implemented recursively, a method using loops may be possible, whereby a time saving is gained. All the relevant code for the vision system implementation is given in Appendix 3.

Fault tolerance and self-checking elements were incorporated into the program. The algorithm and program were implemented, and are in use in the Robocup team of the Singapore Polytechnic, as a part of a joint project under the ASIA 2000 initiative.

3.5 References

[1] http://www.teambots.org

[2] http://ci.etl.go.jp

[3] **C.H. Messom, S. Demidenko, K. Subramaniam, & G. Sen Gupta**; *Size/Position Identification in Real-Time Image Processing using Run Length Encoding*; IEEE Instrumentation and Measurement, Technology Conference, Anchorage, AK, USA, 2002

[4] **Bruce, J., Balch, T., Veloso M.**; *Fast and Cheap Colour Segmentation for Interactive Robots*; IROS 2000, San Francisco, 2000

[5] **Ammeraal, L.**; *Algorithms and Data Structures in C++*; John Wiley and Sons, 1997

[6] **Weiss, M.A.**; *Data Structures and Problem Solving Using C++*; Addison Wesley, 2000

[7] **Hyman, M., & Arnson, B.**; *Visual C++ 6 for dummies*; Foster City, CA : IDG Books Worldwide, c1998

[8] **Andrews, M.**; *Learn Visual C++ now*; Redmond, Wash. : Microsoft Press, c1996

[9] **Flamig, B.**; *Practical algorithms in C++*; New York : Wiley, c1995

[10] **Kruglinski, D.**; *Inside Visual C++*; Redmond, WA. : Microsoft Press, 1997

[11] **Messom, C.H., Sen Gupta, G., Lian, S.H.**; *Distributed Real-time Image Processing for a Dual Camera System*; CIRAS 2001, Singapore, 2001

[12] **Corke, P., & Hutchinson, S.**; *Real-Time Vision, Tracking and Control*; Proc. 2000 IEEE ICRA, San Francisco, U.S.A., 2000

[13] **Kragic, D., & Christensen, H.**; *Tracking Techniques for Visual Servoing Tasks*; Proc. 2000 IEEE ICRA, San Francisco, U.S.A., 2000

[14] **Ehrenmann, M., Ambela, D., Steinhaus, P., & Dillmann, R.**; *A Comparison of Four Fast Vision Based Object Recognition Methods for Programming by Demonstration Applications*; Proc. 2000 IEEE ICRA, San Francisco, U.S.A., 2000

[15] **Eberst, C., & Herbig, T.**; *On The Application of the Concept of Dependability for Analysis and Design of Vision Systems*; IAS-6, IOS Press, Amsterdam, 2000

[16] **Amat, J., Fernandez, J., & Casals, A.**; *Vision-Based Navigation System for Autonomous Vehicles*; IAS-6, IOS Press, Amsterdam, 2000

4 Conclusions

The research presented in this thesis was intended to improve fault tolerance and efficiency in applications of distributed robotics. Robot control and robot vision were selected as fields that provided great scope for research. Achievements of the work are:

- New methods have been researched for representing errors in control units, by examination of the flow graphs representing the program structure
- New techniques have been developed for determining the reliabilities of control units, using the existing concepts of reliability engineering
- Methods have been developed for designing control units that can be fault-tolerant, with the reliability information
- A range of error coding schemes were tested to determine which provided adequate fault coverages – the schemes were tested for a number of common error types
- In the field of vision systems, the Robocup robots were chosen as a suitable platform which possessed an image processing algorithm that could be improved
- Vision system software was developed and customised for the robots, and integrated successfully into the existing system

- Programs were developed specifically for simulation of the error coding and fault-tolerance experiments, and all results presented were acquired from the use of this software
- The research and results from this thesis have been discussed and published at a seminar at Massey University, and two international conferences – ISIC 2001 in Singapore, and IMTC 2002 in Anchorage, U.S.A. The developed software for image processing was done with the support of the ASIA 2002 programme, which supports a collaboration between Massey University and the Singapore Polytechnic. The visual system software was presented to Singapore Polytechnic during a research visit in September 2001. The fully integrated robot system is in use for the world Robocup robotics competitions.
- Future development in the area of fault tolerance can be directed at efficient key assignment for flow graph segments. There is scope here for the application of graph colouring techniques in assigning the keys. The visual system can be improved further by the implementation of a functioning self-checking mechanism in the program. This can be achieved by analysis of the program structure, and incorporation of the most efficient self-checking scheme for the algorithm.

5 Bibliography

This section contains a complete listing of all the references used in the thesis

Agrawal, P.; *Fault Tolerance in Multiprocessor Systems Without Dedicated Redundancy*; IEEE Transactions on Computers, Vol. 37, No. 3, Mar. 1988

Amat, J., Fernandez, J., & Casals, A.; *Vision-Based Navigation System for Autonomous Vehicles*; IAS-6, IOS Press, Amsterdam, 2000

Ammeraal, L.; *Algorithms and Data Structures in C++*; John Wiley and Sons, 1997

Andrews, M.; *Learn Visual C++ now*; Redmond, Wash. : Microsoft Press, c1996

Asama, H., Fukuda, T., Arai, & T., Endo, I.; *Distributed Autonomous Robotic Systems*; Springer-Verlag, 1994

Banks, J.; *Handbook of simulation*; New York : Wiley, c1998

Beasley, M.; *Reliability for engineers*; Macmillan Education, 1991

Bekey, G., Parker, L.E., & Barhen, J.; *Distributed Autonomous Robotic Systems 4*; Springer-Verlag, 2000

Bezerra, E., Vargas, F., & Gough, M.; *Improving Reconfigurable Systems Reliability by Combining Periodical Test and Redundancy Techniques: A Case Study*; Journal of Electronic Testing: Theory and Applications (JETTA), Vol. 17, No. 2, Apr. 2001

Biran, A., & Breiner, M.; *MATLAB 5 for engineers*; Harlow, England ; Reading, Mass. : Addison-Wesley, 1999

Bossel, H.; *Modeling and simulation*; Wellesley, MA : A.K. Peters ; Wiesbaden, Germany : Vieweg, c1994

Bruce, J., Balch, T., Veloso M.; *Fast and Cheap Colour Segmentation for Interactive Robots*; IROS 2000, San Francisco, 2000

Bushnell, M., & Agrawal, V.; *Essentials of Electronic Testing*; Kluwer Academic Publishers, U.S.A., 2000

Cass, S.; *Robosoccer*; IEEE Spectrum, May 2001

Cooray, N., & Czeck, E.; *Guaranteed Fault Detection Sequences for Single Transition Faults in Finite State Machine Models Using Concurrent Fault Simulation*; Journal of Electronic Testing: Theory and Applications (JETTA), Vol. 8, No. 3, June 1996

Corke, P., & Hutchinson, S.; *Real-Time Vision, Tracking and Control*; Proc. 2000 IEEE ICRA, San Francisco, U.S.A., 2000

Damiani, M., Olivo, P., Favalli, M., & Riccò, B.; *An Analytical Model for the Aliasing Probability in Signature Analysis Testing*; IEEE Transactions on Computer-Aided Design, Vol. 8, No. 11, Nov. 1989

Das, D., and Touba, N.A.; *Synthesis of Circuits with Low-Cost Concurrent Error Detection Based on Bose-Lin Codes*; Journal of Electronic Testing: Theory and Applications (JETTA), Vol. 15, No. 1 / 2, Aug/Oct 1999

Demidenko, S.N., Levin, E.M., and Lever, K.V.; *Concurrent self-checking for microprogrammed control units: an analytical survey*; IEE Proceedings-E, Nov. 1991, Vol.138, No.6

Demidenko, S.N., Levin, E.M., and Lever, K.V.; *New approach to synthesis of self-checking microprogrammed control units with specified fault-detection probabilities*; IEE Proceedings-E, Nov. 1991, Vol. 138, No. 6

Demidenko, S.N., Levine, E.M., and Piuri, V.; *Synthesis of On-Line Testing Control Units: Flow Graph Coding/Monitoring Approach*; IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, IEEE Computer Society, Proceedings 2000

Demidenko, S., & Subramaniam, K.; *Correlating Failure Modes With Component Reliability In Synthesis Of Self-Checking Control Unit Circuits*; Proc. ISIC 2001, pp. 190-192

Demidenko, S.N., and Yarmolik, V.N.; *Generation and Application of Pseudorandom Sequences for Random Testing*; John Wiley & Sons Ltd., 1988

Dorigo, M., & Colombetti, M.; *Robot shaping : an experiment in behavior engineering*; Cambridge, Mass. : MIT Press, c1998.

Eberst, C., & Herbig, T.; *On The Application of the Concept of Dependability for Analysis and Design of Vision Systems*; IAS-6, IOS Press, Amsterdam, 2000

Edirisooriya, G., & Robinson, J.; *Aliasing Properties of Circular MISRs*; Journal of Electronic Testing: Theory and Applications (JETTA), Vol. 4, No. 2, May 1993

Ehrenmann, M., Ambela, D., Steinhaus, P., & Dillmann, R.; *A Comparison of Four Fast Vision Based Object Recognition Methods for Programming by Demonstration Applications*; Proc. 2000 IEEE ICRA, San Francisco, U.S.A., 2000

Flamig, B.; *Practical algorithms in C++*; New York : Wiley, c1995

Glover, I., & Grant, P.; *Digital Communications*; Prentice Hall Europe 1998

Green, A., & Bourne, A.; *Reliability technology*; London, New York, Wiley-Interscience, 1972

Gupta, S., Pradhan, D.; *Utilization of On-Line (Concurrent) Checkers during Built-In Self-Test and Vice Versa*; IEEE Transactions on Computers, Vol. 45, No. 1, Jan. 1996

Hellebrand, S., Wunderlich, H., & Hertwig, A.; *Mixed-Mode BIST Using Embedded Processors*; Journal of Electronic Testing: Theory and Applications (JETTA), Vol. 12, No. 1 / 2, Feb./Apr. 1998

http://www.ci.etl.go.jp

http://www.frc.ri.cmu.edu/robotics-faq.html

http://www.sp-edu.sg/schools/eee.news.html

http://www.teambots.org

Hyman, M., & Arnson, B.; *Visual C++ 6 for dummies*; Foster City, CA : IDG Books Worldwide, c1998

Kragic, D., & Christensen, H.; *Tracking Techniques for Visual Servoing Tasks*; Proc. 2000 IEEE ICRA, San Francisco, U.S.A., 2000

Kruglinski, D.; *Inside Visual C++*; Redmond, WA. : Microsoft Press, 1997

Ling, Y., Mi, J., & Lin, X.; *A Variational Calculus Approach to Optimal Checkpoint Placement*; IEEE Transactions on Computers, July 2001, Vol. 50, No. 7

Mahmood, A., and McCluskey, E.J.; *Concurrent Error Detection Using Watchdog Processors - A Survey*; IEEE Transactions on Computers, Feb.1988, Vol.37, No.2

Messom, C.H., Demidenko, S., Subramaniam, K., & Sen Gupta, G.; *Size/Position Identification in Real-Time Image Processing using Run Length Encoding*; IEEE Instrumentation and Measurement, Technology Conference, Anchorage, AK, USA, 2002

Messom, C.H., Sen Gupta, G., Lian, S.H.; *Distributed Real-time Image Processing for a Dual Camera System*; CIRAS 2001, Singapore, 2001

Mukherjee, N., Karri, R., & Chakraborty, T.; *Built-In Self Test: A Complete Test Solution for Telecommunication Systems*; IEEE Communications Magazine, June 1999

Nicolaidis, M.; *On-line testing for VLSI: state of the art and trends*; Integration: the VLSI journal 26, 1998

Nicolaidis, M.; *Efficient UBIST Implementation for Microprocessor Sequencing Parts*; Journal of Electronic Testing: Theory and Applications (JETTA), Vol. 6, No. 3, June 1995

Nicolaidis, M, & Anghel, L.; *Concurrent Checking for VLSI*; Microelectronics Engineering, No. 49, pp. 139-156, 1999

Nicolaidis, M., and Zorian, Y.; *On-Line Testing for VLSI-A Compendium of Approaches*; Journal of Electronic Testing: Theory and Applications (JETTA), Vol. 12, No. 1 / 2, Feb/April 1998

Nicolici, N., Al-Hashimi, B., Brown, A., & Williams, A.; *BIST Hardware Synthesis for RTL Data Paths Based on Test Compatibility Classes*; IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems; Vol. 19, No. 11, Nov. 2000

O'Connor, P.; *Practical reliability engineering; Chichester*; New York : Wiley, c1985

Parekhji, R.A., Venkatesh, G., and Sherlekar, S.D.; *Concurrent Error Detection Using Monitoring Machines*; IEEE Design and Test of Computers, 1995

Parekhji, R.A., Venkatesh, G., and Sherlekar, S.D.; *Monitoring Machine Based Synthesis Technique for Concurrent Error Detection in Finite State Machines*; Journal of Electronic Testing: Theory and Applications (JETTA), Vol. 8, No. 2, April 1996

Parker, L.; *Current State of the Art in Distributed Autonomous Mobile Robotics*; Distributed Autonomous Robotic Systems 4, Springer-Verlag, 2000

Patterson & Hennessy; *Computer Organization and Design 2nd Ed.*; Morgan Kaufman Publishers 1998

Piuri, V., Antola, A., Ferrandi, F., & Sami, M.; *Semiconcurrent Error Detection in Data Paths*; Vol. 50, No. 5, May 2001, IEEE Transactions on Computing

Proceedings: Intelligent Autonomous Systems (Conference) (6th : Giudecca Island, Italy); *Intelligent autonomous systems 6*; Amsterdam ; Oxford : IOS Press, c2000

Ross, S.; *Simulation*; San Diego : Academic Press, c1997

Shen, L., & Su, S.; *A Functional Testing Method for Microprocessors*; IEEE Transactions on Computers, Vol. 37, No. 10, October 1988

Smith, D.; *Reliability, maintainability and risk 5th ed.*; Boston : Butterworth Heinemann, 1997

Spragins, J.; *Telecommunications: Protocols and Design*; Addison Wesley, 1991

Stiffler, J.J.; *On-Line Fault Monitoring*; Journal of Electronic Testing: Theory and Applications (JETTA), Vol. 12, No. 1 / 2, Feb/April 1998

Talkhan, E., Ahmed, A., & Salama, A.; *Microprocessor Functional Testing Techniques*; IEEE Transactions on Computer-Aided Design, Vol. 8, No.3, Mar. 1989

Tsuda, N.; *Fault Tolerant Processor Arrays Using Additional Bypass Linking Allocated by Graph-Node Coloring*; Vol. 49, No.5, May 2000, IEEE Transactions on Computers

Upadhyaya, S.J., and Ramamurthy, B.; *Concurrent Process Monitoring with No Reference Signatures*; IEEE Transactions on Computers, April 1994, Vol.43, No.4

Weiss, M.A.; *Data Structures and Problem Solving Using C++*; Addison Wesley, 2000

Yim, M, Zhang, Y, & Duff, D.; *Modular Robots*; IEEE Spectrum, Feb. 2002

Appendices

Appendix 1

MATLAB Programs

1.1 Introduction

The code for the error-coding simulations is given in this section. The simulation parameters are also provided, along with a description of the simulation process, hardware used, and memory requirements.

All possible combinations were generated for the bit sequences; thus the 8-bit sequences had a total of 2^8=256 combinations, and the 12-bit sequences had a total of 2^{12}=4096 combinations. Generating all possible combinations allowed for a conclusive result of the fault coverage acquired. Once the signatures had been calculated for each pair of sequences (the actual real key, and the corrupted key),

the results are compared, and the difference between them calculated. If this difference is 0, then the error is considered detected.

1.2 System Hardware

The system hardware used is as follows:

CPU: Pentium III
Processor: 1.002GHz Family 6 Model 8 Stepping 10 Intel
OS: Microsoft Windows 2000 Professional
Version: 5.0.2195 Service Pack 2 Build 2195
System Type: X86-based PC
Physical Memory: 327,560kB
Virtual Memory: 1,768,756kB

1.3 Key Simulation

8-bit keys are used for the examples here, and were simulated in the same manner as the 12-bit keys.

1.3.1 Full-Combination Set Deletion

The 8-bit keys contained 256 combinations for each run of bit-deletions. Three runs were employed in total, reducing the number of bits by 1, 2, and 3. Therefore, for single-bit deletions there were a total of ${}^{8}C_{1}=8$ deletion combinations for each of the 256 sequences. Similarly, for 2-bit deletions, there were ${}^{8}C_{2}=28$ combinations for each sequence, making a total of 28x256=7168 sets of calculations and results.

1.3.2 Consecutive Set Deletions

In addition to deleting bits in all the possible combinations, the special case of bits in groups was also considered. Once again, 256 bit sequences are possible for 8-bit keys. The number of combinations x, however, is modelled by:

$$x=k-n+1,$$

where n is the length of the sequence (8 in this case), and k is the number of bits to be shortened by. Thus, for example, to shorten the 8-bit sequences by 2 bits, there will be 8-2+1=7 combinations for introducing errors, making a total of 256x7=1792 results.

1.3.3 Complete-Set Insertions

This variation simply inserts bits into the keys, in the all the possible combinations. Thus, as the number of bits to be lengthened by increases, the number of possible combinations changes accordingly in an nChoosek fashion.

1.3.4 Block Errors

This variation inserts bits into the keys in groups. Thus, if 3-bit errors are used with 6-bit keys, the error sequences resulting would be as follows:

Original Key:	101001
Error Sequence 1:	■■■101001
Error Sequence 2:	1■■■01001

1.3.5 Consecutive Positions

The third variation inserts bits into consecutive positions in the key. The bit insertion patterns in this experiment are essentially a subset of the full-combination insertions. For example, using 3-bit errors with 6-bit keys:

Original Key:	101001
Error Sequence 1:	■1■0■1001
Error Sequence 2:	1■0■1■001

1.4 MATLAB Programs

1.4.1 graph_main.m

```
%graph_main.m

%Karthik Subramaniam
%Institute of Information Sciences and Technology
%Turitea Campus, Massey University

%Constructs graph based on matrices given by the user
%The program then simulates a random traversal of the graph
%by using transition error probabilities definde by the user

%Graph construction for example 1 in book
%graph with 5 vertices, one conditional branch
%all branches with prob 0.5
%3-bit keys

function graph_main
   graph_size=5;
   max_no_of_transitions=10;
   no_of_runs=1;
   sequence_length=31;
   %key_sequence contains a no. of runs per no. of transitions
   %up to a max no. of transitions, plus the key generated for
   %each run.
   %Thus, with a max of 10 transitions, there will be length(1)x5 +
   %length(2)*5 ... = 50 keys generated in total.
   %The keys for each transition length will be different.
   key_sequence=zeros(max_no_of_transitions,no_of_runs,sequence_length);

   [graph, erroneous, keys, vertex_errors]=construct_graph(graph_size);

   for i=1:max_no_of_transitions
      for j=1:no_of_runs
         fprintf('new run:')%test
         i%test
         key_sequence(i,j,:)=traverse_graph(graph,                erroneous,                keys,
graph_size,vertex_errors,i,sequence_length);
      end
```

```
end

function [graph, erroneous, keys, vertex_errors] = construct_graph(graph_size)
  graph=zeros(graph_size,graph_size);  %graph connectivity
  erroneous=zeros(graph_size,graph_size);  %erroneous transitions
  keys=zeros(graph_size,3);  %vertex check keys
  vertex_errors=zeros(graph_size,1); %reliability of each vertex

  %set up graph connectivity
  graph(1,2)=1; graph(1,3)=1;
  graph(2,5)=1;
  graph(3,4)=1;
  graph(4,5)=1;

  %  transition(col -> row)
  %  cell(3,2) is transition (Q3 -> Q2)
  %   _                 _
  %  |  0  1  1  0  0  |
  %  |  0  0  0  0  1  |
  %  |  0  0  0  1  0  |
  %  |  0  0  0  0  1  |
  %  |_ 0  0  0  0  0 _|


  %set up erroneous transition matrix
  %transition (col -> row)
  %cell(2,3) is prob. of erroneous transition (Q2 -> Q3),
  %given that the current vertex is in error
  erroneous(1,1)=0.3; erroneous(1,2)=0; erroneous(1,3)=0; erroneous(1,4)=0.3; erroneous(1,5)=0.4;
  erroneous(2,1)=0.25; erroneous(2,2)=0.25; erroneous(2,3)=0.25; erroneous(2,4)=0.25; erroneous(2,5)=0;
  erroneous(3,1)=0.1; erroneous(3,2)=0.4; erroneous(3,3)=0.3; erroneous(3,4)=0; erroneous(3,5)=0.2;
  erroneous(4,1)=0.25; erroneous(4,2)=0.25; erroneous(4,3)=0.25; erroneous(4,4)=0.25; erroneous(4,5)=0;
  erroneous(5,1)=0.2; erroneous(5,2)=0.2; erroneous(5,3)=0.2; erroneous(5,4)=0.2; erroneous(5,5)=0.2;

  %  erroneous transition matrix
  %  the probabilities of erroneous transitions between each pair of
  %  vertices.
  %   _                          _
  %  |  0.3   0     0     0.3   0.4  |
  %  |  0.25  0.25  0.25  0.25  0    |
  %  |  0.1   0.4   0.3   0     0.2  |
  %  |  0.25  0.25  0.25  0.25  0    |
  %  |_ 0.2   0.2   0.2   0.2   0.2 _|

  %set up probs. that a vertex has an error
  %vertex_errors=[0.2; 0.1; 0.4; 0.3; 0.1];
  vertex_errors=[0.02; 0.01; 0.04; 0.03; 0.01];%test

  %set up vertex check keys
  keys=[1 0 1;0 1 1;1 1 0;1 1 1;0 0 1];

%function to traverse graph once and generate key for traversal
function          [key_sequence]=traverse_graph(graph,          erroneous,          keys,
graph_size,vertex_errors,no_of_transitions,sequence_length)

  current_vertex=1;

  generated_key=keys(1,:);

  for k=2:no_of_transitions
    possible_transitions=zeros(1,2);
    no_of_possible_transitions=0;
    for l=1:graph_size
      if graph(current_vertex,l)==1;
        no_of_possible_transitions=no_of_possible_transitions+1;
        possible_transitions(1,no_of_possible_transitions)=l;
      end %if
    end %for

    %decide whether vertex is going to be in error
```

```
    %decider=rand;
    decider=1;%test
    if decider<vertex_errors(current_vertex,1);
      proper_transition=0;
    else proper_transition=1;
    end

    if proper_transition==0
      decider=rand;

      jump_vertex=1;
      cumulative=erroneous(current_vertex,jump_vertex);
      while(decider > cumulative)
        jump_vertex=jump_vertex+1;
        cumulative=cumulative+erroneous(current_vertex,jump_vertex);
      end %while

      current_vertex=jump_vertex;
    else
      if no_of_possible_transitions==1
        current_vertex=possible_transitions(1,1);
      else
        decider=rand;
        if decider < 0.5
          current_vertex=possible_transitions(1,1);
        else
          current_vertex=possible_transitions(1,2);
        end %if_else
      end %if_else
    end %if_else

    current_vertex%test
    generated_key=[generated_key keys(current_vertex,:)];
  end %main for

  %change this line to change error coding scheme used
  coded_key = parity(generated_key);

  %pad out coded_key to fit full length of allowable sequence length
  pad=size(coded_key);
  pad=pad(1,2);
  pad=zeros(1,sequence_length-pad);
  coded_key=[coded_key pad];

  %return generated key padded with zeros
  key_sequence=coded_key;


%function returns parity coded key
function coded_key = parity(generated_key)
  %check for MATLAB parity generator

  parity_counter=size(generated_key);
  parity_counter=parity_counter(1,2);

  parity_bit=generated_key(1,1);
  for loop_counter=1:parity_counter
    parity_bit=bitxor(parity_bit,generated_key(1,loop_counter));
  end

  coded_key=[generated_key parity_bit];
```

1.4.2 bit_position_shorten_opt.m

```
%bit_position_shorten_opt.m
```

```
%Karthik Subramaniam
%Institute of Information Sciences and Technology
%Turitea Campus, Massey University

%File to generate bit sequences with shortening
%This version selects an error combination, and
%uses this pattern on all the bit sequences
%The erroneous keys are then passed to an error coding technique
%and checked for detected errors

function percentage = shorten(shorten_length)

sequence_length=8; % 16-bit sequences
end_sequence=pow2(sequence_length); % last value, for counting up in decimal
shorten_length=1; % length that sequence is altered by for errors
no_of_error_combinations=nchoosek(sequence_length, shorten_length);
error_overhead=1; % extra spaces added for error-checking data
% resultant_sequences holds the bit sequences after they have been altered
% resultant_sequences=zeros(end_sequence,sequence_length-shorten_length);
counter=1;%debug
coverage=zeros(no_of_error_combinations,1);

% define coding type, used while encoding and decoding
% parity = 1
coding_type=1;

% in both the following matrices:
%       > 1st index is the decimal value of the sequence
%       > 2nd index is the particular combination of errors in the run
%  So, for example, for 2-bit errors there will be 16C2 columns

coefficients_vector=[1:sequence_length];

% generate coefficients/indices
error_coefficients_table=nchoosek(coefficients_vector, shorten_length);

% stores '1' for detected errors, '0' for undetected errors
%error_detection_map=zeros(end_sequence,no_of_error_combinations);

% go through 16C(shorten_length) combinations
for current_error_combination = 1:no_of_error_combinations

% go through all 2^16 combinations
%for current_key_dec = 0:end_sequence-1
%       current_key_bin=dec2bin(current_key_dec,sequence_length)
%   if mod(current_key_dec,1000)==0
%     shorten_length
%     current_key_dec%debug
%     current_key_bin%debug
%   end

        % go through all 2^16 combinations
        for current_key_dec = 0:end_sequence-1
                current_key_bin=dec2bin(current_key_dec,sequence_length)

  % go through 16C(shorten_length) combinations
  %for current_error_combination = 1:no_of_error_combinations
    %remaining_sequence=current_key_bin;

    %remove bits to shorten by
    delete_bits=error_coefficients_table(current_error_combination,:);
    resultant_sequence=current_key_bin;
    resultant_sequence(1,delete_bits)=' ';

    %find white spaces
    white_spaces=~isspace(resultant_sequence);

    %find positions of the white spaces
    [values,positions]=find(white_spaces);
```

```
    %form string of only the valid bits without white spaces
    resultant_sequence=resultant_sequence(1,positions);

    %resultant_sequence%debug

   % do signature compression
                    [resultant_sequence,correct_sequence]=compress_3_0_opt(resultant_sequence,
current_key_bin, coding_type);

                    % do signature checking part
   coverage(current_error_combination,1)        =        coverage(current_error_combination,1)        +
error_detection(resultant_sequence, correct_sequence, coding_type);
   %error_detection_map(current_key_dec+1, current_error_combination)=sanj;
   %counter = counter +1;

        end % end for loop for current error combination

  %resultant_sequence%=resultant_sequence(1,2)

%debug

end % end for loop for current bit sequence

%counter
%error_detection_map%debug

%percentage=(sum(error_detection_map)/(end_sequence*no_of_error_combinations))*100;
%percentage=sum(percentage);
percentage=(coverage/(end_sequence))*100;

% ---------- end main shorten function ----------

% function to select error coding scheme
%function   [erroneous_sequence,actual_sequence]   =   compress(resultant_sequence,   current_key_bin,
coding_type)
        %fprintf('in compression function');%debug

  %dummy assignment%debug
%       coded_sequence=resultant_sequence;%debug

% add switch statement for different error coding schemes

% ---------- end error encoding selection function ----------

% function to select decoding scheme
function error_detection_result = error_detection(resultant_sequence, current_key_dec, coding_type)
% This function simply calculates the difference between the correct and
% erroneous sequences

difference_vector=resultant_sequence-current_key_dec;

% check if result was zero
if sum(abs(difference_vector))==0
  error_detection_result=0;
else
  error_detection_result=1;
end

% ---------- end error decoding selection function ----------
```

1.4.3 compress_1_0_opt.m

```
%compress_1_0_opt.m

%Karthik Subramaniam
```

```
%Institute of Information Sciences and Technology
%Turitea Campus, Massey University

%Signature Compression for two bit streams - an erroneous one
%and its correct counterpart
%Compressor: x+1

function [erroneous_sequence,actual_sequence] = compress(resultant_sequence, current_key_bin,
coding_type)

% Erroneous sequence shift register
erroneous_sr=zeros(1,2);
% True sequence shift register
actual_sr=zeros(1,2);

err_seq_length=size(resultant_sequence,2);
true_seq_length=size(current_key_bin,2);

s = zeros(err_seq_length,1);
for a = 1:1:err_seq_length
  s(a) = str2num(resultant_sequence(a));
end



% Form signature for erroneous sequence
for shift=1:err_seq_length

  % Save 4th bit to avoid over-writing it
  temp_reg=xor(xor(erroneous_sr(1,2),erroneous_sr(1,1)),s(shift));

  % 2nd bit = x^1
  erroneous_sr(1,2)=erroneous_sr(1,1);

  % 1st bit = x^4 xor x^1 xor 1
  erroneous_sr(1,1)=temp_reg;

end

s = zeros(true_seq_length,1);
for a = 1:1:true_seq_length
  s(a) = str2num(current_key_bin(a));
end


% Form signature for true sequence
for shift=1:true_seq_length

  % Save 4th bit to avoid over-writing it
  temp_reg=xor(xor(actual_sr(1,2),actual_sr(1,1)),s(shift));

  % 2nd bit = x^1
  actual_sr(1,2)=actual_sr(1,1);

  % 1st bit = x^4 xor x^1 xor 1
  actual_sr(1,1)=temp_reg;
end

erroneous_sequence=erroneous_sr;
actual_sequence=actual_sr;

%erroneous_sequence
%actual_sequence
```

1.4.4 compress_3_0_opt.m

```
%compress_3_0_opt.m

%Karthik Subramaniam
%Institute of Information Sciences and Technology
%Turitea Campus, Massey University

%Signature Compression for two bit streams - an erroneous one
%and its correct counterpart
%Compressor: x^3+1

function [erroneous_sequence,actual_sequence] = compress(resultant_sequence, current_key_bin,
coding_type)

% Erroneous sequence shift register
erroneous_sr=zeros(1,4);
% True sequence shift register
actual_sr=zeros(1,4);

err_seq_length=size(resultant_sequence,2);
true_seq_length=size(current_key_bin,2);

s = zeros(err_seq_length,1);
for a = 1:1:err_seq_length
  s(a) = str2num(resultant_sequence(a));
end


% Form signature for erroneous sequence
for shift=1:err_seq_length

  % Save 4th bit to avoid over-writing it
  temp_reg=xor(xor(erroneous_sr(1,4),erroneous_sr(1,3)),s(shift));

  %erroneous_sr(1,5)=erroneous_sr(1,4);

  % 4th bit = x^3
  erroneous_sr(1,4)=erroneous_sr(1,3);

  % 3rd bit = x^2
  erroneous_sr(1,3)=erroneous_sr(1,2);

  % 2nd bit = x^1
  erroneous_sr(1,2)=erroneous_sr(1,1);

  % 1st bit = x^4 xor x^1 xor 1
  erroneous_sr(1,1)=temp_reg;

end

s = zeros(true_seq_length,1);
for a = 1:1:true_seq_length
  s(a) = str2num(current_key_bin(a));
end


% Form signature for true sequence
for shift=1:true_seq_length

  % Save 4th bit to avoid over-writing it
  temp_reg=xor(xor(actual_sr(1,4),actual_sr(1,3)),s(shift));

  % 4th bit = x^3
  %actual_sr(1,5)=actual_sr(1,4);

  % 4th bit = x^3
  actual_sr(1,4)=actual_sr(1,3);

  % 3rd bit = x^2
  actual_sr(1,3)=actual_sr(1,2);
```

```
  % 2nd bit = x^1
  actual_sr(1,2)=actual_sr(1,1);

  % 1st bit = x^4 xor x^1 xor 1
  actual_sr(1,1)=temp_reg;
end

erroneous_sequence=erroneous_sr;
actual_sequence=actual_sr;

%erroneous_sequence
%actual_sequence
```

1.4.5 compress_6_1_opt.m

```
%compress_6_1_opt.m

%Karthik Subramaniam
%Institute of Information Sciences and Technology
%Turitea Campus, Massey University

%Signature Compression for two bit streams - an erroneous one
%and its correct counterpart
%Compressor: x^6 + x^1

function [erroneous_sequence,actual_sequence] = compress(resultant_sequence, current_key_bin,
coding_type)

% Erroneous sequence shift register
erroneous_sr=zeros(1,7);
% True sequence shift register
actual_sr=zeros(1,7);

err_seq_length=size(resultant_sequence,2);
true_seq_length=size(current_key_bin,2);

s = zeros(err_seq_length,1);
for a = 1:1:err_seq_length
  s(a) = str2num(resultant_sequence(a));
end


% Form signature for erroneous sequence
for shift=1:err_seq_length

  % Save 4th bit to avoid over-writing it
  temp_reg=xor(xor(erroneous_sr(1,7),erroneous_sr(1,3)),s(shift));

  erroneous_sr(1,7)=erroneous_sr(1,6);

  erroneous_sr(1,6)=erroneous_sr(1,5);

  erroneous_sr(1,5)=erroneous_sr(1,4);

  % 4th bit = x^3
  erroneous_sr(1,4)=erroneous_sr(1,3);

  % 3rd bit = x^2
  erroneous_sr(1,3)=erroneous_sr(1,2);

  % 2nd bit = x^1
  erroneous_sr(1,2)=erroneous_sr(1,1);
```

```
  % 1st bit = x^4 xor x^1 xor 1
  erroneous_sr(1,1)=temp_reg;

end

s = zeros(true_seq_length,1);
for a = 1:1:true_seq_length
  s(a) = str2num(current_key_bin(a));
end


% Form signature for true sequence
for shift=1:true_seq_length

  % Save 4th bit to avoid over-writing it
  temp_reg=xor(xor(actual_sr(1,7),actual_sr(1,3)),s(shift));

  actual_sr(1,7)=actual_sr(1,6);

  actual_sr(1,6)=actual_sr(1,5);

  actual_sr(1,5)=actual_sr(1,4);

  % 4th bit = x^3
  actual_sr(1,4)=actual_sr(1,3);

  % 3rd bit = x^2
  actual_sr(1,3)=actual_sr(1,2);

  % 2nd bit = x^1
  actual_sr(1,2)=actual_sr(1,1);

  % 1st bit = x^4 xor x^1 xor 1
  actual_sr(1,1)=temp_reg;
end

erroneous_sequence=erroneous_sr
actual_sequence=actual_sr

%erroneous_sequence
%actual_sequence
```

1.4.6 edge_detector.m

```
%edge_detector.m

%Karthik Subramaniam
%Institute of Information Sciences and Technology
%Turitea Campus, Massey University

%Edge detector - sums the transitions in a bit sequence
%Returns a '1' if error detected, or '0' if error undetected

function [edge_detector_result] = edge_detector(resultant_sequence, current_key_bin, coding_type)

% calculate sizes of both sequences
true_seq_length=size(current_key_bin,2);
err_seq_length=size(resultant_sequence,2);

true_sequence_sum=0;
resultant_sequence_sum=0;

% convert both strings to vectors
%s1 = zeros(true_seq_length-1,1);
%for a = 1:true_seq_length
%   s1(a) = str2num(current_key_bin(a));
```

```
%end

%s2 = zeros(err_seq_length-1,1);
%for a = 1:err_seq_length
%   s2(a) = str2num(resultant_sequence(a));
%end

% find edges in true sequence
for a = 2:true_seq_length
  if current_key_bin(a)~=current_key_bin(a-1)
    true_sequence_sum=true_sequence_sum+1;
  end
end

% find edges in true sequence
for a = 2:err_seq_length
  if resultant_sequence(a)~=resultant_sequence(a-1)
    resultant_sequence_sum=resultant_sequence_sum+1;
  end
end

%true_sequence_sum%debug
%resultant_sequence_sum%debug

if true_sequence_sum-resultant_sequence_sum==0
  edge_detector_result=0;
else
  edge_detector_result=1;
end
```

1.4.7 Lengthen_consec_alt.m

```
%Lengthen_consec_alt.m

%Karthik Subramaniam
%Institute of Information Sciences and Technology
%Turitea Campus, Massey University

%Inserts bits into the bit sequence in blocks
%All bits are inserted together
%Optimised for LFSR
%After lengthening, the keys are sent to
%an error detection mechanism

function percentage = Lengthen_new(lengthen_size)

sequence_length=6; % n-bit sequences
end_sequence=pow2(sequence_length); % last value, for counting up in decimal
lengthen_size = 4; % no. of bits to lengthen keys by
insertion_sequence = '1001';% this bit pattern is inserted into the keys
% if a specific pattern is required, it can be defined:
% e.g. if for lengthening by 3 bits, the sequence 1 0 1 is to be inserted,
% it can be defined as '101'
no_of_error_combinations=sequence_length+1;
temp_sum=0;
counter=0;%debug

error_coefficients_table=1:no_of_error_combinations;

% go through all 2^n sequences
for current_key_dec = 0:end_sequence-1
        current_key_bin=dec2bin(current_key_dec,sequence_length);

  % do all the combinations for this bit sequence
  for current_error_combination = 1:no_of_error_combinations
```

```
        counter=counter+1;%debug

                if current_error_combination == 1
        resultant_sequence = [insertion_sequence current_key_bin];
        else if current_error_combination == no_of_error_combinations
                resultant_sequence = [current_key_bin insertion_sequence];
    else
      temp_seq = current_key_bin(1:current_error_combination-1);
                                                resultant_sequence = [insertion_sequence
current_key_bin(current_error_combination:sequence_length)];
          resultant_sequence = [temp_seq resultant_sequence];
    end
    end

        % do signature compression
        [resultant_sequence,correct_sequence]=compress_3_0_opt(resultant_sequence, current_key_bin);

        temp_sum = temp_sum + error_detection(resultant_sequence, correct_sequence);

        %current_key_bin%debug

        end % end for loop for current error combination

end

percentage=(temp_sum/(end_sequence*no_of_error_combinations))*100
counter

% ----------end of lengthening part---------- %

function error_detection_result = error_detection(resultant_sequence_compressed, correct_key_compressed)
% This function simply calculates the difference between the correct and
% erroneous sequences

difference_vector=resultant_sequence_compressed-correct_key_compressed;

% check if result was zero
if sum(abs(difference_vector))==0
  error_detection_result=0;
else
  error_detection_result=1;
end

% ---------- end error decoding selection function ----------
```

1.4.8 Lengthen_consec_new.m

```
%Lengthen_consec_new.m

%Karthik Subramaniam
%Institute of Information Sciences and Technology
%Turitea Campus, Massey University

%Inserts bits into the bit sequence in consecutive positions
%Optimised for LFSR
%The erroneous and correct keys are sent
%to an error coding algorithm

function percentage = Lengthen_new(lengthen_size)

sequence_length=4; % n-bit sequences
end_sequence=pow2(sequence_length); % last value, for counting up in decimal
lengthen_size = 3; % no. of bits to lengthen keys by
insertion_sequence = '1';% this bit pattern is inserted into the keys
% if a specific pattern is required, it can be defined:
% e.g. if for lengthening by 3 bits, the sequence 1 0 1 is to be inserted,
```

```
% it can be defined as '101'
no_of_error_combinations=sequence_length+1;
temp_sum=0;
counter=0;%debug

error_coefficients_table=zeros(no_of_error_combinations, lengthen_size);
for j=1:no_of_error_combinations
  error_coefficients_table(j,:)=[j:j+lengthen_size-1];
end

% go through all 2^n sequences
for current_key_dec = 0:end_sequence-1
        current_key_bin=dec2bin(current_key_dec,sequence_length);

        % create spaces between bits in character string
        for i=1:sequence_length
    template(1,2*i-1)=' ';
    template(1,2*i)=current_key_bin(1,i);
  end
  end_bits=[2*sequence_length+1:2*sequence_length+lengthen_size];
  template(1,end_bits)=' ';

  % do all the combinations for this bit sequence
  for current_error_combination = 1:no_of_error_combinations

  counter=counter+1;%debug

        % select positions to insert into
        insert_bits=error_coefficients_table(current_error_combination,:);
  resultant_sequence=template;

  insert_bits=2*insert_bits-1;
  for k=1:lengthen_size
    if insert_bits(1,k)>2*sequence_length+lengthen_size;
      insert_bits(1,k)=insert_bits(1,k)-(lengthen_size-1);
    end
  end

  % insert the defined bit sequence
  resultant_sequence(1,insert_bits)=insertion_sequence;

  % find all gaps
  white_spaces=~isspace(resultant_sequence);

  [values,positions]=find(white_spaces);

  resultant_sequence=resultant_sequence(1,positions);

  % do signature compression
  [resultant_sequence,correct_sequence]=compress_1_0_opt(resultant_sequence, current_key_bin);

  temp_sum = temp_sum + error_detection(resultant_sequence, correct_sequence);

  %current_key_bin%debug

end % end for loop for current error combination

end

percentage=(temp_sum/(end_sequence*no_of_error_combinations))*100
counter

% ----------end of lengthening part---------- %

function error_detection_result = error_detection(resultant_sequence_compressed, correct_key_compressed)
% This function simply calculates the difference between the correct and
% erroneous sequences

difference_vector=resultant_sequence_compressed-correct_key_compressed;
```

```
% check if result was zero
if sum(abs(difference_vector))==0
  error_detection_result=0;
else
  error_detection_result=1;
end

% ---------- end error decoding selection function ----------
```

1.4.9 Lengthen_new.m

```
%Lengthen_new.m

%Karthik Subramaniam
%Institute of Information Sciences and Technology
%Turitea Campus, Massey University

%Inserts bits into the bit sequence in all possible error combinations
%Optimised for LFSR

function percentage = Lengthen_new(lengthen_size)

sequence_length=8; % n-bit sequences
end_sequence=pow2(sequence_length); % last value, for counting up in decimal
lengthen_size = 1; % no. of bits to lengthen keys by
insertion_sequence = '1';% this bit pattern is inserted into the keys
% if a specific pattern is required, it can be defined:
% e.g. if for lengthening by 3 bits, the sequence 1 0 1 is to be inserted,
% it can be defined as '101'
no_of_error_combinations=nchoosek(sequence_length+lengthen_size, lengthen_size);
temp_sum=0;
counter=0;%debug

coefficients_vector=[1:sequence_length+lengthen_size];

% vector storing all the combinations in which the bits will be deleted
error_coefficients_table=nchoosek(coefficients_vector, lengthen_size);

% go through all 2^n sequences
for current_key_dec = 0:end_sequence

        current_key_bin=dec2bin(current_key_dec,sequence_length);

        % create spaces between bits in character string
        for i=1:sequence_length
    template(1,2*i-1)=' ';
    template(1,2*i)=current_key_bin(1,i);
  end
  end_bits=[2*sequence_length+1:2*sequence_length+lengthen_size];
  template(1,end_bits)=' ';

  % do all the combinations for this bit sequence
  for current_error_combination = 1:no_of_error_combinations

  counter=counter+1;%debug

        % select positions to insert into
        insert_bits=error_coefficients_table(current_error_combination,:);
  resultant_sequence=template;

  insert_bits=2*insert_bits-1;
  for k=1:lengthen_size
    if insert_bits(1,k)>2*sequence_length+lengthen_size;
      insert_bits(1,k)=insert_bits(1,k)-(lengthen_size-1);
    end
  end
```

```
  % insert the defined bit sequence
  resultant_sequence(1,insert_bits)=insertion_sequence;

  % find all gaps
  white_spaces=~isspace(resultant_sequence);

  [values,positions]=find(white_spaces);

  resultant_sequence=resultant_sequence(1,positions);

%   current_key_dec%debug
%   current_key_bin%debug

  % do signature compression
  [resultant_sequence,correct_sequence]=compress_3_0_opt(resultant_sequence, current_key_bin);

  temp_sum = temp_sum + error_detection(resultant_sequence, correct_sequence);


end % end for loop for current error combination

end

percentage=(temp_sum/(end_sequence*no_of_error_combinations))*100
counter

% ----------end of lengthening part---------- %

function error_detection_result = error_detection(resultant_sequence_compressed, correct_key_compressed)
% This function simply calculates the difference between the correct and
% erroneous sequences

difference_vector=resultant_sequence_compressed-correct_key_compressed;

% check if result was zero
if sum(abs(difference_vector))==0
  error_detection_result=0;
else
  error_detection_result=1;
end

% ---------- end error decoding selection function ----------
```

1.4.10 Lengthen_position_opt.m

```
%Lengthen_position_opt.m

%Karthik Subramaniam
%Institute of Information Sciences and Technology
%Turitea Campus, Massey University

%Inserts bits into the bit sequence in blocks
%All bits are inserted together
%Optimised for LFSR
%separates the results by bit position

function percentage = Lengthen_new(lengthen_size)

sequence_length=8; % n-bit sequences
end_sequence=pow2(sequence_length); % last value, for counting up in decimal
lengthen_size = 2; % no. of bits to lengthen keys by
insertion_sequence = '11';% this bit pattern is inserted into the keys
% if a specific pattern is required, it can be defined:
% e.g. if for lengthening by 3 bits, the sequence 1 0 1 is to be inserted,
% it can be defined as '101'
```

```
no_of_error_combinations=sequence_length+1;
temp_sum=zeros(no_of_error_combinations,1);
counter=0;%debug

error_coefficients_table=1:no_of_error_combinations;

% go through all 2^n sequences
for current_key_dec = 0:end_sequence-1
        current_key_bin=dec2bin(current_key_dec,sequence_length);

  % do all the combinations for this bit sequence
  for current_error_combination = 1:no_of_error_combinations

                counter=counter+1;%debug

                if current_error_combination == 1
        resultant_sequence = [insertion_sequence current_key_bin];
        else if current_error_combination == no_of_error_combinations
                resultant_sequence = [current_key_bin insertion_sequence];
    else
      temp_seq = current_key_bin(1:current_error_combination-1);
                                        resultant_sequence = [insertion_sequence
current_key_bin(current_error_combination:sequence_length)];
          resultant_sequence = [temp_seq resultant_sequence];
    end
    end

        % do signature compression
        [resultant_sequence,correct_sequence]=compress_3_0_opt(resultant_sequence, current_key_bin);

        temp_sum(current_error_combination,1) = temp_sum(current_error_combination,1) +
error_detection(resultant_sequence, correct_sequence);

        %current_key_bin%debug

        end % end for loop for current error combination

end

percentage=(temp_sum/(end_sequence))*100
counter;

% ----------end of lengthening part---------- %

function error_detection_result = error_detection(resultant_sequence_compressed, correct_key_compressed)
% This function simply calculates the difference between the correct and
% erroneous sequences

difference_vector=resultant_sequence_compressed-correct_key_compressed;

% check if result was zero
if sum(abs(difference_vector))==0
  error_detection_result=0;
else
  error_detection_result=1;
end

% ---------- end error decoding selection function ----------
```

1.4.11 parity_encode.m

```
%parity_encode.m

%Karthik Subramaniam
%Institute of Information Sciences and Technology
%Turitea Campus, Massey University
```

```
%Parity encoder
%Generates even parity for the true bit sequence
%Once the parity bit has been calculated for the true sequence,
%it is appended to the erroneous sequence as well

function [parity_result] = parity_encode(resultant_sequence, current_key_bin, coding_type)

% ind sizes of both sequences
true_seq_length=size(current_key_bin,2);
err_seq_length=size(resultant_sequence,2);

% convert both strings to vectors
s1 = zeros(true_seq_length,1);
for a = 1:1:true_seq_length
  s1(a) = str2num(current_key_bin(a));
end

s2 = zeros(err_seq_length,1);
for a = 1:1:err_seq_length
  s2(a) = str2num(resultant_sequence(a));
end

% find parity of both sequences
% if parities are the same, then the erroneous sequence
% would still hold the parity check, even after deletion
% e.g.
% true sequence 10110101 -> even parity = 1
% erroneous sequence (a) 1110101 -> even parity = 1 -> undetected
% erroneous sequence (b) 1010101 -> even parity = 0 -> detected
parity_bit1=mod(sum(s1),2);
parity_bit2=mod(sum(s2),2);

% check if parity bits calculated would be the same
if parity_bit1==parity_bit2
  % if they are equal, error goes undetected
  parity_result=0;
else
  % results are different, so error is detected
  parity_result=1;
end
```

1.4.12 position_check_opt.m

```
%position_check_opt.m

%Karthik Subramaniam
%Institute of Information Sciences and Technology
%Turitea Campus, Massey University

%File to generate bit sequences with shortening
%Bit sequences are then compressed and a signature is formed
%These signatures are then analysed to find the differences between
%the signatures of the erroneous sequences, and their corresponding
%correct sequences
%This version differs from shorten_consec_new.m in that
%it separates the results by position
%Thus, for 1-bit errors in 8-bit sequences, it generates a matrix with
%8 results - one for the fault coverage of each deletion position

function percentage = shorten(shorten_length)

sequence_length=8; % 16-bit sequences
end_sequence=pow2(sequence_length); % last value, for counting up in decimal
shorten_length=1; % length that sequence is altered by for errors
no_of_error_combinations=sequence_length-shorten_length+1;
```

```
error_overhead=1; % extra spaces added for error-checking data
% resultant_sequences holds the bit sequences after they have been altered
% resultant_sequences=zeros(end_sequence,sequence_length-shorten_length);
counter=1;%debug
coverage=zeros(no_of_error_combinations,1);

% define coding type, used while encoding and decoding
% parity = 1
coding_type=1;

% in both the following matrices:
%       > 1st index is the decimal value of the sequence
%       > 2nd index is the particular combination of errors in the run
%  So, for example, for 2-bit errors there will be 16C2 columns

error_coefficients_table=zeros(no_of_error_combinations, shorten_length);
for j=1:no_of_error_combinations
  error_coefficients_table(j,:)=[j:j+shorten_length-1];
end

% stores '1' for detected errors, '0' for undetected errors
%error_detection_map=zeros(end_sequence,no_of_error_combinations);

% go through all 2^16 combinations
for current_key_dec = 0:end_sequence-1
        current_key_bin=dec2bin(current_key_dec,sequence_length);
  if mod(current_key_dec,1000)==0
    shorten_length
    current_key_dec%debug
    current_key_bin%debug
  end
 % go through 16C(shorten_length) combinations
 for current_error_combination = 1:no_of_error_combinations
    %remaining_sequence=current_key_bin;

    %remove bits to shorten by
    delete_bits=error_coefficients_table(current_error_combination,:);
    resultant_sequence=current_key_bin;
    resultant_sequence(1,delete_bits)=' ';

    %find white spaces
    white_spaces=~isspace(resultant_sequence);

    %find positions of the white spaces
    [values,positions]=find(white_spaces);

    %form string of only the valid bits without white spaces
    resultant_sequence=resultant_sequence(1,positions);

    %resultant_sequence%debug

   % do signature compression
                [resultant_sequence,correct_sequence]=compress_1_0_opt(resultant_sequence,
current_key_bin, coding_type);

                % do signature checking part
   coverage(current_error_combination,1)        =        coverage(current_error_combination,1)        +
error_detection(resultant_sequence, correct_sequence, coding_type);
   %error_detection_map(current_key_dec+1, current_error_combination)=sanj;
   %counter = counter +1;

        end % end for loop for current error combination

  %resultant_sequence%=resultant_sequence(1,2)

%debug

end % end for loop for current bit sequence

%counter
```

```
%error_detection_map%debug

%percentage=(sum(error_detection_map)/(end_sequence*no_of_error_combinations))*100;
%percentage=sum(percentage);
percentage=(coverage/(end_sequence))*100;

% ---------- end main shorten function ----------

% function to select error coding scheme
%function [erroneous_sequence,actual_sequence] = compress(resultant_sequence, current_key_bin,
coding_type)
        %fprintf('in compression function');%debug

  %dummy assignment%debug
%       coded_sequence=resultant_sequence;%debug

% add switch statement for different error coding schemes

% ---------- end error encoding selection function ----------

% function to select decoding scheme
function error_detection_result = error_detection(resultant_sequence, current_key_dec, coding_type)
% This function simply calculates the difference between the correct and
% erroneous sequences

difference_vector=resultant_sequence-current_key_dec;

% check if result was zero
if sum(abs(difference_vector))==0
  error_detection_result=0;
else
  error_detection_result=1;
end

% ---------- end error decoding selection function ----------
```

1.4.13 shorten_consec_new_opt.m

```
%shorten_consec_new_opt.m

%Karthik Subramaniam
%Institute of Information Sciences and Technology
%Turitea Campus, Massey University

%File to generate bit sequences with shortening, by deleting consecutive positions
%Bit sequences are then compressed and a signature is formed
%These signatures are then analysed to find the differences between
%the signatures of the erroneous sequences, and their corresponding
%correct sequences

function percentage = shorten(shorten_length)

sequence_length=12; % 16-bit sequences
end_sequence=pow2(sequence_length); % last value, for counting up in decimal
shorten_length=shorten_length; % length that sequence is altered by for errors
no_of_error_combinations=sequence_length-shorten_length+1;
error_overhead=1; % extra spaces added for error-checking data
% resultant_sequences holds the bit sequences after they have been altered
% resultant_sequences=zeros(end_sequence,sequence_length-shorten_length);
counter=1;%debug
sanj=0;

% define coding type, used while encoding and decoding
% parity = 1
coding_type=1;
```

```
% in both the following matrices:
%           > 1st index is the decimal value of the sequence
%           > 2nd index is the particular combination of errors in the run
%   So, for example, for 2-bit errors there will be 16C2 columns

error_coefficients_table=zeros(no_of_error_combinations, shorten_length);
for j=1:no_of_error_combinations
  error_coefficients_table(j,:)=[j:j+shorten_length-1];
end

% stores '1' for detected errors, '0' for undetected errors
%error_detection_map=zeros(end_sequence,no_of_error_combinations);

% go through all 2^16 combinations
for current_key_dec = 0:end_sequence-1
          current_key_bin=dec2bin(current_key_dec,sequence_length);
   if mod(current_key_dec,1000)==0
     shorten_length
     current_key_dec%debug
     current_key_bin%debug
   end
  % go through 16C(shorten_length) combinations
  for current_error_combination = 1:no_of_error_combinations
     %remaining_sequence=current_key_bin;

     %remove bits to shorten by
     delete_bits=error_coefficients_table(current_error_combination,:);
     resultant_sequence=current_key_bin;
     resultant_sequence(1,delete_bits)=' ';

     %find white spaces
     white_spaces=~isspace(resultant_sequence);

     %find positions of the white spaces
     [values,positions]=find(white_spaces);

     %form string of only the valid bits without white spaces
     resultant_sequence=resultant_sequence(1,positions);

     %resultant_sequence%debug

    % do signature compression
                  [resultant_sequence,correct_sequence]=compress_3_0_opt(resultant_sequence,
current_key_bin, coding_type);

                  % do signature checking part
    sanj = sanj + error_detection(resultant_sequence, correct_sequence, coding_type);
    %error_detection_map(current_key_dec+1, current_error_combination)=sanj;
    %counter = counter +1;

          end % end for loop for current error combination

  %resultant_sequence%=resultant_sequence(1,2)

%debug

end % end for loop for current bit sequence

%counter
%error_detection_map%debug

%percentage=(sum(error_detection_map)/(end_sequence*no_of_error_combinations))*100;
%percentage=sum(percentage);
percentage=(sanj/(end_sequence*no_of_error_combinations))*100;

% ---------- end main shorten function ----------

% function to select error coding scheme
%function   [erroneous_sequence,actual_sequence]   =   compress(resultant_sequence,   current_key_bin,
coding_type)
```

```
        %fprintf('in compression function');%debug

  %dummy assignment%debug
%       coded_sequence=resultant_sequence;%debug

% add switch statement for different error coding schemes

% ---------- end error encoding selection function ----------

% function to select decoding scheme
function error_detection_result = error_detection(resultant_sequence, current_key_dec, coding_type)
% This function simply calculates the difference between the correct and
% erroneous sequences

difference_vector=resultant_sequence-current_key_dec;

% check if result was zero
if sum(abs(difference_vector))==0
  error_detection_result=0;
else
  error_detection_result=1;
end

% ---------- end error decoding selection function ----------
```

1.4.14 shorten_new_opt.m

```
%shorten_new_opt.m

%Karthik Subramaniam
%Institute of Information Sciences and Technology
%Turitea Campus, Massey University

%File to generate bit sequences with shortening
%Bit sequences are then compressed and a signature is formed
%These signatures are then analysed to find the differences between
%the signatures of the erroneous sequences, and their corresponding
%correct sequences

function percentage = shorten(shorten_length)

sequence_length=8; % 16-bit sequences
end_sequence=pow2(sequence_length); % last value, for counting up in decimal
shorten_length=1; % length that sequence is altered by for errors
no_of_error_combinations=nchoosek(sequence_length, shorten_length);
error_overhead=1; % extra spaces added for error-checking data
% resultant_sequences holds the bit sequences after they have been altered
% resultant_sequences=zeros(end_sequence,sequence_length-shorten_length);
counter=1;%debug
sanj=0;

% define coding type, used while encoding and decoding
% parity = 1
coding_type=1;

% in both the following matrices:
%       > 1st index is the decimal value of the sequence
%       > 2nd index is the particular combination of errors in the run
%  So, for example, for 2-bit errors there will be 16C2 columns

coefficients_vector=[1:sequence_length];

% generate coefficients/indices
error_coefficients_table=nchoosek(coefficients_vector, shorten_length);

% stores '1' for detected errors, '0' for undetected errors
```

```
%error_detection_map=zeros(end_sequence,no_of_error_combinations);

% go through all 2^16 combinations
for current_key_dec = 0:end_sequence-1
        current_key_bin=dec2bin(current_key_dec,sequence_length);
  if mod(current_key_dec,1000)==0
    shorten_length
    current_key_dec%debug
    current_key_bin%debug
  end
  % go through 16C(shorten_length) combinations
  for current_error_combination = 1:no_of_error_combinations
    %remaining_sequence=current_key_bin;

    %remove bits to shorten by
    delete_bits=error_coefficients_table(current_error_combination,:);
    resultant_sequence=current_key_bin;
    resultant_sequence(1,delete_bits)=' ';

    %find white spaces
    white_spaces=~isspace(resultant_sequence);

    %find positions of the white spaces
    [values,positions]=find(white_spaces);

    %form string of only the valid bits without white spaces
    resultant_sequence=resultant_sequence(1,positions);

    %resultant_sequence%debug

   % do signature compression
                [resultant_sequence,correct_sequence]=compress_3_0_opt(resultant_sequence,
current_key_bin, coding_type);

                % do signature checking part
    sanj = sanj + error_detection(resultant_sequence, correct_sequence, coding_type);
    %error_detection_map(current_key_dec+1, current_error_combination)=sanj;
    %counter = counter +1;

        end % end for loop for current error combination

  %resultant_sequence%=resultant_sequence(1,2)

%debug

end % end for loop for current bit sequence

%counter
%error_detection_map%debug

%percentage=(sum(error_detection_map)/(end_sequence*no_of_error_combinations))*100;
%percentage=sum(percentage);
percentage=(sanj/(end_sequence*no_of_error_combinations))*100

% ---------- end main shorten function ----------

% function to select error coding scheme
%function  [erroneous_sequence,actual_sequence]  =  compress(resultant_sequence,  current_key_bin,
coding_type)
        %fprintf('in compression function');%debug

  %dummy assignment%debug
%       coded_sequence=resultant_sequence;%debug

% add switch statement for different error coding schemes

% ---------- end error encoding selection function ----------

% function to select decoding scheme
function error_detection_result = error_detection(resultant_sequence, current_key_dec, coding_type)
```

```
% This function simply calculates the difference between the correct and
% erroneous sequences

difference_vector=resultant_sequence-current_key_dec;

% check if result was zero
if sum(abs(difference_vector))==0
  error_detection_result=0;
else
  error_detection_result=1;
end

% ---------- end error decoding selection function ----------
```

1.4.15 syndrome_coding.m

```
%syndrome_coding.m

%Karthik Subramaniam
%Institute of Information Sciences and Technology
%Turitea Campus, Massey University

%Syndrome coding - counts number of '1's in vectors

function [syndrome_result] = syndrome_coding(resultant_sequence, current_key_bin, coding_type)

% calculate sizes of both sequences
true_seq_length=size(current_key_bin,2);
err_seq_length=size(resultant_sequence,2);

% convert both strings to vectors
s1 = zeros(true_seq_length,1);
for a = 1:1:true_seq_length
  s1(a) = str2num(current_key_bin(a));
end

s2 = zeros(err_seq_length,1);
for a = 1:1:err_seq_length
  s2(a) = str2num(resultant_sequence(a));
end

% compare numbers of 1s in both sequences
if(sum(s1)==sum(s2))
  % if no. of 1s is the same, error goes undetected
  syndrome_result=0;
else
  % if no. of 1s differs, error is detected
  syndrome_result=1;
end
```

Appendix 2

Simulation Results

2.1 Signature Analysis

2.1.1 Key Shortening

The following graphs illustrate the results obtained from the 4-bit compression register, for the 8-bit keys.

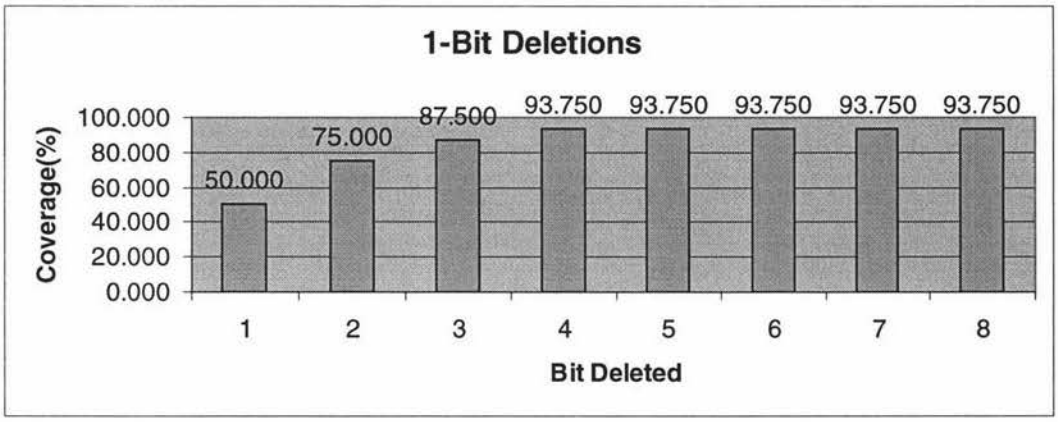


Figure A0-1: Signature analysis: 8-bit key, 4-bit compression, 1-bit deletion.

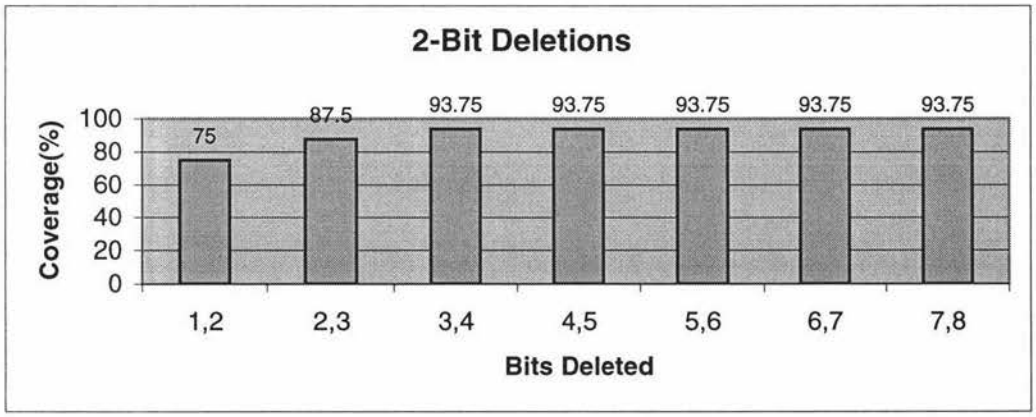


Figure A0-2: Signature analysis: 8-bit key, 4-bit compression, 2-bit deletion.

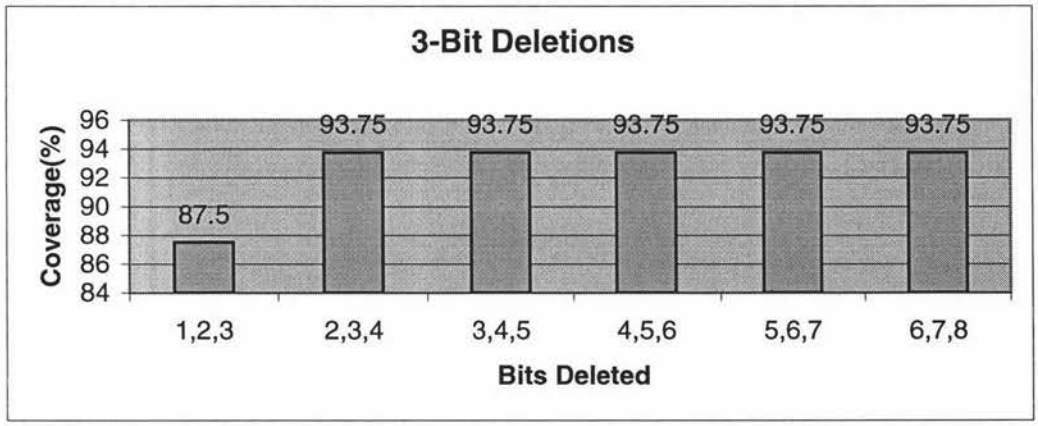


Figure A0-3: Signature analysis: 8-bit key, 4-bit compression, 3-bit deletion.

12-bit keys were also used for the simulations. All the possible combinations of bit sequences were used, producing a total of 2^12 = 4096 combinations for each set of errors. A 7-bit compression technique was used, using the polynomial x^6+x^5.

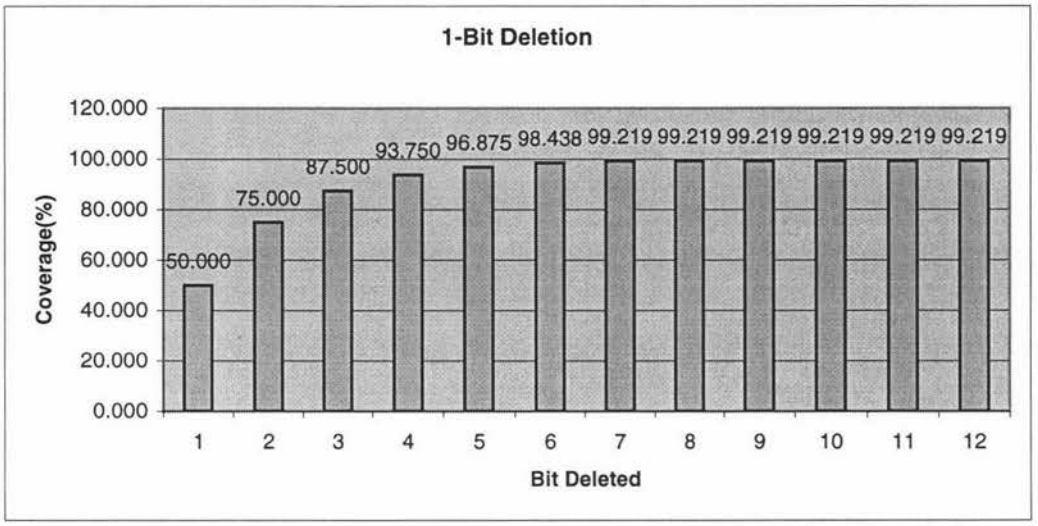


Figure A0-4: Signature analysis: 12-bit key, 7-bit compression, 1-bit deletion.

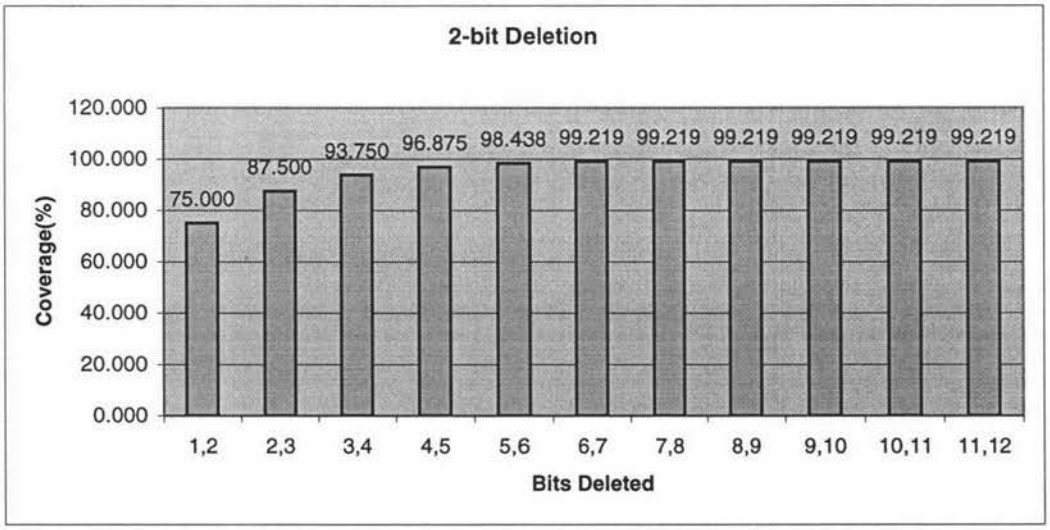


Figure A0-5: Signature analysis: 12-bit key, 7-bit compression, 2-bit deletion.

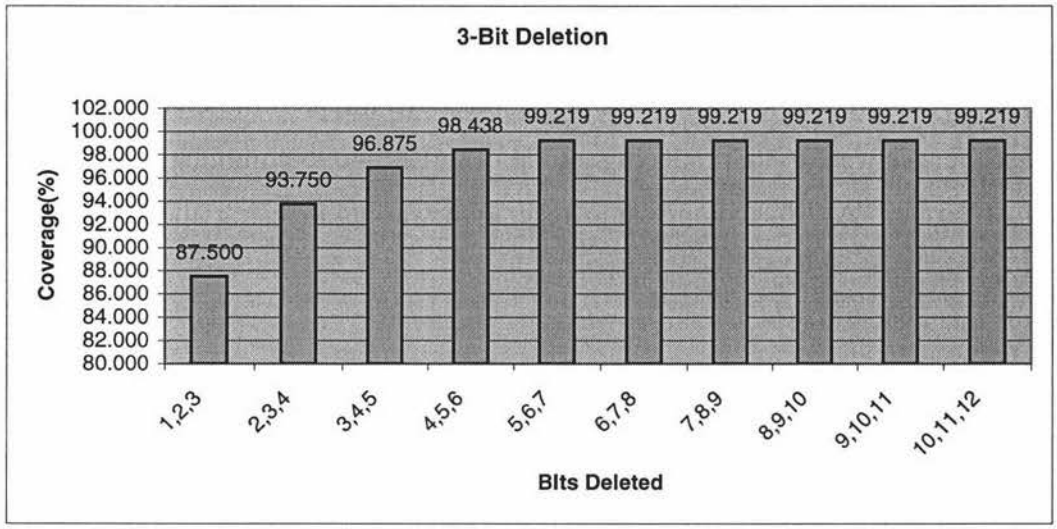


Figure A0-6: Signature analysis: 12-bit key, 7-bit compression, 3-bit deletion.

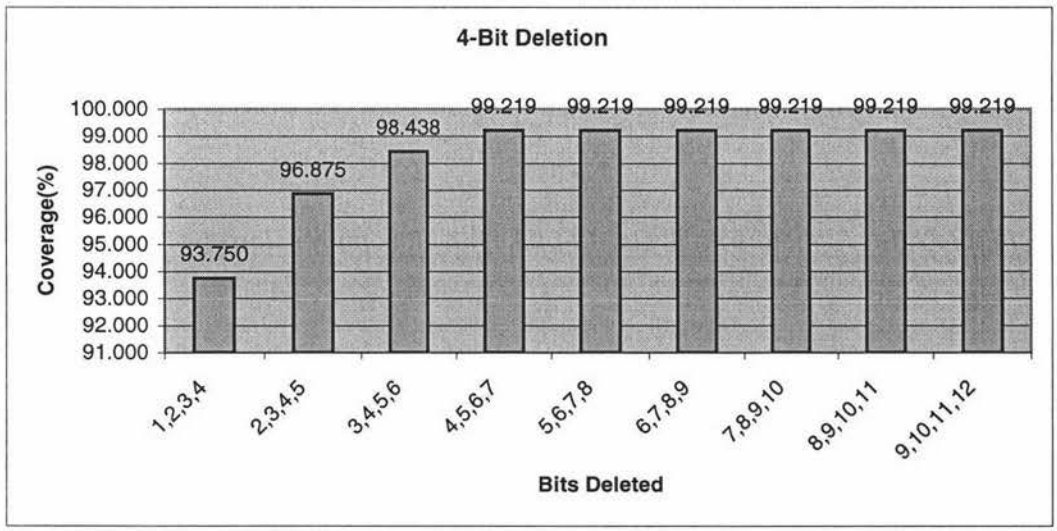


Figure A0-7: Signature analysis: 12-bit key, 7-bit compression, 4-bit deletion.

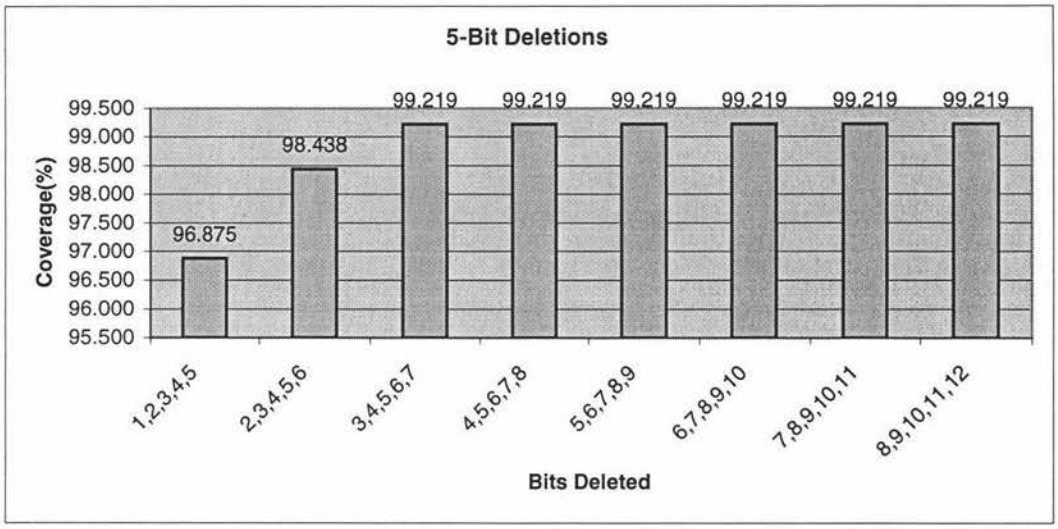


Figure A0-8 Signature analysis: 12-bit key, 7-bit compression, 5-bit deletion.

2.1.2 Key Lengthening

The following graphs are for 8-bit key lengthening, with 4-bit compression.

Figure A0-9: Signature analysis: 8-bit key, 4-bit compression, 1-bit lengthening by inserted error.

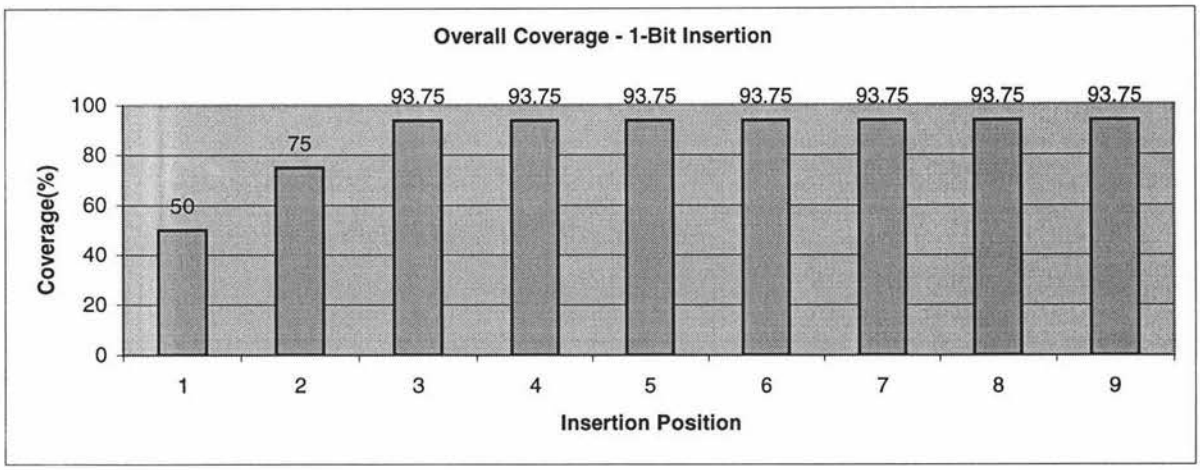


Figure A0-10: Signature analysis: 8-bit key, 4-bit compression, 1-bit lengthening. Average Coverage = 86.11%.

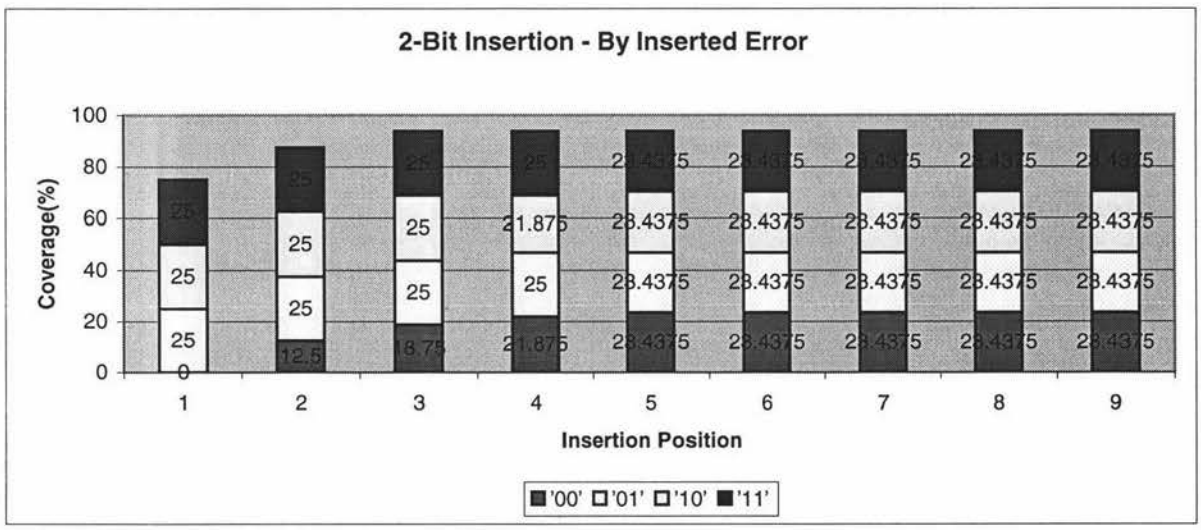


Figure A0-11: Signature analysis: 8-bit key, 4-bit compression, 2-bit lengthening by inserted error.

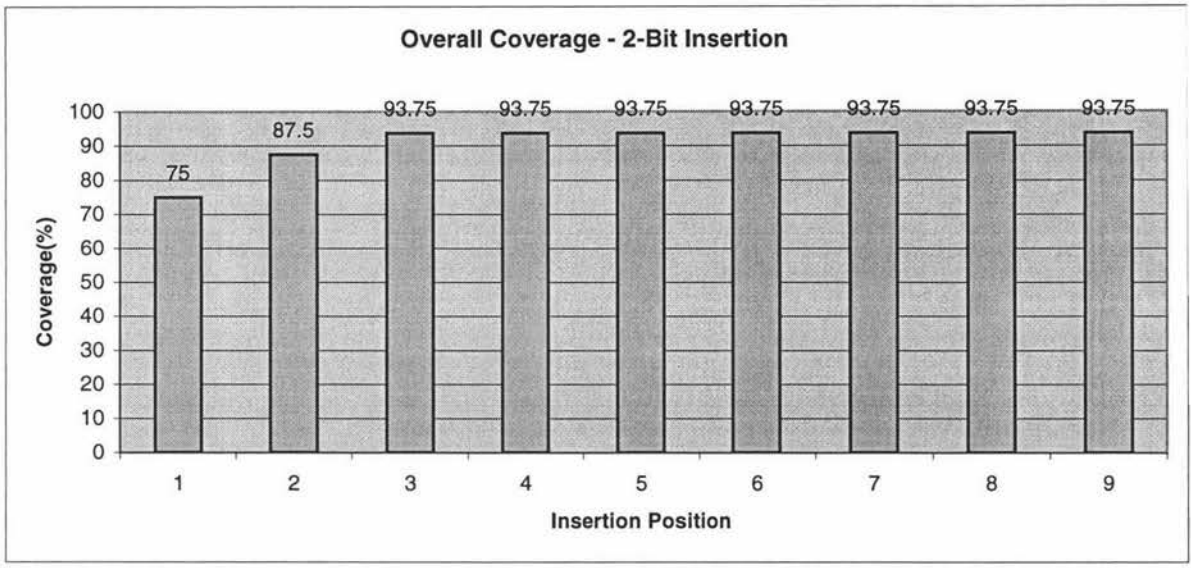


Figure A0-12: Signature analysis: 8-bit key, 4-bit compression, 2-bit lengthening. Average Coverage = 90.97%.

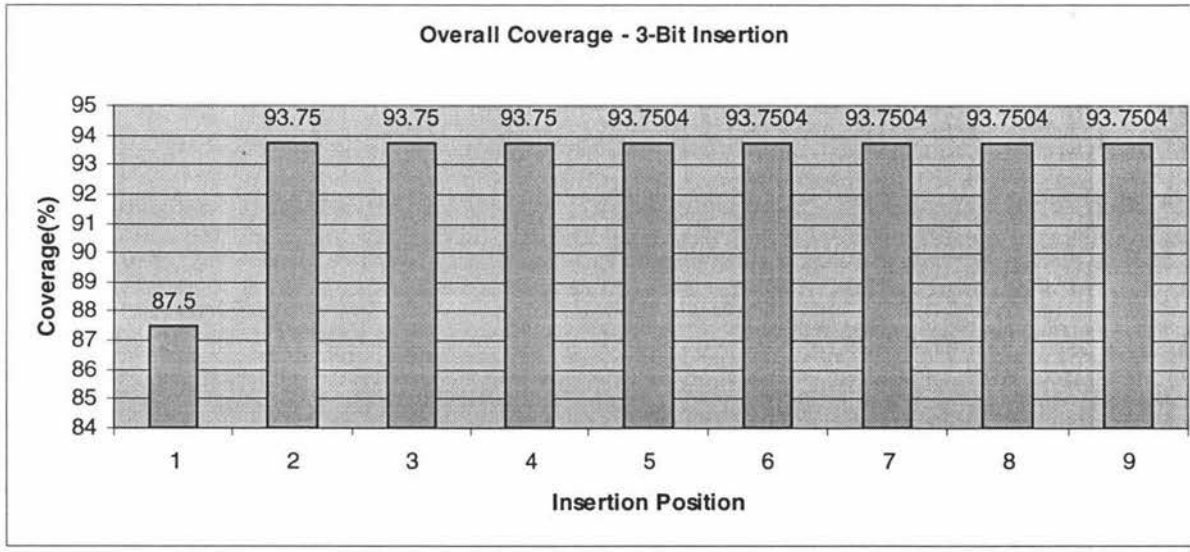


Figure A0-13: Signature analysis: 8-bit key, 4-bit compression, 3-bit lengthening. Average Coverage = 93.06%.

2.2 Parity Checking

2.2.1 Key Shortening

The parity checking was accomplished by using the same code used in previous simulations for generating shortened keys, and simply plugging in the parity checking module in place of the signature analyser. Shortening was done for all possible combinations of the keys, as well as for consecutive deletions.

The following graphs are for 8-bit keys.

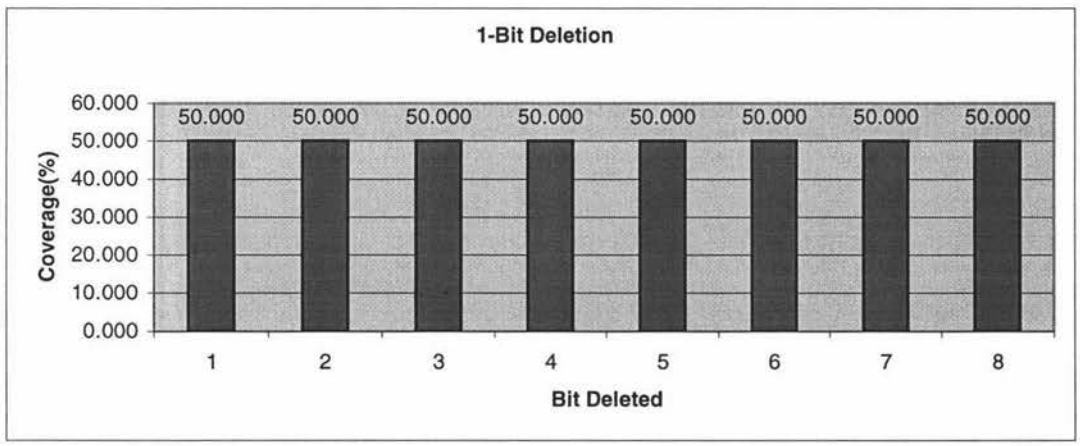


Figure A0-14: Parity checking: 8-bit key, 1-bit deletion, individual bit position coverage.

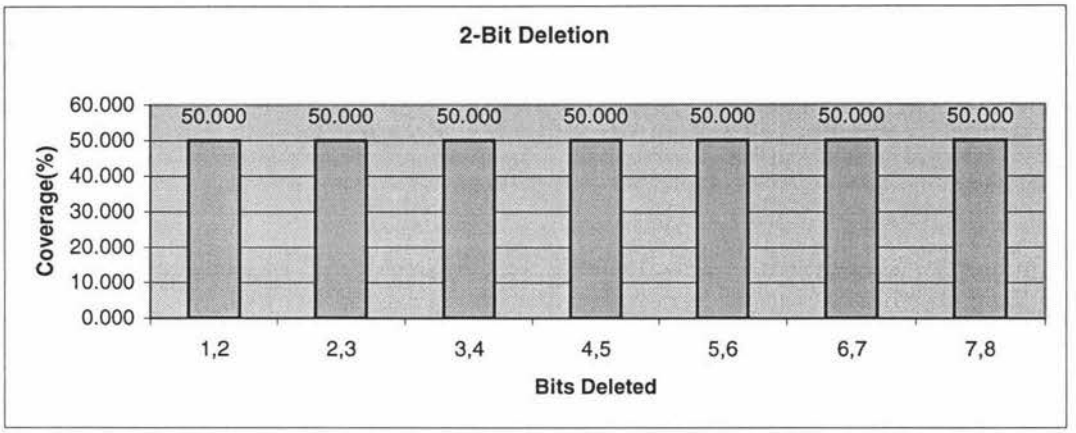


Figure A0-15: Parity checking: 8-bit key, 2-bit deletion, individual bit position coverage.

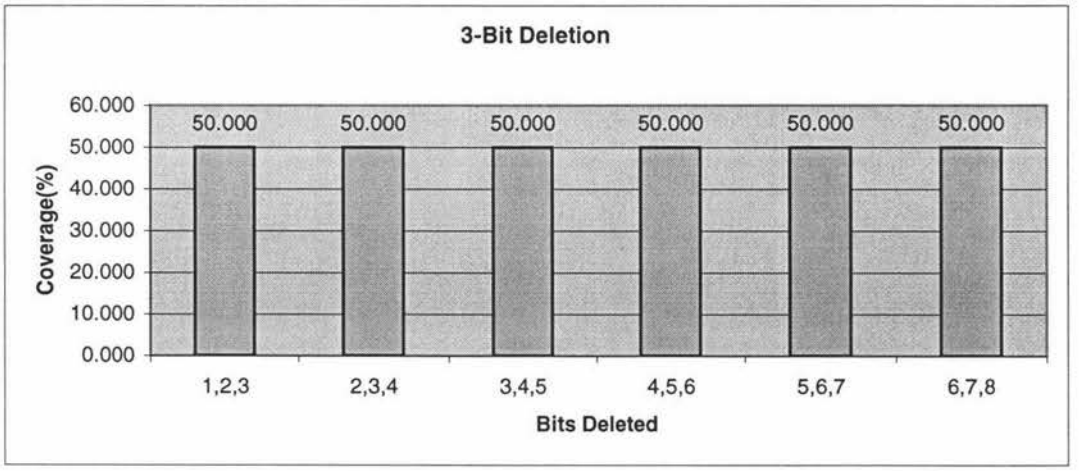


Figure A0-16: Parity checking: 8-bit key, 3-bit deletion, individual bit position coverage.

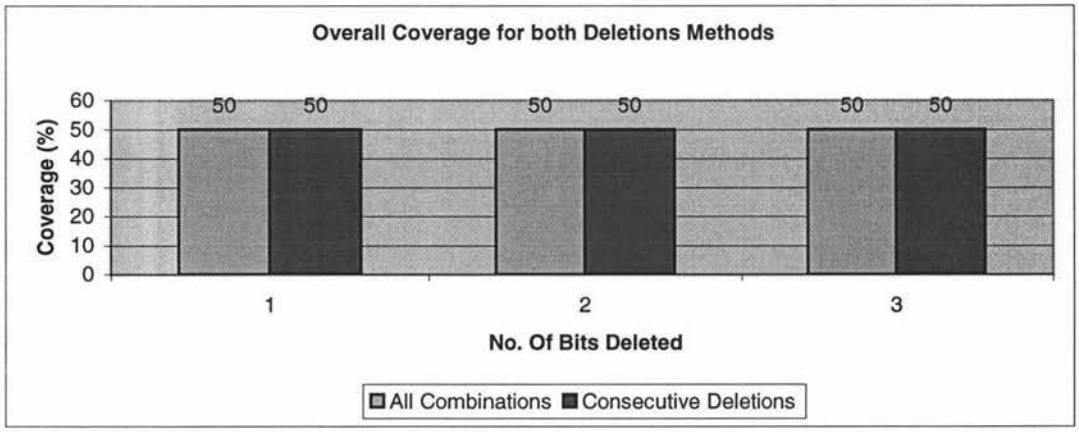


Figure A0-17: Parity checking: 8-bit key, overall coverages for both shortening methods.

The following graphs are for 12-bit key shortening.

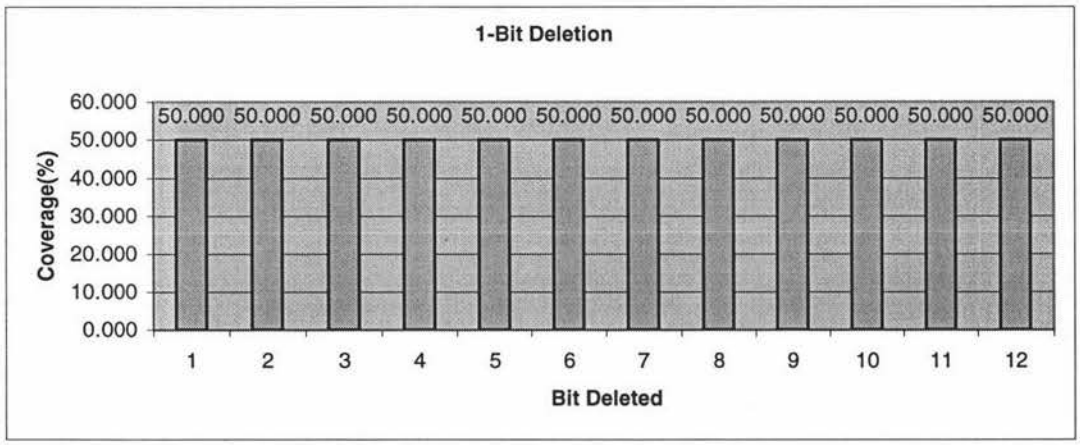


Figure A0-18: Parity checking: 12-bit key, 1-bit deletion, individual bit position coverage.

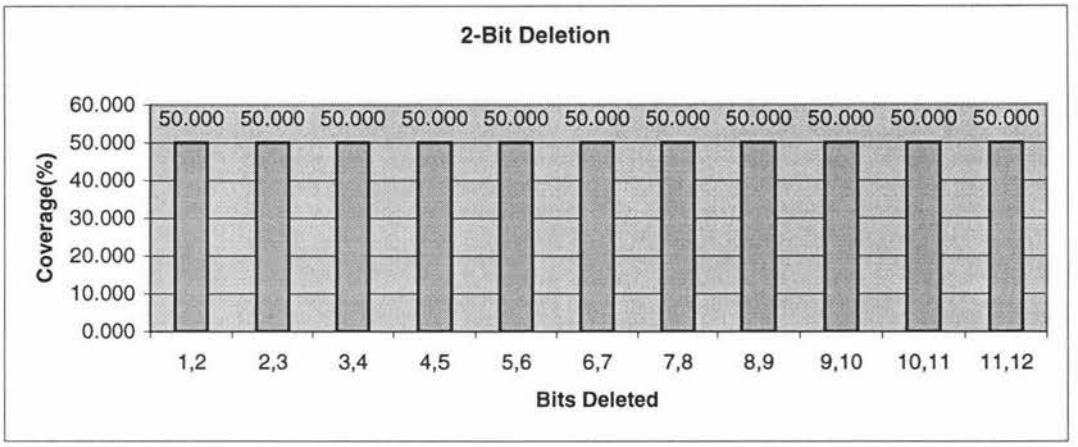


Figure A0-19: Parity checking: 12-bit key, 2-bit deletion, individual bit position coverage.

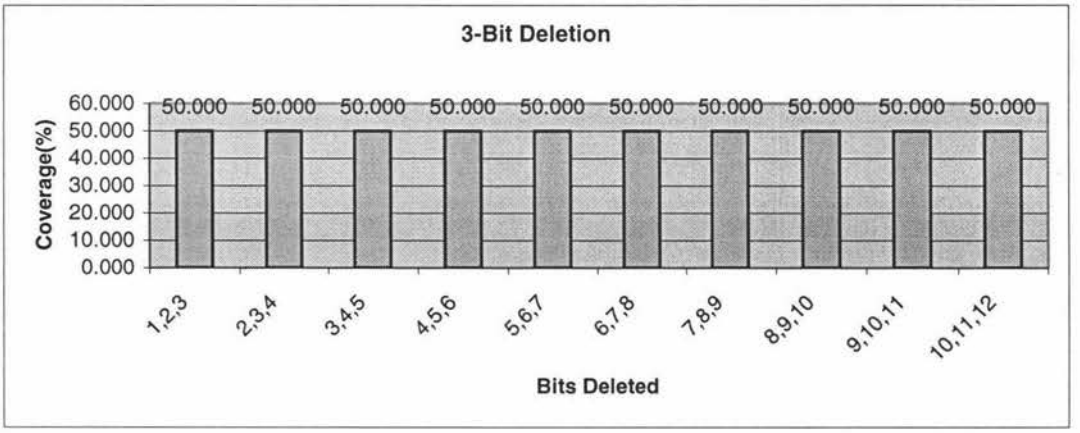


Figure A0-20: Parity checking: 12-bit key, 3-bit deletion, individual bit position coverage.

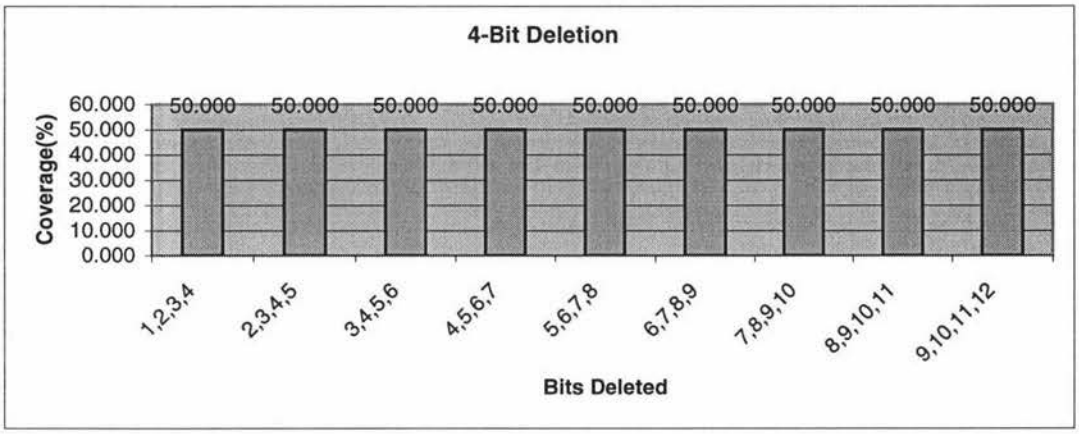


Figure A0-21: Parity checking: 12-bit key, 4-bit deletion, individual bit position coverage.

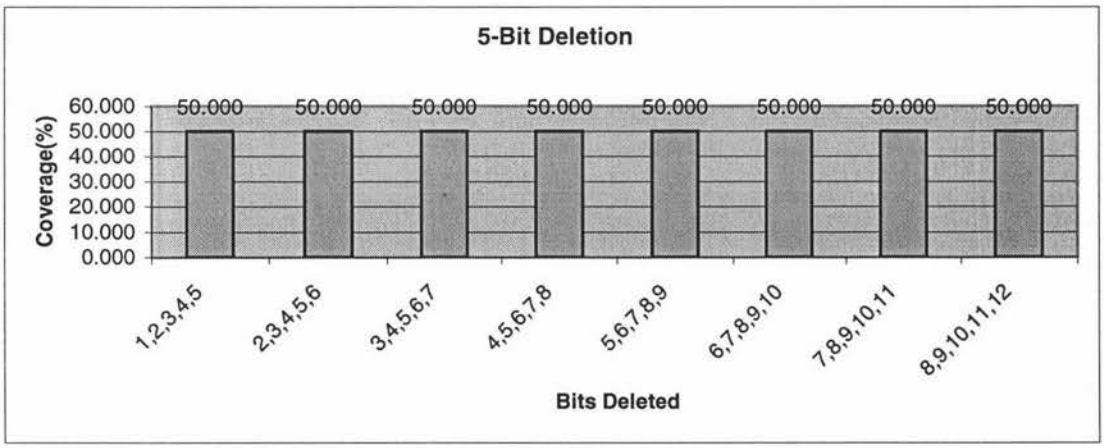


Figure A0-22: Parity checking: 12-bit key, 5-bit deletion, individual bit position coverage.

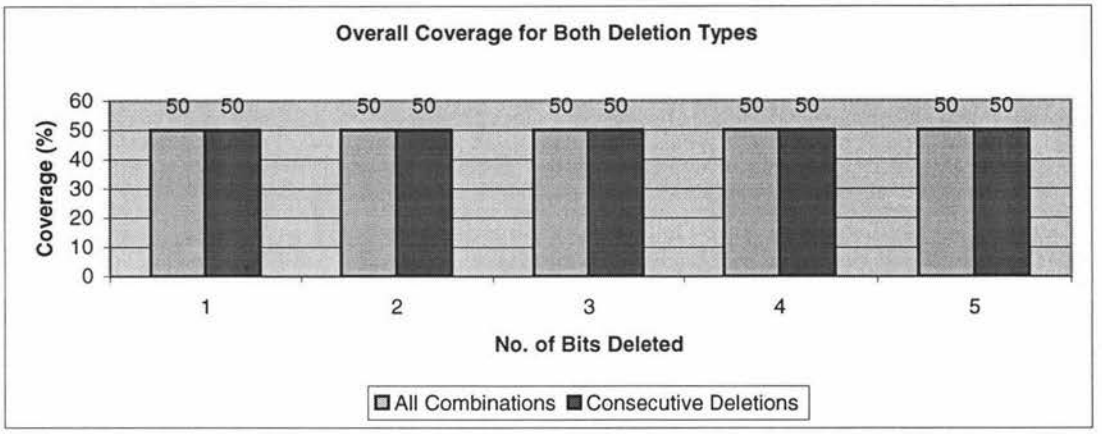


Figure A0-23: Parity checking: 12-bit key, overall coverages for both shortening methods.

2.2.2 Key Lengthening

The parity checking was accomplished by using the same code used in previous simulations for generating lengthened keys, and simply plugging in the parity checking module in place of the signature analyser. Lengthening was done for all possible combinations of the keys, as well as for consecutive insertions in blocks for 8-bit keys.

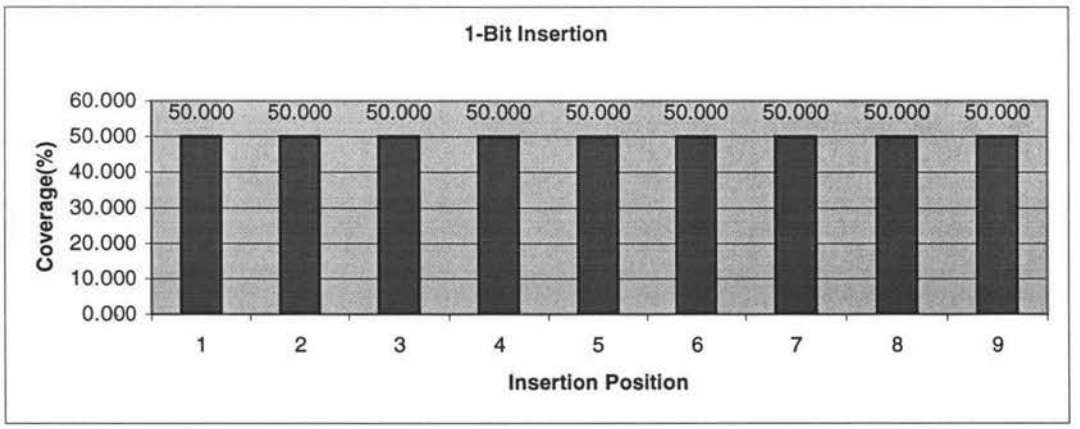


Figure A0-24: Parity checking: 8-bit key, 1-bit lengthening, individual bit position coverage.

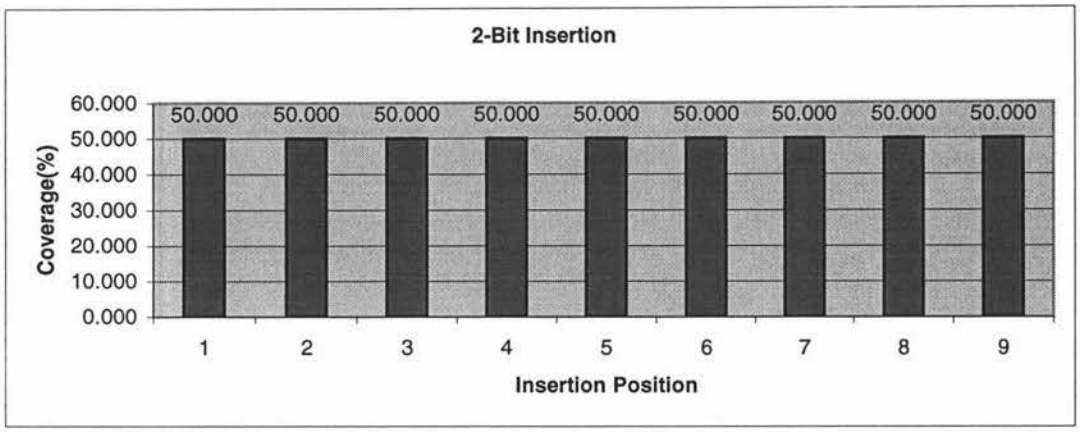


Figure A0-25: Parity checking: 8-bit key, 2-bit lengthening, individual bit position coverage.

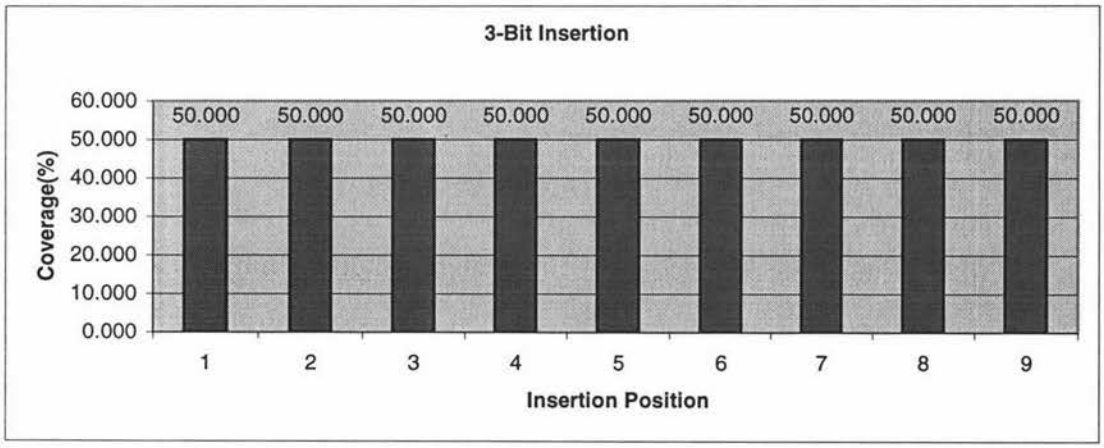


Figure A0-26: Parity checking: 8-bit key, 3-bit lengthening, individual bit position coverage.

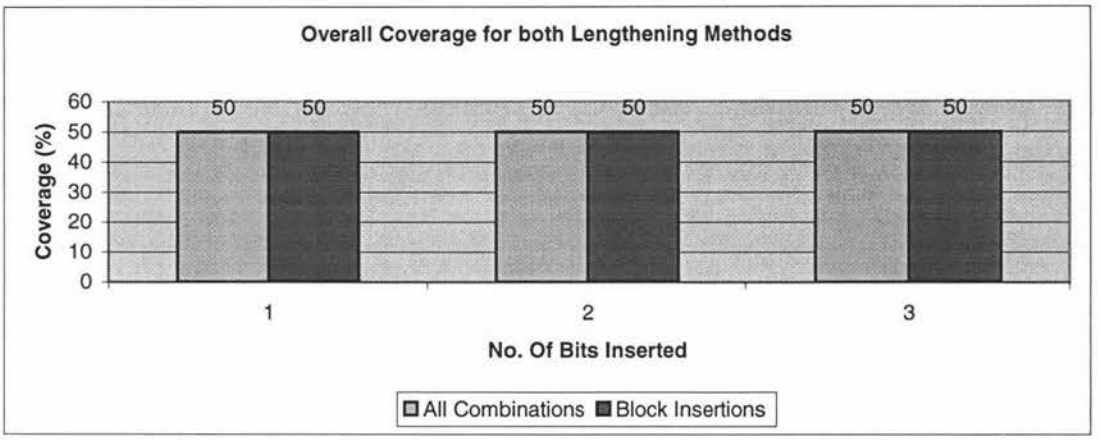


Figure A0-27: Parity checking: 8-bit key, overall coverages for both lengthening methods.

2.3 Syndrome Coding

2.3.1 Key Shortening

The syndrome coding was accomplished by using the same code used in previous simulations for generating shortened keys, and simply plugging in the syndrome coding module in place of the signature analyser. Shortening was done for all possible combinations of the keys, as well as for consecutive deletions.

The following graphs are for syndrome coding of 8-bit keys.

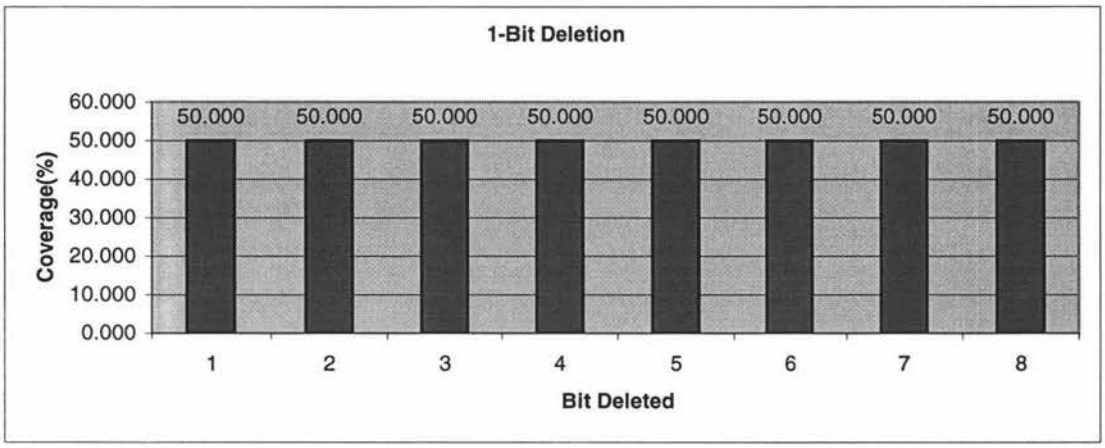


Figure A0-28: Syndrome Coding: 8-bit key, 1-bit shortening, individual bit position coverage.

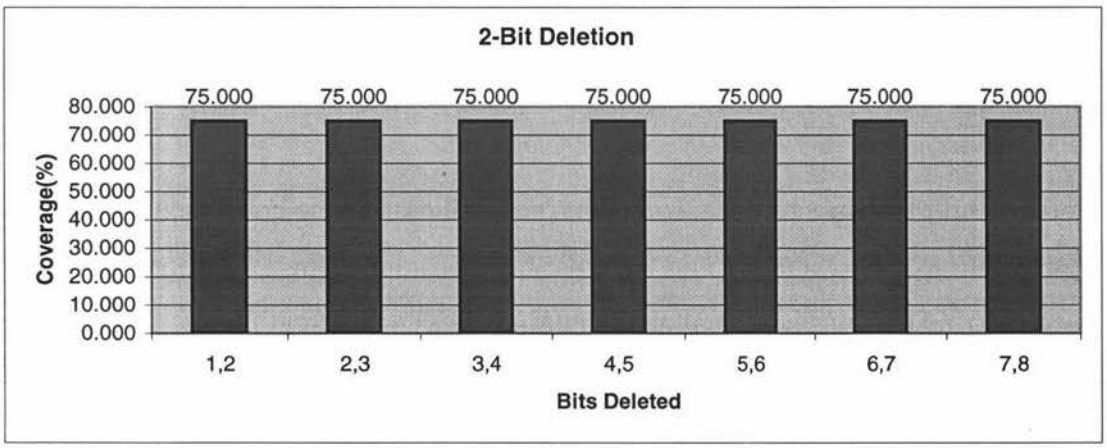


Figure A0-29: Syndrome Coding: 8-bit key, 2-bit shortening, individual bit position coverage.

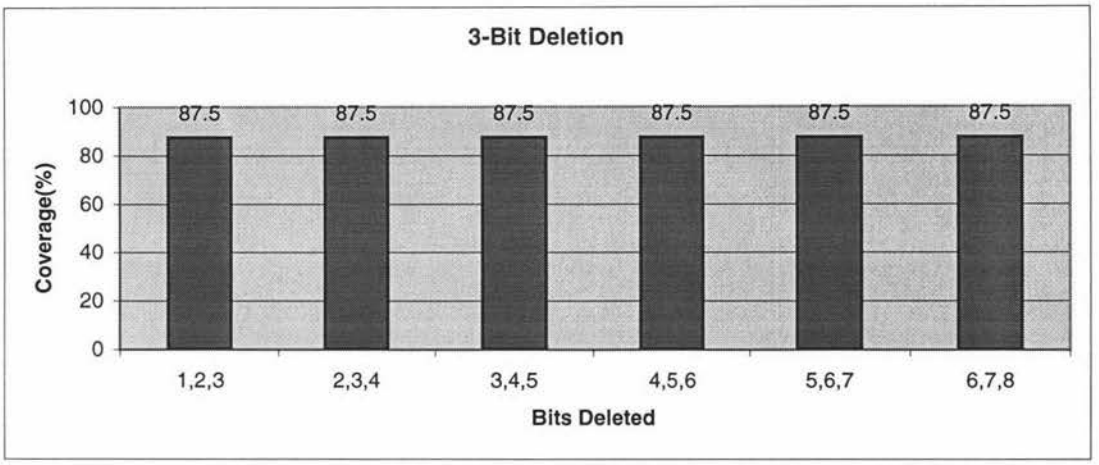


Figure A0-30: Syndrome Coding: 8-bit key, 3-bit shortening, individual bit position coverage.

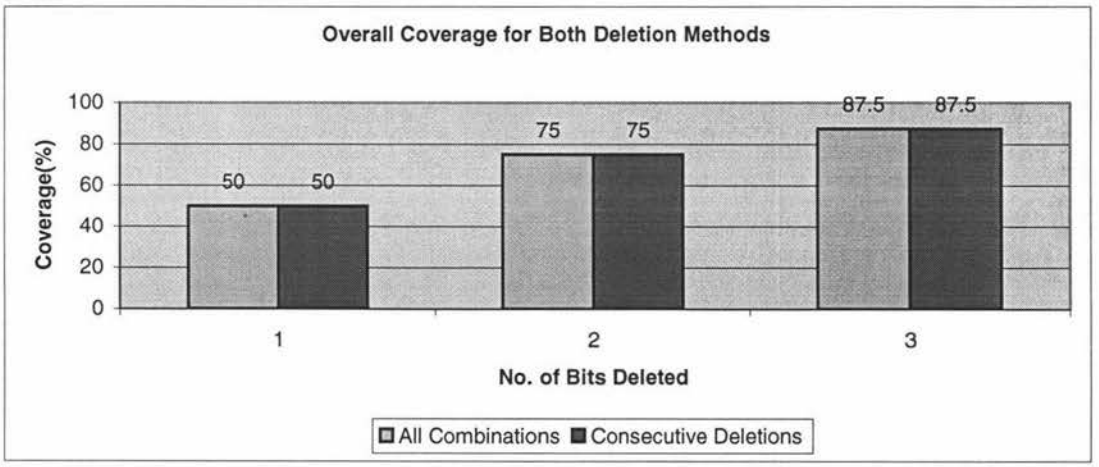


Figure A0-31: Syndrome Coding: 8-bit key, overall coverages for both shortening methods.

The following graphs are for 12-bit keys.

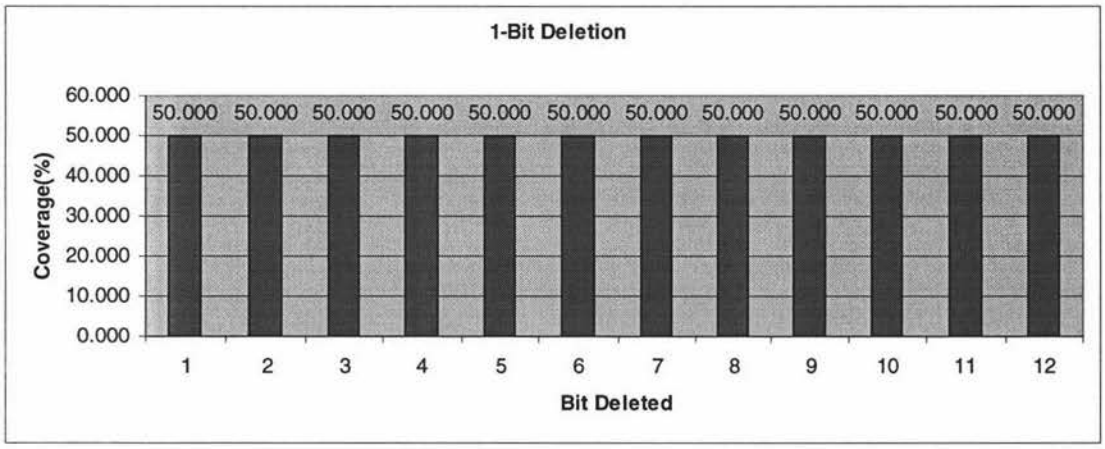


Figure A0-32: Syndrome Coding: 12-bit key, 1-bit shortening, individual bit position coverage.

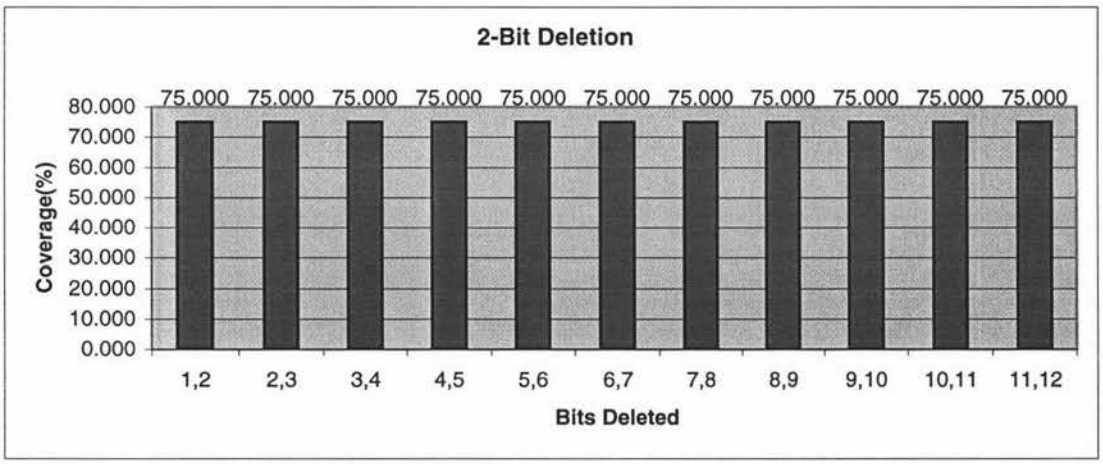


Figure A0-33: Syndrome Coding: 12-bit key, 2-bit shortening, individual bit position coverage.

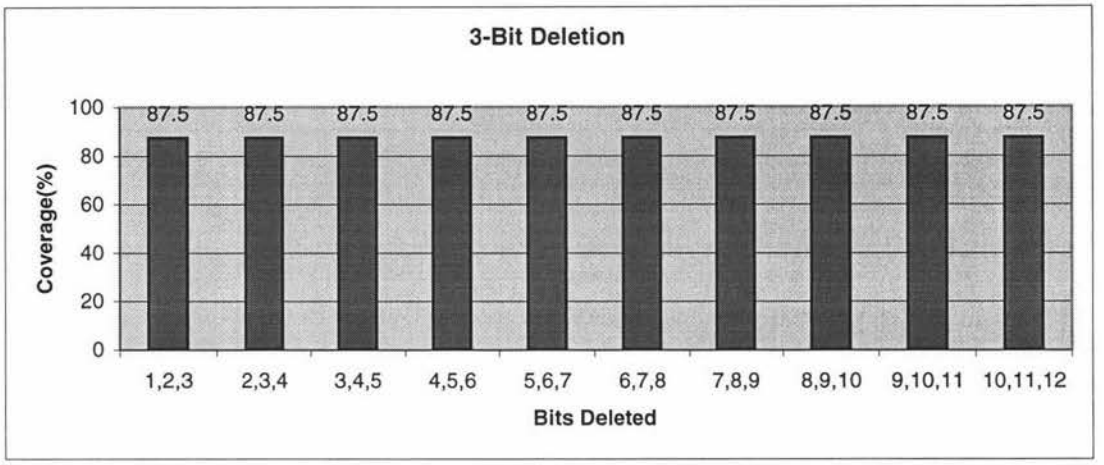


Figure A0-34: Syndrome Coding: 12-bit key, 3-bit shortening, individual bit position coverage.

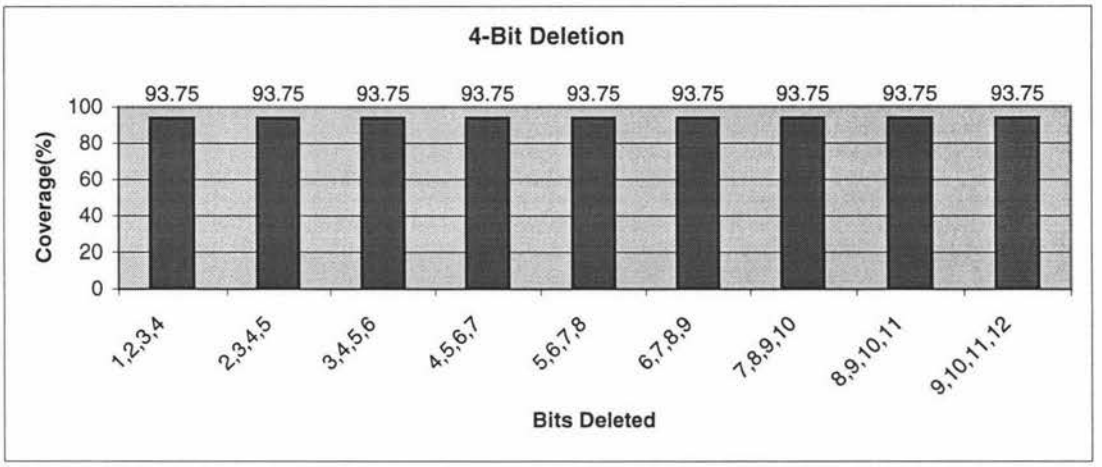


Figure A0-35: Syndrome Coding: 12-bit key, 4-bit shortening, individual bit position coverage.

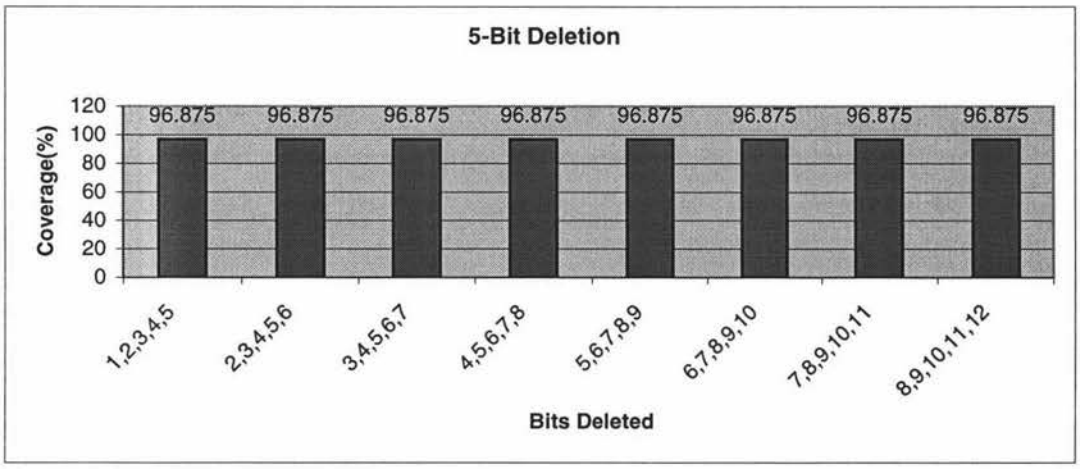


Figure A0-36: Syndrome Coding: 12-bit key, 5-bit shortening, individual bit position coverage.

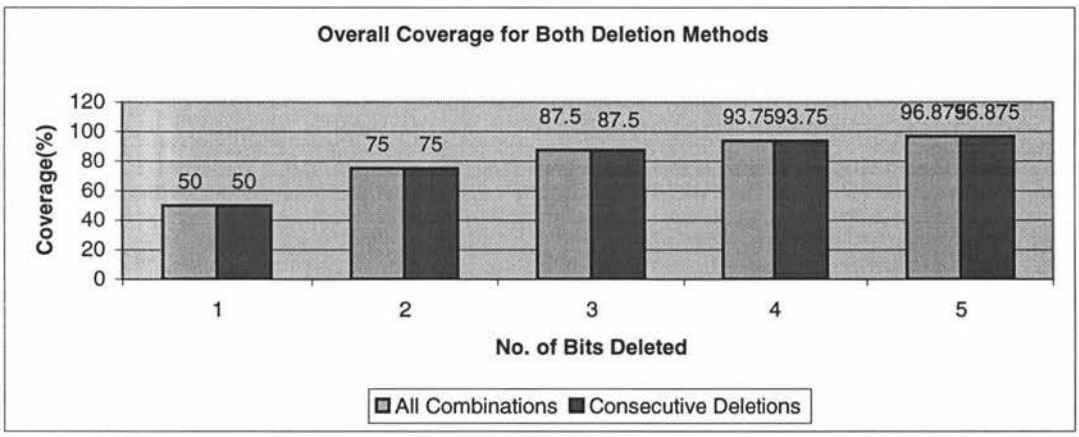


Figure A0-37: Syndrome Coding: 12-bit key, overall coverages for both shortening methods.

2.3.2 Key Lengthening

The syndrome coding was accomplished by using the same code used in previous simulations for generating lengthened keys, and simply plugging in the syndrome coding module in place of the signature analyser. Lengthening was done for all possible combinations of the keys, as well as for insertions in blocks.

The following graphs are for 8-bit key lengthening.

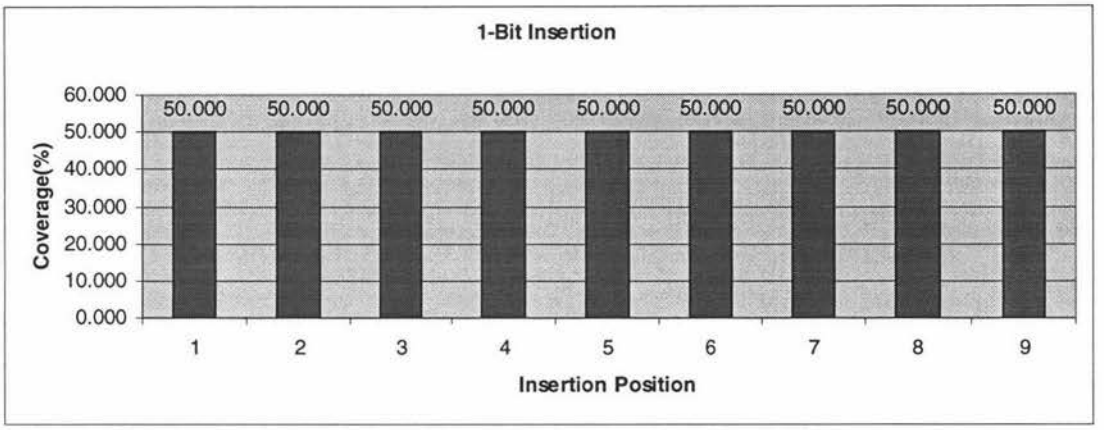


Figure A0-38: Syndrome Coding: 8-bit key, 1-bit lengthening, individual bit position coverage.

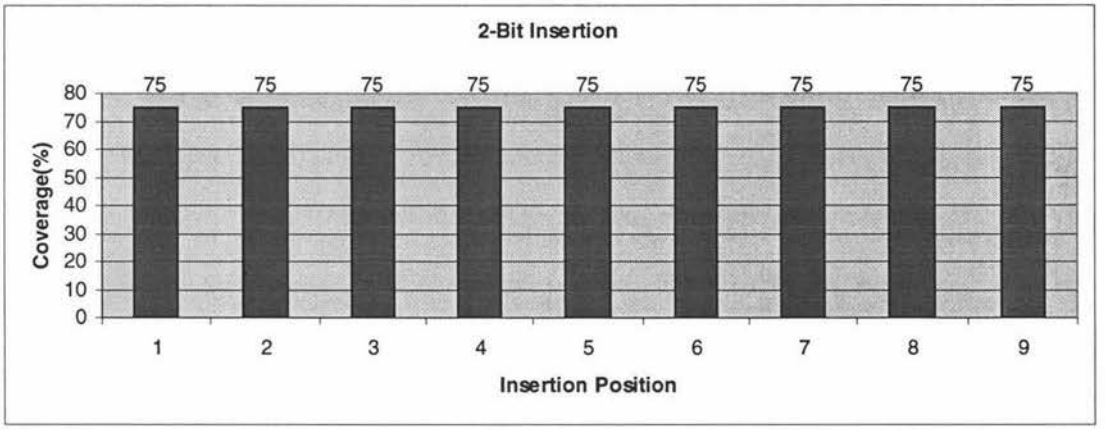


Figure A0-39: Syndrome Coding: 8-bit key, 2-bit lengthening, individual bit position coverage.

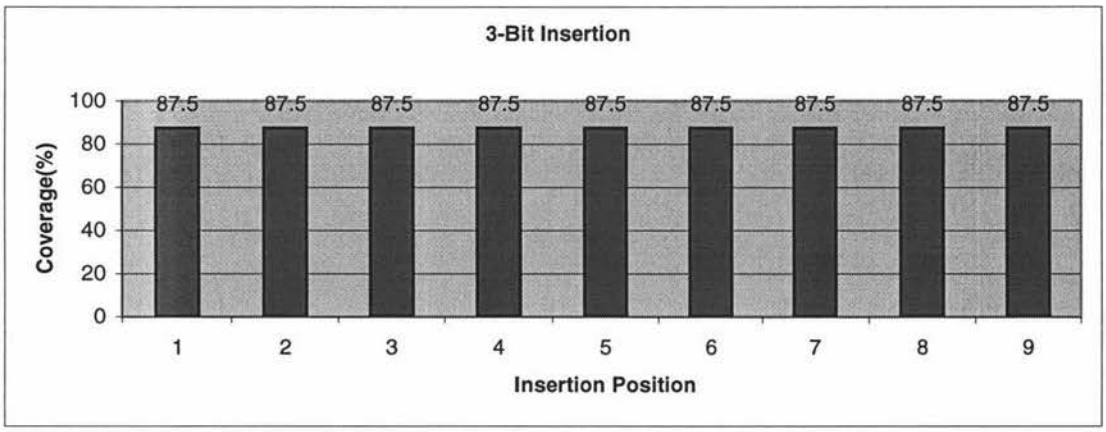


Figure A0-40: Syndrome Coding: 8-bit key, 3-bit lengthening, individual bit position coverage.

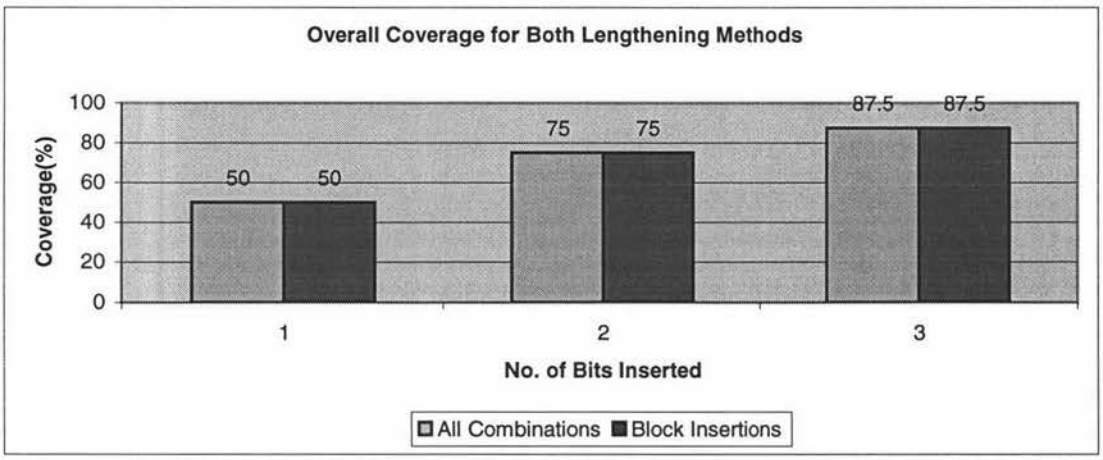


Figure A0-41: Syndrome Coding: 8-bit key, overall coverages for both lengthening methods.

2.4 Transition Counting

2.4.1 Key Shortening

The transition counting was accomplished by using the same code used in previous simulations for generating shortened keys, and simply plugging in the transition counting module in place of the signature analyser. Shortening was

done for all possible combinations of the keys, as well as for consecutive deletions. In essence, every transition incremented a counter, with the resultant number being a count of how many transitions occurred in the key.

The following graphs are for 8-bit key shortening.

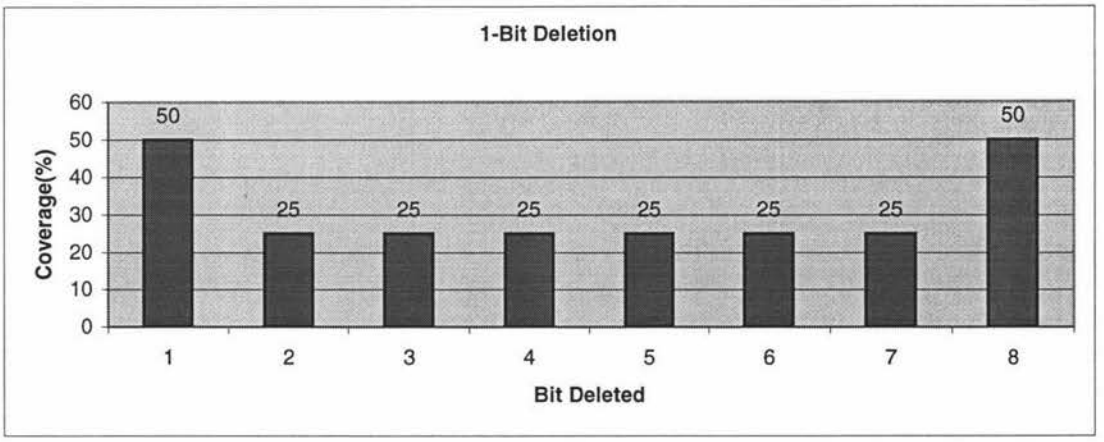


Figure A0-42: Transition Counting: 8-bit key, 1-bit shortening, individual bit position coverage.

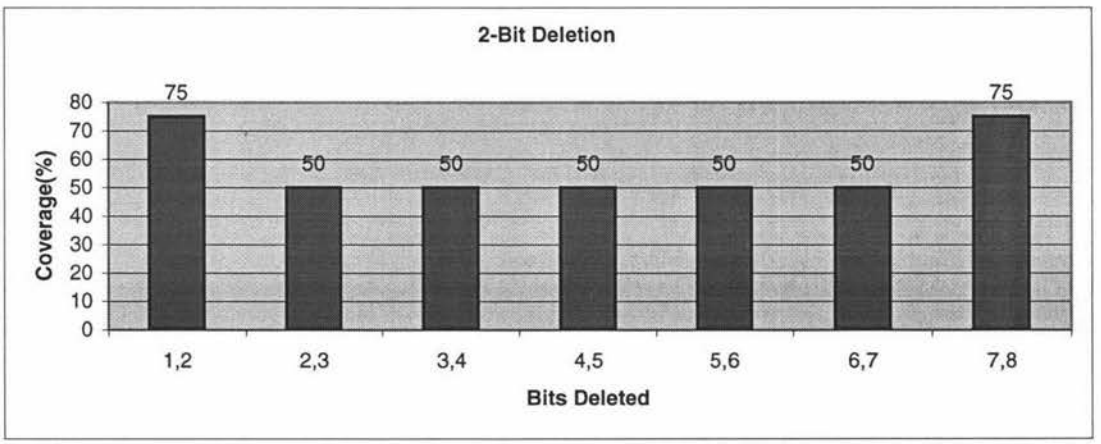


Figure A0-43: Transition Counting: 8-bit key, 2-bit shortening, individual bit position coverage.

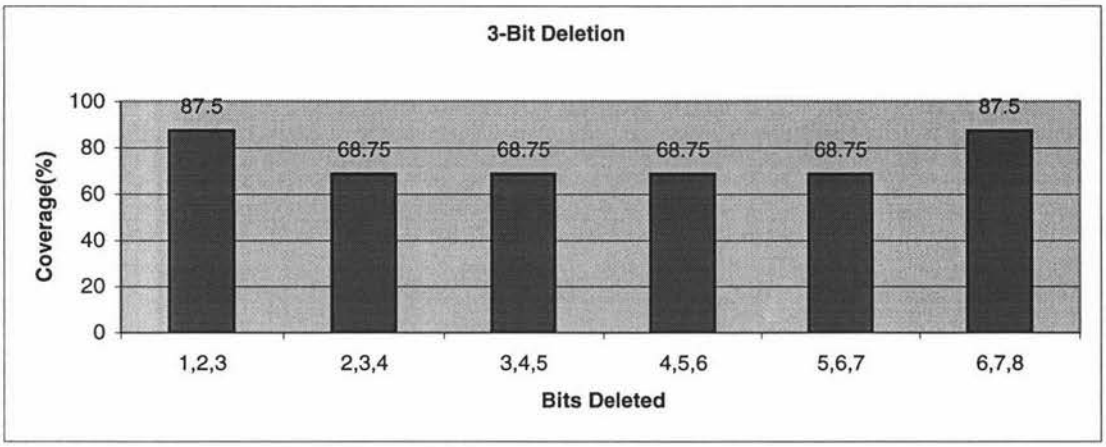


Figure A0-44: Transition Counting: 8-bit key, 3-bit shortening, individual bit position coverage.

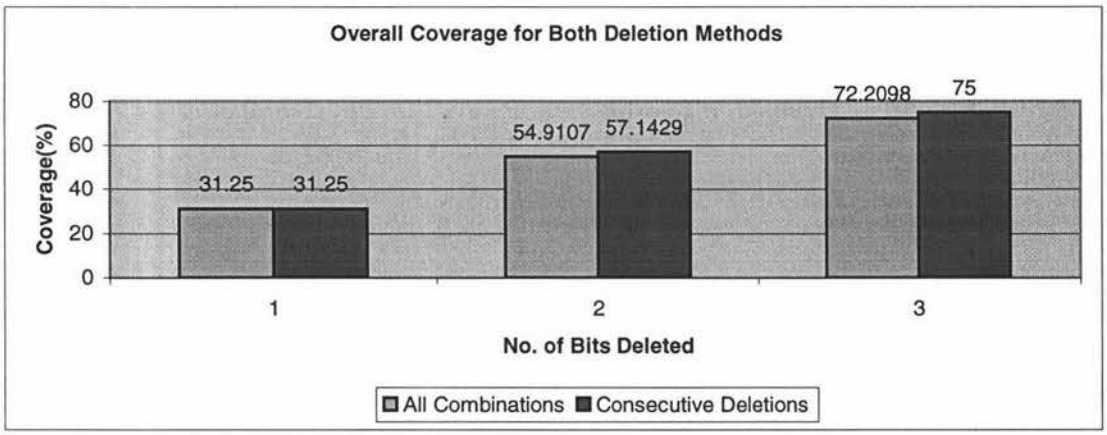


Figure A0-45: Transition Counting: 8-bit key, overall coverages for both shortening methods.

The following graphs are for 12-bit key shortening.

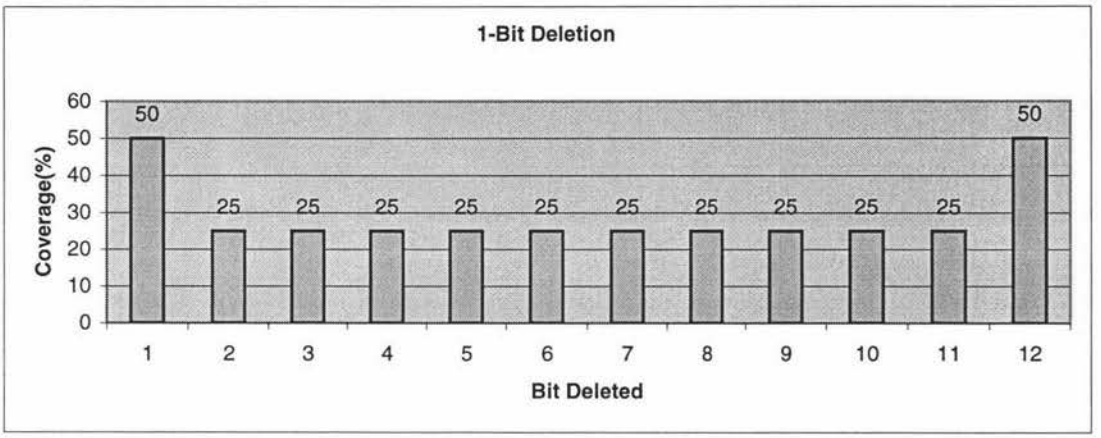


Figure A0-46: Transition Counting: 12-bit key, 1-bit shortening, individual bit position coverage.

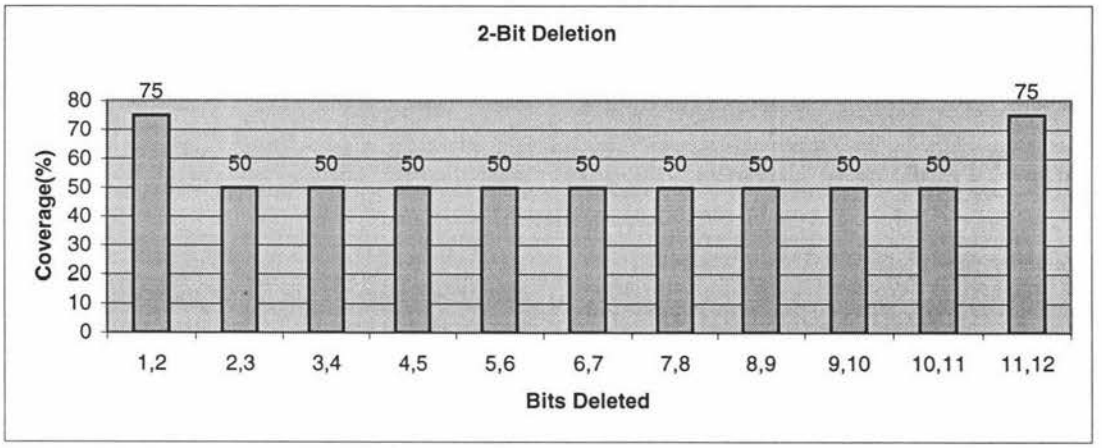


Figure A0-47: Transition Counting: 12-bit key, 2-bit shortening, individual bit position coverage.

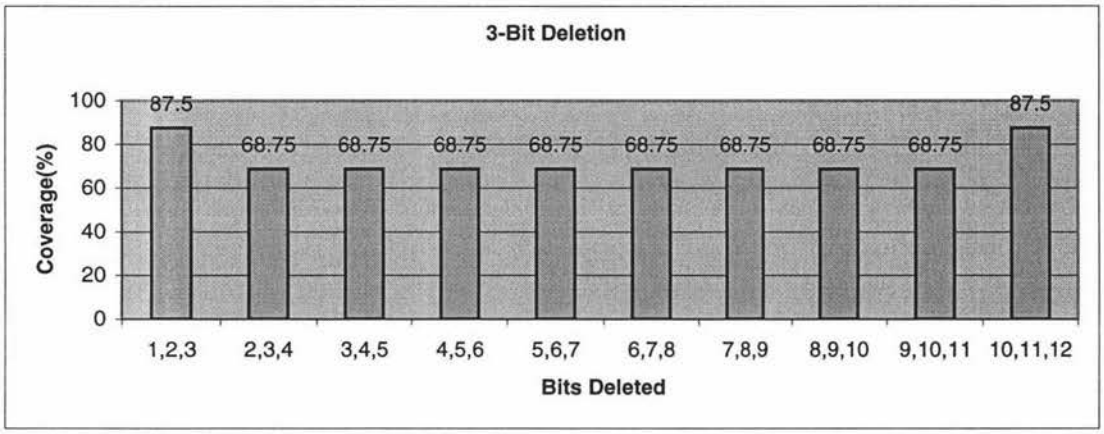


Figure A0-48: Transition Counting: 12-bit key, 3-bit shortening, individual bit position coverage.

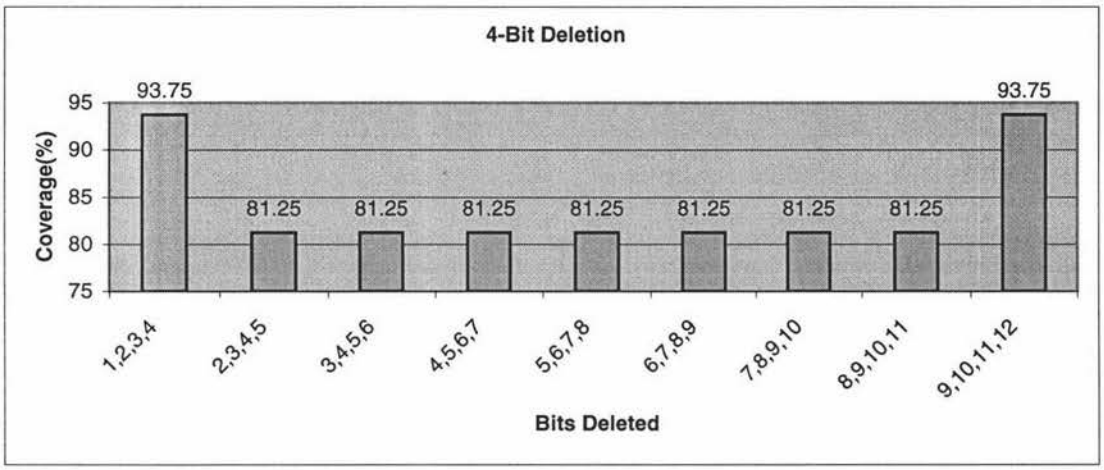


Figure A0-49: Transition Counting: 12-bit key, 4-bit shortening, individual bit position coverage.

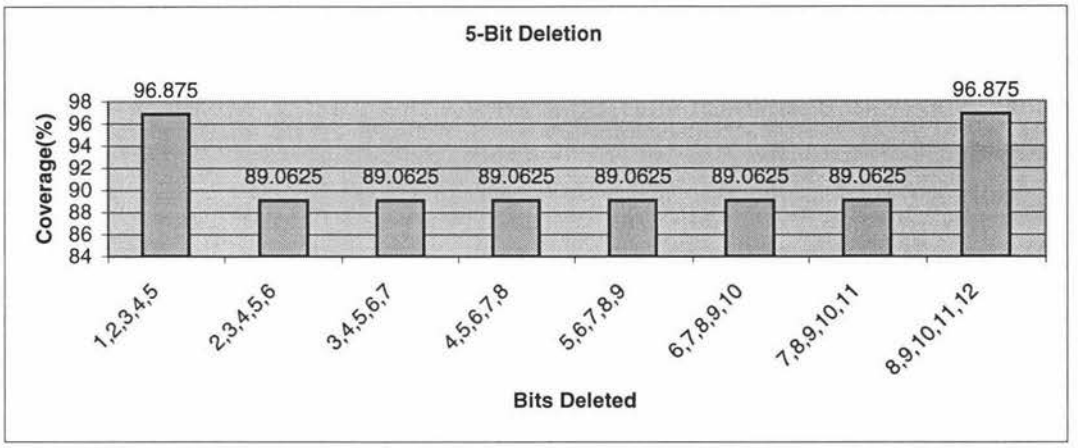


Figure A0-50: Transition Counting: 12-bit key, 5-bit shortening, individual bit position coverage.

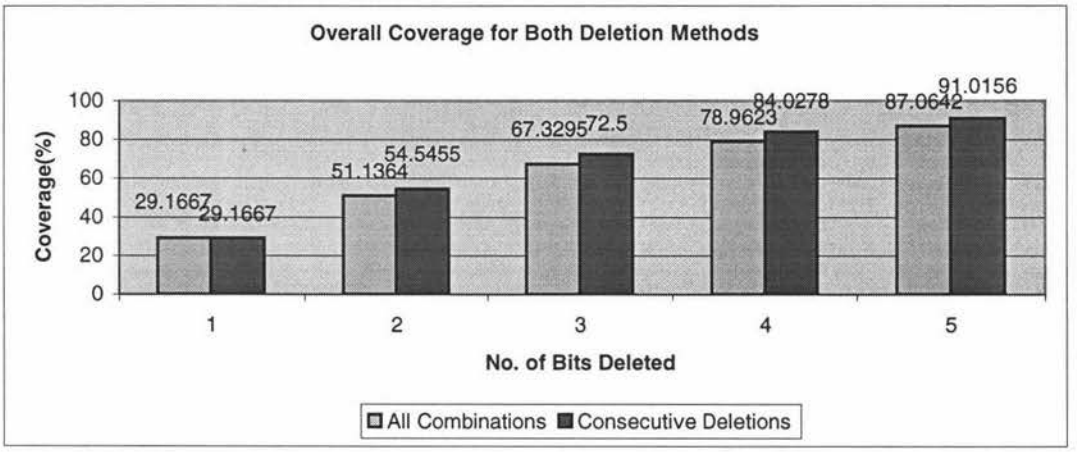


Figure A0-51: Transition Counting: 12-bit key, overall coverages for both shortening methods.

2.4.2 Key Lengthening

The transition counting was accomplished by using the same code used in previous simulations for generating lengthened keys, and simply plugging in the transition counting module in place of the signature analyser. Lengthening was done for all possible combinations of the keys, as well as for insertion in blocks. In essence, every transition incremented a counter, with the resultant number being a count of how many transitions occurred in the key.

The following graphs are for 8-bit lengthening.

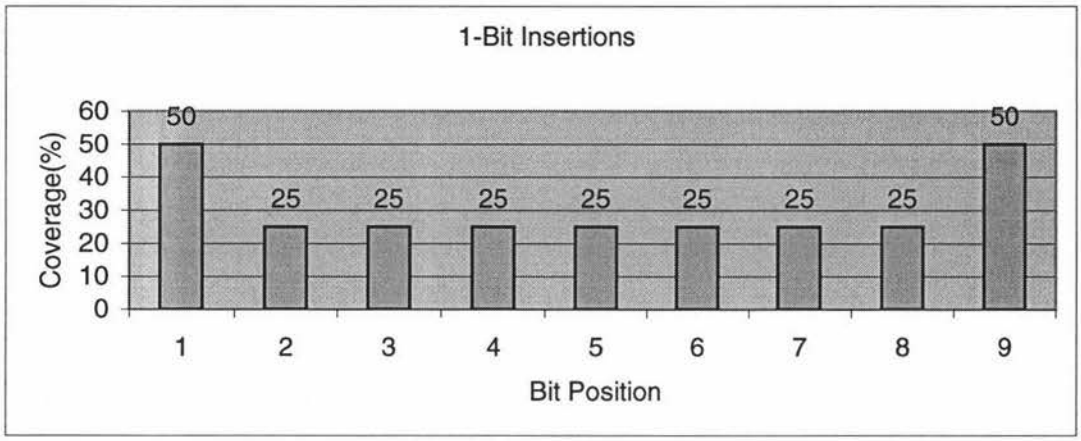


Figure A0-52: Transition Counting: 8-bit key, 1-bit lengthening, individual bit position coverage.

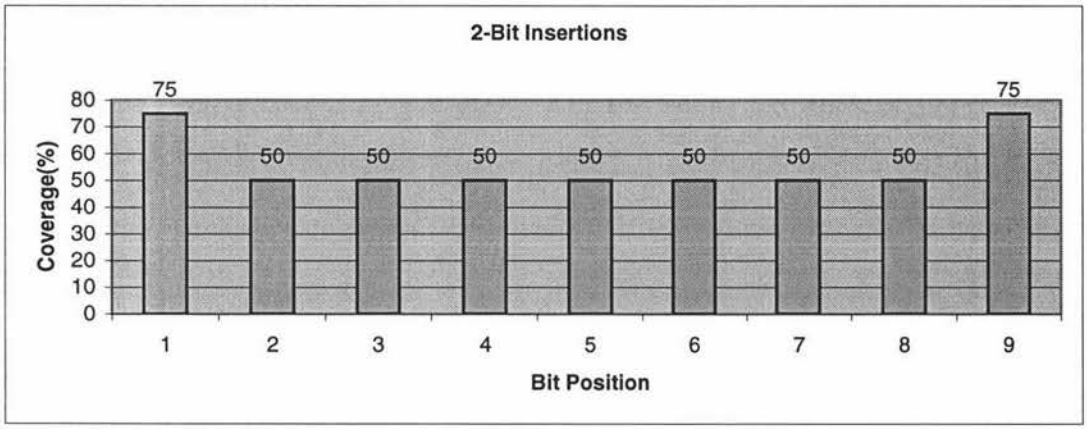


Figure A0-53: Transition Counting: 8-bit key, 2-bit lengthening, individual bit position coverage.

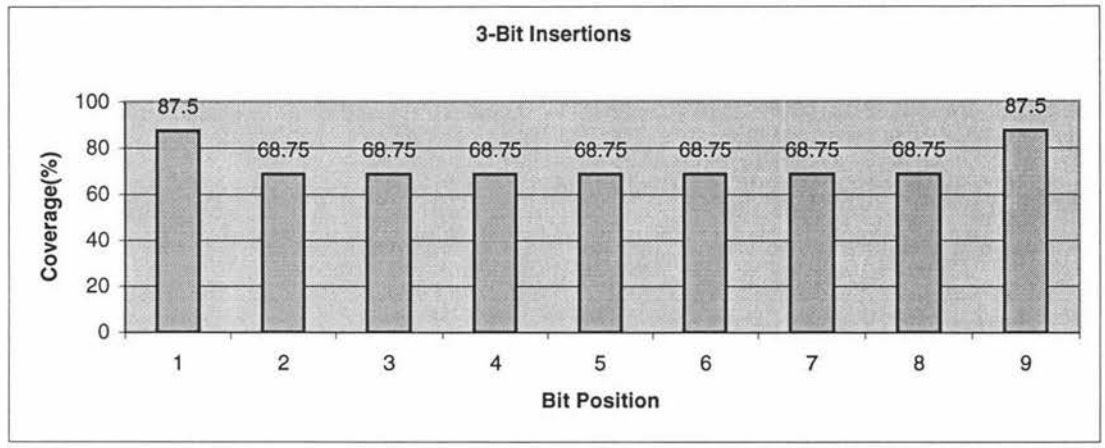


Figure A0-54: Transition Counting: 8-bit key, 3-bit lengthening, individual bit position coverage.

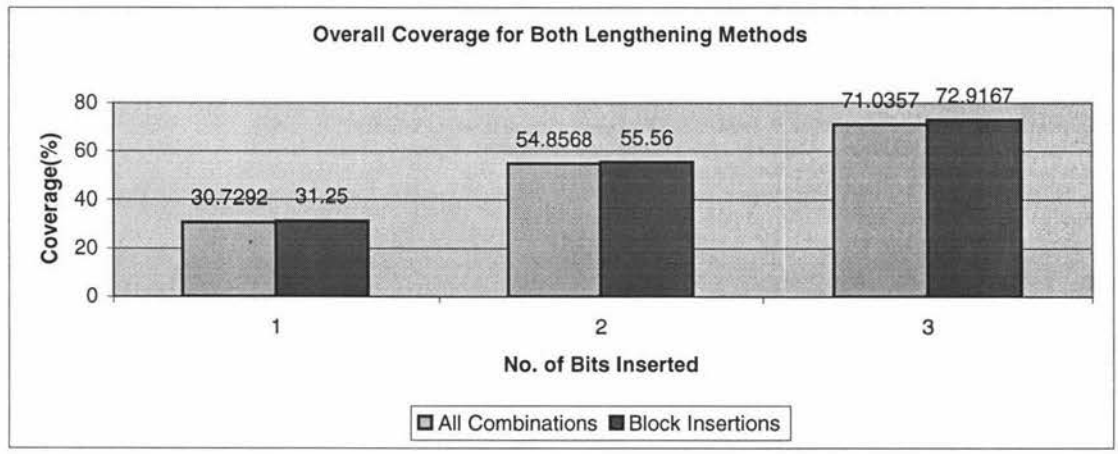


Figure A0-55: Transition Counting: 8-bit key, overall coverages for both lengthening methods.

Appendix 3

Vision System Code

This section contains the code used to implement the robot vision system. The code was integrated into the code for the whole system. Only the code for the algorithm implemented is included here, since there is a lot of code for the modules it was integrated with, and they are not relevant to this thesis.

3.1 Vision System Program

```
// Dr. Chris Messom - Institute of Information and Mathematical Sciences
// Albany Campus, Auckland, Massey University

// Karthik Subramaniam - Institute of Information Sciences and Technology
// Turitea Campus, Palmerston North, Massey University

// Main file for vision algorithm

#include <math.h>
#include <stdlib.h>
#include "stdAfx.h"


/*Constants*/

#define IMAGE_WIDTH 15
#define IMAGE_HEIGHT 5
#define NUMOBJECTS 3
#define SIZEOBJECTS 2


#define COLOUR1_REF 1
#define COLOUR2_REF 2
#define BALL_REF 4
#define TEAM_REF 8
#define OPPONENT_REF 16
#define BACKGROUND_REF 0

/*----------*/

/*Structs*/

//struct for every run of pixels
struct RLE_element{
        int iElement_ID;//element id
        int iObject_ID;//classified object
        int iStart_Pos, iEnd_Pos;
        int iColourRef;//colour reference of pixel
        int iParent_Child_Exists;//-1 if no parent, child
        int iRow;
};

struct Neighbour{
        int iParent_Count, iChild_Count; //-1 if none exists
        int iParents[20];//array holds RLE_IDs of parents
        int iChildren[20];//holds RLE_IDs of chilren
        int iCurrent_Child, iCurrent_Parent;//used in object location
                                                                    //to identify how
many parents and children
                                                                    //have been visited
in traversal
};

/*----------*/

/*Globals*/
#define ELEMENTS (IMAGE_WIDTH)*(IMAGE_HEIGHT/2)

//RLE array is 240 Rows x 80 Cols for 320x240 image
RLE_element RLE[ELEMENTS];
int RLE_index=-1;

int RowStart[IMAGE_HEIGHT];//array storing index of first RLE in row

//Array of 'Neighbour' structs to hold connectivity relationships
Neighbour Neighbours[ELEMENTS];
```

```
//hold image matrix as a global variable
BYTE iImage_Matrix[IMAGE_WIDTH*IMAGE_HEIGHT*3];

//used for the test image to determine
//which colour is to be placed
//value in YUV
BYTE Colour[3];

//Look-up table
BYTE LUT_Y[256]={0}, LUT_U[256]={0}, LUT_V[256]={0};

//YUV value ranges for each type
BYTE ballcolYmin=10, ballcolYmax=50;
BYTE ballcolUmin=60, ballcolUmax=80;
BYTE ballcolVmin=60, ballcolVmax=80;

BYTE colour1Ymin=80, colour1Ymax=100;
BYTE colour1Umin=90, colour1Umax=110;
BYTE colour1Vmin=90, colour1Vmax=100;

BYTE colour2Ymin=150, colour2Ymax=200;
BYTE colour2Umin=90, colour2Umax=140;
BYTE colour2Vmin=110, colour2Vmax=120;

BYTE teamcolYmin=120, teamcolYmax=150;
BYTE teamcolUmin=110, teamcolUmax=150;
BYTE teamcolVmin=125, teamcolVmax=150;

BYTE oppocolYmin=100, oppocolYmax=200;
BYTE oppocolUmin=155, oppocolUmax=180;
BYTE oppocolVmin=80, oppocolVmax=110;

BYTE minYVal=10;
BYTE minUVal=50;
BYTE minVVal=60;
//currently used object id number
int iCurrent_ID;

//Parameters for self-checking

int iKey_Sequence[80]={0};//checksum at the end of the program
int iKey_Counter=0;

//Values that the syndrome could be if program
//executed correctly
int iAccepted_Syndrome_Values[20];

//Parity bits have to be assigned for maximum efficiency
int iParity;
int iAccepted_Parity;

/*----------*/

/*Functions*/

//creates artificial image for testing
void create_image();
//returns a random colour in one of the colour spaces
int get_colour();
//main routine to start RLE process
void RLE_image();
//sets colour ranges for the five spaces and initialises YUV LUT's
void initialise_colours();
//places image of a specified shape into image matrix
void place_image(int ix, int iy, int iSize_of_Objects, int rand_range);
//find objects by checking connectivity
void object_location();
//perform depth-first search through all connections
void traverse_graph(int j);
```

```
//print out the image content (Y, U or V), i = 0, 1, 2
void printImage(int i);

//Output the contents of the RLE array
void printRLE(int debugLevel);

//output LUT array values (YUV), f = 0, 1, 2
void printLUT(int f);

//assign parent and children to RLE
void parentChildLink();

//get a random colour of one of the given objects
void get_colour(int o);

//self-checking data
void self_check();
/*----------*/

/*Implementation*/
#define ROUNDS 1

void main(){

        initialise_colours();//set colour ranges
        /*
        printLUT(0);
        printf("\n");
        getchar();
        printLUT(1);
        printf("\n");
        getchar();
        printLUT(2);
        getchar();
*/
        create_image();
        //get_image(); pointer to real image
        //image is 320x240
        if(IMAGE_WIDTH<=20){
                printImage(0);
                printf("\n");
                printImage(1);
                printf("\n");
                printImage(2);
                getchar();
        }
        for(int i =0;i<ROUNDS;i++)
        {
                for(int j=0;j<RLE_index;j++){
                        if(RLE[j].iParent_Child_Exists){
                                Neighbours[RLE[j].iElement_ID].iChild_Count=0;
                                Neighbours[RLE[j].iElement_ID].iParent_Count=0;
                        }
                }
        RLE_index=-1;
        //first run to RLE image
        RLE_image();

        if(ROUNDS==1){
                printRLE(2);
                getchar();
        //      printRLE(4);
        //      getchar();
        }
        //find objects by checking connectivity
        object_location();
        }
        printRLE(6);
```

```
        self_check();
}

//randomly places pixels in the image
void create_image(){
        //number of objects in image
        const int iNumber_of_Objects = NUMOBJECTS;//40
        int i,j;

        //start off with squares being placed
        const int iSize_of_Objects = SIZEOBJECTS;

        int ix, iy;

        //Set a noisy background
        for( j = 0; j<IMAGE_HEIGHT;j++)
                for(i = 0; i<IMAGE_WIDTH;i++)
                {
                        Colour[0]=(rand()%minYVal);
                        Colour[1]=(rand()%minUVal);
                        Colour[2]=(rand()%minVVal);

                        iImage_Matrix[(j*IMAGE_WIDTH + i)*3]=Colour[0];
                        iImage_Matrix[ 1+ (j*IMAGE_WIDTH +i)*3]=Colour[1];
                        iImage_Matrix[ 2+ (j*IMAGE_WIDTH +i)*3]=Colour[2];

                }

        //place objects on playing area
        for( i=1;i<=iNumber_of_Objects;i++){
                ix=(rand()%(IMAGE_WIDTH-iSize_of_Objects));
                iy=(rand()%(IMAGE_HEIGHT-iSize_of_Objects));

                int rand_range = get_colour();

                place_image(ix, iy, iSize_of_Objects,rand_range);
        }
}

//returns colour of object to place
int get_colour(){
        BYTE rand_range;
        BYTE y_min, y_max, u_min, u_max, v_min, v_max;
        //select which set of colour ranges to use
        rand_range=rand()%5;

//set range values
        switch(rand_range){
        case 0:
                        y_min=colour1Ymin;
                        y_max=colour1Ymax;
                        u_min=colour1Umin;
                        u_max=colour1Umax;
                        v_min=colour1Vmin;
                        v_max=colour1Vmax;
                        break;

        case 1:
                        y_min=colour2Ymin;
                        y_max=colour2Ymax;
                        u_min=colour2Umin;
                        u_max=colour2Umax;
                        v_min=colour2Vmin;
                        v_max=colour2Vmax;
                        break;

        case 2:
                        y_min=ballcolYmin;
                        y_max=ballcolYmax;
                        u_min=ballcolUmin;
```

```
                                u_max=ballcolUmax;
                                v_min=ballcolVmin;
                                v_max=ballcolVmax;
                                break;

        case 3:
                                y_min=teamcolYmin;
                                y_max=teamcolYmax;
                                u_min=teamcolUmin;
                                u_max=teamcolUmax;
                                v_min=teamcolVmin;
                                v_max=teamcolVmax;
                                break;

        case 4:
                                y_min=oppocolYmin;
                                y_max=oppocolYmax;
                                u_min=oppocolUmin;
                                u_max=oppocolUmax;
                                v_min=oppocolVmin;
                                v_max=oppocolVmax;
                                break;
        }

        Colour[0]=(rand()%(y_max-y_min))+y_min;
        Colour[1]=(rand()%(u_max-u_min))+u_min;
        Colour[2]=(rand()%(v_max-v_min))+v_min;
        return rand_range;
}

//get a random colour of one of the given objects
void get_colour(int o){
        BYTE rand_range;
        BYTE y_min, y_max, u_min, u_max, v_min, v_max;
        //select which set of colour ranges to use
        rand_range=o;

//set range values
        switch(rand_range){
        case 0:
                                y_min=colour1Ymin;
                                y_max=colour1Ymax;
                                u_min=colour1Umin;
                                u_max=colour1Umax;
                                v_min=colour1Vmin;
                                v_max=colour1Vmax;
                                break;

        case 1:
                                y_min=colour2Ymin;
                                y_max=colour2Ymax;
                                u_min=colour2Umin;
                                u_max=colour2Umax;
                                v_min=colour2Vmin;
                                v_max=colour2Vmax;
                                break;

        case 2:
                                y_min=ballcolYmin;
                                y_max=ballcolYmax;
                                u_min=ballcolUmin;
                                u_max=ballcolUmax;
                                v_min=ballcolVmin;
                                v_max=ballcolVmax;
                                break;

        case 3:
                                y_min=teamcolYmin;
                                y_max=teamcolYmax;
                                u_min=teamcolUmin;
```

```
                                u_max=teamcolUmax;
                                v_min=teamcolVmin;
                                v_max=teamcolVmax;
                                break;

        case 4:
                                y_min=oppocolYmin;
                                y_max=oppocolYmax;
                                u_min=oppocolUmin;
                                u_max=oppocolUmax;
                                v_min=oppocolVmin;
                                v_max=oppocolVmax;
                                break;
        }

        Colour[0]=(rand()%(y_max-y_min))+y_min;
        Colour[1]=(rand()%(u_max-u_min))+u_min;
        Colour[2]=(rand()%(v_max-v_min))+v_min;
}


//place YUV BYTE values in memory as image data
void place_image(int ix, int iy, int iSize_of_Objects, int rand_range){
        int offset = (iy*IMAGE_WIDTH+ix)*3;

        for(int i=0;i<iSize_of_Objects;i++){
                for(int j=0;j<iSize_of_Objects;j++){
                        get_colour(rand_range);
                        iImage_Matrix[(offset)+ (j*IMAGE_WIDTH + i)*3]=Colour[0];
                        iImage_Matrix[(offset+1)+ (j*IMAGE_WIDTH +i)*3]=Colour[1];
                        iImage_Matrix[(offset+2)+ (j*IMAGE_WIDTH +i)*3]=Colour[2];
                }
        }
}

/*----------*/ //end of image creation section


/*Main vision algorithm*/

//start RLE process
void RLE_image(){
        int row_counter, col_counter;
        int pixel_in_image;
        iCurrent_ID = 1;
        BYTE * pImage=iImage_Matrix;
        BYTE pY,pU,pV;
        //main loop ends at last row of image matrix
        for(row_counter=0;row_counter<IMAGE_HEIGHT;row_counter++){

                RowStart[row_counter]=RLE_index + 1; //since it will be incremented later
                for(col_counter=0;col_counter<IMAGE_WIDTH;col_counter++){

                        pixel_in_image =
                        LUT_Y[iImage_Matrix[((row_counter*IMAGE_WIDTH)+col_counter)*3]]&

        LUT_U[iImage_Matrix[((row_counter*IMAGE_WIDTH)+col_counter)*3+sizeof(BYTE)]] &

        LUT_V[iImage_Matrix[((row_counter*IMAGE_WIDTH)+col_counter)*3+2*sizeof(BYTE)]];

                        RLE_index++;

                        RLE[RLE_index].iStart_Pos=col_counter;
                        RLE[RLE_index].iEnd_Pos=col_counter;
                        RLE[RLE_index].iColourRef=pixel_in_image;
                        if(pixel_in_image!=BACKGROUND_REF){
                                RLE[RLE_index].iElement_ID=iCurrent_ID;
                                iKey_Sequence[iKey_Counter++]={1 0 0};
                        } else {
                                RLE[RLE_index].iElement_ID=0;
```

```
        iKey_Sequence[iKey_Counter++]={1 0 1};
                                }
                                RLE[RLE_index].iRow=row_counter;
                                col_counter++;

                                //continue while same pixel class
                                pixel_in_image =
                                LUT_Y[iImage_Matrix[((row_counter*IMAGE_WIDTH)+col_counter)*3]]&

        LUT_U[iImage_Matrix[((row_counter*IMAGE_WIDTH)+col_counter)*3+sizeof(BYTE)]] &

        LUT_V[iImage_Matrix[((row_counter*IMAGE_WIDTH)+col_counter)*3+2*sizeof(BYTE)]];
                                while((pixel_in_image==RLE[RLE_index].iColourRef)&&
(col_counter<IMAGE_WIDTH)){

        RLE[RLE_index].iEnd_Pos=col_counter;//(row_counter*IMAGE_WIDTH)+
                                        col_counter++;
                                        pixel_in_image =

        LUT_Y[iImage_Matrix[((row_counter*IMAGE_WIDTH)+col_counter)*3]]&

        LUT_U[iImage_Matrix[((row_counter*IMAGE_WIDTH)+col_counter)*3+sizeof(BYTE)]] &

        LUT_V[iImage_Matrix[((row_counter*IMAGE_WIDTH)+col_counter)*3+2*sizeof(BYTE)]];
                                        iKey_Sequence[iKey_Counter++]={1 1 1};
                                }

                                col_counter--;//since it will be incremented by the for loop
                                pImage-=3; //move the pointer back as well
                                //iCurrent_ID++;

                                if(RLE[RLE_index].iColourRef != BACKGROUND_REF) {
                                        iCurrent_ID++;
                                        parentChildLink();
                                }

                                /*----------end of Child-Parent Portion----------*/

                        }//end col_counter while loop

        }//end row_counter while loop
}

//assign parent and children to RLE
void parentChildLink(){

                                /*----------end of RLE portion----------*/

                                //check for parent-child relationship by:
                                //checking pixel directly above
                                //pixel to left should already have been RLEed
                                //SKIP the lines where iColourRef == BACKGROUND_REF (i.e. 0)

        if(RLE[RLE_index].iRow!=0){//only link for rows that are not the first row
                                for(int i=RowStart[RLE[RLE_index].iRow-1];i<RLE_index;i++){
                                        if(RLE[RLE_index].iColourRef == RLE[i].iColourRef){//Check
for same colour

        if(RLE[i].iEnd_Pos>=RLE[RLE_index].iStart_Pos&&RLE[i].iStart_Pos<=RLE[RLE_index].iEnd_Pos){

        RLE[RLE_index].iParent_Child_Exists=1;
                                                                RLE[i].iParent_Child_Exists=1;
                                                                //update this object's parent array
                                                                int
NeighbourID=RLE[RLE_index].iElement_ID;
                                                                int N_P_ID=RLE[i].iElement_ID;

        Neighbours[NeighbourID].iParents[Neighbours[NeighbourID].iParent_Count++] = i;

                                                                //update parent's child array
```

```
		Neighbours[N_P_ID].iChildren[Neighbours[N_P_ID].iChild_Count++] = RLE_index;

                                                        }
                                                }
                                        iKey_Sequence[iKey_Counter++]={1 1 0};
                                                //}
                                        }//end parent-child for loop
		}

                                        /*----------end of Child-Parent Portion----------*/

}

//check classified pixel connections and form objects
void object_location(){
		iCurrent_ID=1;

		//loop to start graph traversal using each RLEed element as root
		for(int j=0;j<=RLE_index;j++){
			if(RLE[j].iColourRef!=BACKGROUND_REF
				&&RLE[j].iObject_ID==0) {
				traverse_graph(j);iCurrent_ID++;
				iKey_Sequence[iKey_Counter++]={1 0 1};
			}

		}
}

//perform depth-first traversal over all vertices
void traverse_graph(int j){
		int iCurrent_Vertex=j, i;

		//stop and RETURN at this point if current vertex has (no parents or children) OR
		//(all parents and children have already been visited)
		if(RLE[j].iParent_Child_Exists==0//ParentChild doesn't exist has value 0
			||RLE[j].iObject_ID!=0//Check to see if node already visited
			){
			RLE[j].iObject_ID=iCurrent_ID;
			iKey_Sequence[iKey_Counter++]={0 0 1};
			return;
		}
		else{
			//go to next parent if parent list not exhausted
			//go to next child if child list not exhausted
			//call traverse_graph() and send as parameter the vertex number of the
			//next vertex
			RLE[j].iObject_ID=iCurrent_ID;
			for(i=0; i<Neighbours[RLE[j].iElement_ID].iParent_Count; i++)
			{
				traverse_graph(Neighbours[RLE[j].iElement_ID].iParents[i]);
			}
			for(i=0; i<Neighbours[RLE[j].iElement_ID].iChild_Count; i++)
			{
				traverse_graph(Neighbours[RLE[j].iElement_ID].iChildren[i]);
			}


		}
}

//sets colour ranges for the five spaces and initialises YUV LUT's
void initialise_colours(){
		//Find Range Values and add flags to LUT's
		int i;
		for(i =0; i<=255;i++)
		{

			if(i>=ballcolYmin&&i<=ballcolYmax){
				LUT_Y[i]|= BALL_REF;
```

```
                        iKey_Sequence[iKey_Counter++]={1 1 1};
                }
                if(i>=ballcolUmin&&i<=ballcolUmax){
                        LUT_U[i]|= BALL_REF;
        iKey_Sequence[iKey_Counter++]={1 0 1};
                }
                if(i>=ballcolVmin&&i<=ballcolVmax){
                        LUT_V[i]|= BALL_REF;
        iKey_Sequence[iKey_Counter++]={1 1 0};
                }

                if(i>=colour1Ymin&&i<=colour1Ymax){
                        LUT_Y[i]|= COLOUR1_REF;
                        iKey_Sequence[iKey_Counter++]={0 0 0};
                }
                if(i>=colour1Umin&&i<=colour1Umax){
                        LUT_U[i]|= COLOUR1_REF;
        iKey_Sequence[iKey_Counter++]={1 1 0};
                }
                if(i>=colour1Vmin&&i<=colour1Vmax){
                        LUT_V[i]|= COLOUR1_REF;
        iKey_Sequence[iKey_Counter++]={1 0 1};
                }

                if(i>=colour2Ymin&&i<=colour2Ymax){
                        LUT_Y[i]|= COLOUR2_REF;
        iKey_Sequence[iKey_Counter++]={1 1 0};
                }
                if(i>=colour2Umin&&i<=colour2Umax){
                        LUT_U[i]|= COLOUR2_REF;
                        iKey_Sequence[iKey_Counter++]={0 1 0};
                }
                if(i>=colour2Vmin&&i<=colour2Vmax){
                        LUT_V[i]|= COLOUR2_REF;
        iKey_Sequence[iKey_Counter++]={1 1 1};
                }


                if(i>=teamcolYmin&&i<=teamcolYmax){
                        LUT_Y[i]|= TEAM_REF;
        iKey_Sequence[iKey_Counter++]={1 1 0};
                }
                if(i>=teamcolUmin&&i<=teamcolUmax){
                        LUT_U[i]|= TEAM_REF;
        iKey_Sequence[iKey_Counter++]={1 0 1};
                }
                if(i>=teamcolVmin&&i<=teamcolVmax){
                        LUT_V[i]|= TEAM_REF;
        iKey_Sequence[iKey_Counter++]={0 0 0};
                }

                if(i>=oppocolYmin&&i<=oppocolYmax){
                        LUT_Y[i]|= OPPONENT_REF;
                        iKey_Sequence[iKey_Counter++]={1 1 1};
                }
                if(i>=oppocolUmin&&i<=oppocolUmax){
                        LUT_U[i]|= OPPONENT_REF;
        iKey_Sequence[iKey_Counter++]={1 0 1};
                }
                if(i>=oppocolVmin&&i<=oppocolVmax){
                        LUT_V[i]|= OPPONENT_REF;
        iKey_Sequence[iKey_Counter++]={0 0 0};
                }
        }

        return;
}
//output LUT array values (YUV), f = 0, 1, 2
void printLUT(int f){
        int i;
```

```
		switch(f){
		case 0:
			for(i=0;i<=255;i++)
				printf("%3d ", LUT_Y[i]);
			break;
		case 1:
			for(i=0;i<=255;i++)
				printf("%3d ", LUT_U[i]);
			break;
		case 2:
			for(i=0;i<=255;i++)
				printf("%3d ", LUT_V[i]);
			break;


		}
		return;
}

//print out the image content (Y, U or V), i = 0, 1, 2
void printImage(int f){
int i, j;
		for(j = 0; j <  IMAGE_HEIGHT; j++){
			for( i = 0; i < IMAGE_WIDTH; i++){
				printf("%3d ", iImage_Matrix[(IMAGE_WIDTH*j + i)*3 + f]);
			}
			printf("\n");
		}

 return;
}

//Output the contents of the RLE array
void printRLE(int debugLevel){
		int i,j;
		switch(debugLevel){
		case 3:
			for(i = 0; i<=RLE_index; i++){
				printf ("CREF %d,S %d, E %d, R %d, EID %d, PC: %d OID %d\n",
RLE[i].iColourRef, RLE[i].iStart_Pos, RLE[i].iEnd_Pos, RLE[i].iRow, RLE[i].iElement_ID,
RLE[i].iParent_Child_Exists,RLE[i].iObject_ID);
			}
			break;
		case 6:
			for(i = 0; i<=RLE_index; i++){
			if(RLE[i].iElement_ID!=0)
				printf ("CREF %d,S %d, E %d, R %d, EID %d, PC: %d OID %d\n",
RLE[i].iColourRef, RLE[i].iStart_Pos, RLE[i].iEnd_Pos, RLE[i].iRow, RLE[i].iElement_ID,
RLE[i].iParent_Child_Exists,RLE[i].iObject_ID);
			}
			break;

		case 4:
			for(i = 0; i<=RLE_index; i++){
				printf ("CREF %d,S %d, E %d, R %d, EID %d, PC: %d     ", RLE[i].iColourRef,
RLE[i].iStart_Pos, RLE[i].iEnd_Pos, RLE[i].iRow, RLE[i].iElement_ID, RLE[i].iParent_Child_Exists);
				//printf("\n");
				if(RLE[i].iParent_Child_Exists!=0){
					int N_ID=RLE[i].iElement_ID;
					printf("Pc %d Pi: ",Neighbours[N_ID].iParent_Count);
					for(j=0;j<Neighbours[N_ID].iParent_Count;j++){
						printf("%d ", Neighbours[N_ID].iParents[j]);
					}
					//printf("\n");
					printf("Cc %d Ci: ",Neighbours[N_ID].iChild_Count);
					for(j=0;j<Neighbours[N_ID].iChild_Count;j++){
						printf("%d ", Neighbours[N_ID].iChildren[j]);
					}
				}
				printf("\n");
```

```
                }

                break;
        case 5:
                for(i = 0; i<=RLE_index; i++){
                        if(RLE[i].iElement_ID!=0){
                                printf ("CREF %d,S %d, E %d, R %d, EID %d, PC: %d    ",
RLE[i].iColourRef, RLE[i].iStart_Pos, RLE[i].iEnd_Pos, RLE[i].iRow, RLE[i].iElement_ID,
RLE[i].iParent_Child_Exists);
                                //printf("\n");
                                if(RLE[i].iParent_Child_Exists!=0){
                                        int N_ID=RLE[i].iElement_ID;
                                        printf("Pc %d Pi: ",Neighbours[N_ID].iParent_Count);
                                        for(j=0;j<Neighbours[N_ID].iParent_Count;j++){
                                                printf("%d ", Neighbours[N_ID].iParents[j]);
                                        }
                                        //printf("\n");
                                        printf("Cc %d Ci: ",Neighbours[N_ID].iChild_Count);
                                        for(j=0;j<Neighbours[N_ID].iChild_Count;j++){
                                                printf("%d ", Neighbours[N_ID].iChildren[j]);
                                        }
                                }
                        printf("\n");
                        }
                }

                break;
        case 2:
                for(i = 0; i<=RLE_index; i++){
                        printf ("CREF %d,S %d, E %d, R %d, EID %d, PC: %d    ", RLE[i].iColourRef,
RLE[i].iStart_Pos, RLE[i].iEnd_Pos, RLE[i].iRow, RLE[i].iElement_ID, RLE[i].iParent_Child_Exists);
                        //printf("\n");
                        if(RLE[i].iParent_Child_Exists!=0){
                                int N_ID=RLE[i].iElement_ID;
                                printf("Pc %d P: ",Neighbours[N_ID].iParent_Count);
                                for(j=0;j<Neighbours[N_ID].iParent_Count;j++){
                                        printf("%d ",
RLE[Neighbours[N_ID].iParents[j]].iElement_ID);
                                }
                                //printf("\n");
                                printf("Cc %d C: ",Neighbours[N_ID].iChild_Count);
                                for(j=0;j<Neighbours[N_ID].iChild_Count;j++){
                                        printf("%d ",
RLE[Neighbours[N_ID].iChildren[j]].iElement_ID);
                                }
                        }
                        printf("\n");
                }

                break;
        case 1:
                for(i = 0; i<=RLE_index; i++){
                        printf ("CREF %d,S %d, E %d, R %d, EID %d, PC: %d\n", RLE[i].iColourRef,
RLE[i].iStart_Pos, RLE[i].iEnd_Pos, RLE[i].iRow, RLE[i].iElement_ID, RLE[i].iParent_Child_Exists);
                }
                break;

        case 0:
                for(i = 0; i<=RLE_index; i++){
                        printf ("ColourREF %d,Start %d, End %d, Row %d\n", RLE[i].iColourRef,
RLE[i].iStart_Pos, RLE[i].iEnd_Pos, RLE[i].iRow );
                }
                break;
        }
        return;
}

void self_check(){

  int iChecksum=0;
```

```
  for(int i=0;i<=iKey_Counter;i++)
        iChecksum += iKey_Sequence[i];

        //iChecksum is the syndrome value for the program run

        if(iChecksum%2==0)
              iParity=0;
        else
              iParity=1;

        //Calculate what correct parity would be
        if(iParity!=iAccepted_Parity)
              //possible error has occurred
        else
              //either no error occurred OR
              //even number of errors occurred

        //Calculate whether syndrome value is acceptable

        //Calculate the possible correct syndrome values
        //if syndrome value acquired is not equal to any of these
        //error has occurred
  //otherwise, the program executed correctly

        return;
}
```

Appendix 4

Publications

The papers attached in this section are those accepted for presentation at international seminars, detailing portions of the work in this thesis.

The first paper, "Correlating Failure Modes With Component Reliability In Synthesis Of Self-Checking Control Unit Circuits", was presented in an international conference on Electronics in Singapore, in September 2001.

The second paper, "Size/Position Identification in Real-Time Image Processing using Run Length Encoding", will be presented in the U.S.A., in May 2002, at a conference on Instrumentation and Measurement.

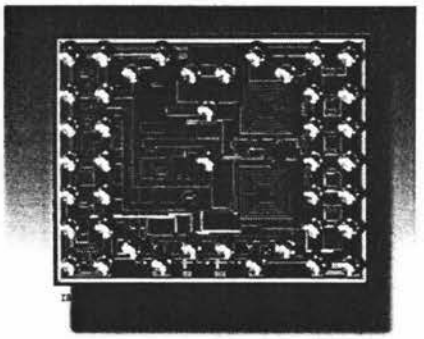
ISIC-2001

9th International Symposium on
Integrated Circuits, Devices & Systems

3 -5 September 2001
Marina Mandarin, Singapore

PROCEEDINGS

Low Power and Low Voltage Integrated Systems



Organised by:
Nanyang Technological University
School of Electrical and Electronic Engineering

Supported by:
IEEE Singapore Section
IEE Singapore Centre
Singapore Exhibition & Convention Bureau

Sponsored by:
Celestry Design Technologies (former BTA-Ultima)

Nanyang Technological University

IEEE Singapore Section

IEE Singapore Centre

celestry

CORRELATING FAILURE MODES WITH COMPONENT RELIABILITY IN SYNTHESIS OF SELF-CHECKING CONTROL UNIT CIRCUITS

S. Demidenko and K. Subramaniam

Institute of Information Science & Technology, Massey University,
Riddett Complex, Palmerston Nortn, Private Bag 11222, New Zealand

Abstract: The paper deals with design of on-line self-testing control unit circuits. The control units of the micro-program type are considered in particular. The main topic discussed in the paper is the relationship between hardware errors in micro-program control units and the faulty transitions in the program flow-graph caused by the errors.

1. INTRODUCTION

Traditionally, *Micro-Program Control Unit* (MPCU) is designed around a memory containing micro-instructions [1-3]. Each micro-instruction consists of a control code field and a next address code field. The sequence of micro-instructions is determined by the sequence in the next address generator (Fig. 1). The generator produces addresses depending on the code in the next address field, on the external control signals (start, stop, set at specific address, etc.), and on the values of logic signals (flags) coming out of the controlled device or system (the flags are required to organize conditional jumps in the program).

Two major approaches in on-line monitoring of MPCU are usually employed in order to satisfy high dependability criteria. The first is related to checking MPCU instruction outputs by employing the techniques developed for the highly dependable data path architectures [4]. The other includes techniques ranging from MPCU state description based on error-detecting codes and to control-flow monitoring by on-chip/off-chip watchdogs [5-8]. The approaches are significantly different in their fault coverage, hardware overhead, performance degradation, and error detection latency.

In this paper we deal with the method based on employing special check tokens (keys) incorporated into the body of the micro-program. The key (binary word) is put into correspondence to each micro-instruction or to a set of micro-instructions. The keys are then interleaved with micro-instruction during the program execution. This allows checking the control flow by comparing the actual sequence of the check keys with a stored (or calculated) reference sequence corresponding to fault-free operation of the control unit. Different key generation and compression techniques can be used to process of sequences of reference and actual keys. It leads to a wide variety of the possible architectural realizations for self-monitored MPCUs.

2. ON-LINE PROGRAM FLOW MONITORING

A control unit can be described using a flow graph representing its operation in terms of succession of generated instructions. Every vertex (denoted Q_i, where $i \in \{1,2,3,...,N\}$, N is the total number of the vertices) corresponds to an instruction (in some cases a vertex could correspond to a group of instructions, however here this case is not discussed). The check keys are put into correspondence to the vertices of the flow graph.

Faults of different types can occur during the program execution because of errors in MPCU:

(a) disappearance of an instruction(s) in the sequence;
(b) appearance of an additional instruction(s);
(c) erroneous transition on logical conditions;
(d) erroneous transition in a functional vertex;
(e) substitution of one micro-instruction by another.

Faults in the instruction fetch sequence correspond to faulty transitions between flow graph vertices, denoted as $Q_i \rightarrow Q_j$. The fault detection probability value P_d is normally used to estimate the quality of the flow graph checking. Expression $P_d(Q_i \rightarrow Q_j)$ is used to denote probability value that a single faulty transition from vertex Q_i to Q_j is detected by on-line means. The value depends on the flow graph itself, on the type and distribution of the check keys, on the chosen checking strategy, and on the employed check key processing methods. These notations can be used to get a general description for the fault detection in the MPCU program flow [9].

(i) Complex event of detection of faulty transition $Q_i \rightarrow Q_j$ includes the following three sub-events:

(a) Occurrence of a fault in some vertex Q_i,

(b) Occurrence of the faulty transition $Q_i \rightarrow Q_j$,

(c) Detection of the transition by diagnostic tools.

(ii) Probability that a fault is occurred in some vertex of a flow graph depends on the reliability characteristics of the MPCU hardware, on the flow graph, on the ranges of the input data and distribution of the data over these ranges, as well as on the probabilities of logic signals (flags) used in conditional transitions (case statements, "if", "while"). Numerical evaluation of the probability can be obtained theoretically (on the basis on reliability of MPCU components, characteristics of input data, and parameters of expected operation of the system controlled by MPCU), or empirically using an actual MPCU. The values can be summarized as one column matrix - vector $\boldsymbol{F}$ (*Vector of Fault Occurrence*) of

dimension N (N is the number of flow graph vertices). Elements of $\boldsymbol{F}$ correspond to vertices of the flow graph and are equal to the probabilities of fault occurrence.

(iii) Probabilities of faulty transitions between specific vertices can be presented by the square matrix $\boldsymbol{T}$ (*Transition Matrix*) of dimension $N \times N$. Each element of the matrix is the probability of a faulty transition between the two vertices. The probability evaluations can be found empirically by operating actual MPCU. Alternatively, they can be found analytically or from computer simulation using fault insertion into different parts of the MPCU.

(iv) Probability of detecting faulty transitions by means of on-line checking tools can be given by $N \times N$ square matrix $\boldsymbol{D}$ (*Detection Matrix*). Values of $\boldsymbol{D}$ are determined by the employed method of on-line checking. For example, if assuming a single transition error, and checking performed on every cycle of MPCU operation, then an element of such a matrix is equal to *1* when a faulty transition is detected; otherwise it is equal to *0*. If the checking is organized in the interval-by-interval mode (i.e., just one reference value is used for a group of linked vertices), and if occurrence of multiple errors is possible [8], the detection probabilities may have values in the range between *1* and *0*.

(v) Matrix expression for the probability of faulty transitions detection can be presented as: $\boldsymbol{P = F \times T \bullet D}$, where "$\bullet$" is the element-by-element multiplication (not the conventional matrix-by-matrix multiplication operation). The elements of $\boldsymbol{P}$ are probabilities of detecting faulty transitions between the corresponding vertices of the flow graph.

3. SYNTHESIS OF SELF-CHECKING MPCU

The general procedure [8] for synthesis of self-monitoring MPCU with pre-specified level of fault coverage can be presented as following steps:

(i) By analyzing the flow-graph, the architecture of MPCU as well as the other relevant data (probability of external control signals, probabilities of conditional jumps initiated by the flag signals, etc.) elements of the vector $\boldsymbol{F}$ and matrix $\boldsymbol{T}$ are found.

(ii) General attainability of the fault detection is checked by calculating *max* possible fault coverage for given $\boldsymbol{F}$ and $\boldsymbol{T}$. At this stage it is assumed that all elements of $\boldsymbol{D}$ are 1's (i.e., all possible faulty transitions can be detected). If it is found that the required fault coverage can not be reached, then the algorithm, instruction set, data range, etc., must be modified, and MPCU architecture is re-configured, if necessary. Then - return to the step (i). Once the required coverage is found to be attainable - proceed to step (iii).

(iii) Elements of $\boldsymbol{D}$ are defined from the required fault coverage and values of $\boldsymbol{F}$ and $\boldsymbol{T}$.

(iv) On-line checking technique is selected and the check keys are assigned to the flow graph vertices.

(v) Corresponding changes are introduced into the MPCU to technically realize the checking.

The presented methodology allows also evaluating the fault coverage, latency and hardware overhead when the reliability parameters, check key values, method of fault detection are given *a priori*.

A simple example can be used as an illustration. The program flow graph is shown in Fig. 2 along with the vertex check keys and the reference values. Let the probability of a fault occurring is 1/2 in the vertex Q_1; 1/8 in the vertex Q_2; 1/16 in each of the vertices Q_3 and Q_4, and 1/4 in the vertex Q_5. The corresponding vector $\boldsymbol{F}$ is shown in Fig. 3. Let the faulty transition probability graph and the transition probability matrix $\boldsymbol{T}$ is as shown in Fig. 3. According to the flow graph and the check keys allocation the following faulty transitions will be detected by diagnostic means: $Q_1 \to Q_4, Q_1 \to Q_5, Q_2 \to Q_1$, $Q_2 \to Q_2, Q_2 \to Q_3, Q_2 \to Q_4, Q_3 \to Q_1, Q_3 \to Q_2$, $Q_3 \to Q_3, Q_3 \to Q_5, Q_4 \to Q_1, Q_4 \to Q_2, Q_4 \to Q_3$, $Q_4 \to Q_4; Q_5 \to Q_1, Q_5 \to Q_2, Q_5 \to Q_3, Q_5 \to Q_4$. Other faulty transitions remain undetected. The matrix $\boldsymbol{D}$ is as presented in Fig. 3. Finally, the matrix $\boldsymbol{P}$ of the detection probability will be as follows:

$$P = \begin{bmatrix} 1/2 \\ 1/8 \\ 1/16 \\ 1/16 \\ 1/4 \end{bmatrix} \times \begin{bmatrix} 1/2 & 0 & 0 & 1/4 & 1/4 \\ 1/4 & 1/4 & 1/4 & 1/4 & 0 \\ 1/4 & 0 & 1/2 & 0 & 1/4 \\ 1/4 & 0 & 1/2 & 1/4 & 0 \\ 1/2 & 0 & 0 & 0 & 1/2 \end{bmatrix} * \begin{bmatrix} 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} 0 & 0 & 0 & 1/8 & 1/8 \\ 1/32 & 1/32 & 1/32 & 1/32 & 0 \\ 1/64 & 0 & 1/32 & 0 & 1/64 \\ 1/64 & 0 & 1/32 & 1/64 & 0 \\ 1/8 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The total detecting probability can be obtained by summing all the elements of matrix $\boldsymbol{P}$:

P_d = (1/8 + 1/8) + (1/32 + 1/32 + 1/32 + 1/32) + (1/64+1/32 + 1/64) + (1/64 + 1/32 + 1/64) + 1/8 = 5/8.

Thus, the faulty transitions in the MPCU will be detected with the probability of 62.5%.

4. ELEMENTS OF MATRIX *F* AND *T*

<u>*Fault occurrence vector*</u> $\boldsymbol{F}$ can be represented as element-by-element (middle) product of two vectors $\boldsymbol{F'}$ and $\boldsymbol{F''}$. The first specifies the probabilities of reaching the corresponding flow graph vertices in course of program execution. The second – the probabilities of a fault occurrence while the program is in the vertices.

Vector $\boldsymbol{F'}$ depends mainly on the program flow. For example, for a linear flow-graph all the elements of the vector are of equal value *1*, while for the flow-graph having conditional transitions or loops some vertices have higher reaching probability than the other ones. Higher reaching probabilities correspond to the vertices that are inside loops or those of the convergence of different branches.

Values of $\boldsymbol{F''}$ depend on parameters of reliability of the hardware used to implement the MPCU. The results of calculations based on the traditional reliability and failure prediction methods can be used. The reliability R_i of MPCU when the program is at some vertex i of the program flow-graph can be presented as $R_i = e^{-\lambda t_i}$, where λ is the MPCU failure rate, and t_i is the time elapsed since the beginning of

MPCU operation and the moment of reaching the vertex. The failure probability at the vertex is equal to $f_i'' = 1 - R_i$, $i \in \{1,2,3,..., N\}$.

Transition matrix $\boldsymbol{T}$ depends, first of all, on the character and the place of the physical fault occurrence within MPCU, as well as on the flow-graph of the micro-program. When no *a priori* data is available, all the transitions can be taken as equally probable. However, this could be used only as a last resource. It would ultimately lead to an overall improvement for the fault coverage, however not as fast as it is often required. More realistic data can be obtained by considering reliability of different part of MPCU used to generate new micro-instruction at each program-graph vertex, and inserting corresponding fault models (stuck-at, bridging, inverting, etc.) into the architecture.

As an example, let a linear program fragment consists of 8 consecutive vertices and their addresses are called in the program in the incrementing by one way. The initial address is 000_2, the final – 111_2. No transitions are originated from the final vertex. Let the MPCU fault causes a single inversion error in the 3-bit address field (***) (Fig. 1). Finally, let the probability of occurrence of the inversion error are equal for all the bits of the address. In such a case, the state transition diagram and the matrix $\boldsymbol{T}$ will be as shown in Fig. 4.

The presentation will discuss properties and values of the matrix $\boldsymbol{T}$ for some most common type of errors (stuck-at, inversion, shorts, etc.).

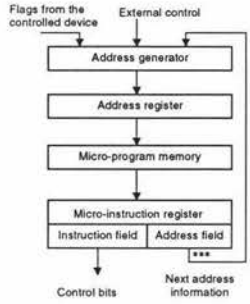


Fig. 1. Micro Program Control Unit

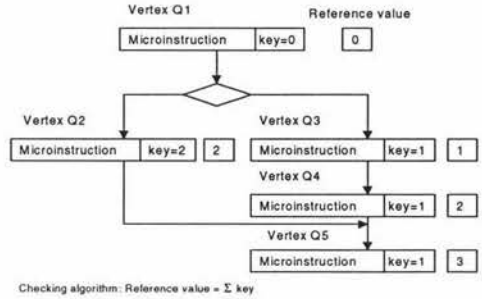
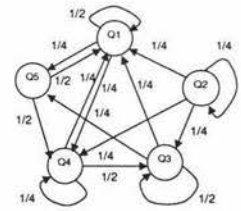


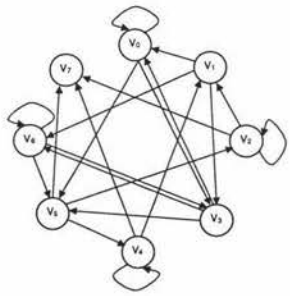
Fig. 2. Coded flow-graph (checking algorithm – summing)

$$F = \begin{bmatrix} 1/2 \\ 1/8 \\ 1/16 \\ 1/16 \\ 1/4 \end{bmatrix}$$



$$T = \begin{bmatrix} 1/2 & 0 & 0 & 1/4 & 1/4 \\ 1/4 & 1/4 & 1/4 & 1/4 & 0 \\ 1/4 & 0 & 1/2 & 0 & 1/4 \\ 1/4 & 0 & 1/2 & 1/4 & 0 \\ 1/2 & 0 & 0 & 0 & 1/2 \end{bmatrix} \qquad D = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

Fig. 3. Fault occurrence vector $\boldsymbol{F}$, fault transition diagram and fault transition matrix $\boldsymbol{T}$, fault detection matrix $\boldsymbol{D}$.



$$T = \begin{bmatrix} 1/3 & 0 & 0 & 1/3 & 0 & 1/3 & 0 & 0 \\ 1/3 & 0 & 0 & 1/3 & 0 & 0 & 1/3 & 0 \\ 0 & 1/3 & 1/3 & 0 & 0 & 0 & 0 & 1/3 \\ 1/3 & 0 & 0 & 0 & 0 & 1/2 & 1/3 & 0 \\ 0 & 1/3 & 0 & 0 & 1/3 & 0 & 0 & 1/3 \\ 0 & 0 & 1/3 & 0 & 1/3 & 0 & 0 & 1/3 \\ 0 & 0 & 0 & 1/3 & 0 & 1/3 & 1/3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Fig. 4. Transition diagram and corresponding transition matrix $\boldsymbol{T}$ for linear program fragment.

5. REFERENCES

1. C. Kraft, W. Toy: Microprogrammed control reliable design of small computers, Prentice-Hall, 1981
2. M. Mano: Computer System Architecture, Prentice-Hall, 1993
3. W. Stallings: Computer Organization and Architecture, Prentice-Hall, 2000
4. K. Pradhan: Fault-tolerant computer system design, Prentice-Hall, 1996
5. R. Parekhji, G. Venkatesh, S. Sherlekar: Concurrent error detection using monitoring machines, IEEE Design & Test of Computers, Vol. 12, No. 3, 1995, pp. 24-31
6. M. Schutte, J. Shen: Exploiting instruction-level parallelism for integrated control-flow monitoring, IEEE Trans. Comp., 44, pp. 129-140, 1994
7. A. Mahmood, E. McCluskey: Concurrent error detection using watchdog processors, IEEE Trans. Comp., 37, pp.160-174, 1988
8. S. Demidenko *et al*: Concurrent self-checking for microprogrammed control units, IEE Proc-E, 138, pp. 377-388
9. S. Demidenko *et al*: New approach to synthesis of self-checking microprogrammed control units with specified fault-detection probabilities, IEE Proc.-E, 138, pp. 389-396

IEEE Instrumentation and Measurement
Technology Conference
Anchorage, AK, USA, 21-23 May 2002

Size/Position Identification in Real-Time Image Processing using Run Length Encoding

C. H. Messom[1], S. Demidenko[2], K. Subramaniam[2] and G. Sen Gupta[3]
[1]II&MS, Massey University, Albany, New Zealand
[2]IIS&T, Massey University, Palmerston North, New Zealand
[3]School EEE, Singapore Polytechnic, Singapore
Email: C.H.Messom@massey.ac.nz, S.Demidenko@massey.ac.nz, Karthik@xtra.co.nz, G.SenGupta@sp.edu.sg

Abstract *– This paper presents the use of Run Length Encoding (RLE) for real-time image processing. RLE compresses the image and allows the processing algorithm to efficiently identify the size and position of the objects in the image. The algorithm is application to a robotic vision system enable to reliably identify 11 objects in the 16.67ms sample time of an interlaced NTSC video image. It is shown that the time complexity of the compression phase is linear in the size of the image while the image processing phase is linear in the number of object in the image and the number of lines occupied by the objects in the image.*

Keywords *– Real-time Image Processing, Compressed Image Processing, Robotics Vision System*

I. INTRODUCTION

Vision systems are the primary sensory input to advanced robotic applications such as mobile robotics, agricultural robotics and inspection systems [1]. However, vision causes a significant processing burden in intelligent robotics since the majority of image processing algorithms are computationally extremely expensive. For this reason any improvement in the vision algorithm leads to significant benefits in terms of achieving real time performance. Until recently reliable real-time performance has only been possible using dedicated hardware implementations. Vision systems implemented on the basis of dedicated hardware (for example, using special full custom design ASICs) are fast and reliable. However they are inflexible and expensive.

Intelligent robot control systems that make use of commodity hardware (readily available, or programmable systems) can be flexible and cheap. However they normally suffer from limited processing speed. Most commodity vision systems use video signals as input to the image capture subsystems and so provide frame rates of approximately 30Hz and field rates of 60Hz for interlaced images. Processing these images with useful resolution (240 x 320 and above) in the 33.3 ms and 16.67 ms sample times respectively presents a significant challenge. The sample time cannot be devoted completely to vision since the intelligent control algorithms will not be able to complete within the real time constraints.

In general image processing algorithms are applied to uncompressed images. The reason for this is that most of the popular image compression techniques distort the color, size and position of the objects in the image that can only be recovered by decompressing the image. Processing of compressed images in real time is also problematic when the compression algorithm is computationally expensive.

Paper [2] reported a real-time vision system using incremental tracking, whereby a window around each object is created. Once the location of an object has been determined, its movements are tracked only within the window. The position of this window is adjusted each frame after identifying the new position of the object.

This technique is used mainly due to the savings it offers in processing time, as the whole image of the playing field does not need to be analyzed. Instead, only the tracking windows for each object need to be analyzed. The technique analyzes a very small proportion of the field, leading to the intermittent problem of losing objects that stray outside their tracking windows. This problem is particularly hard to solve for fast moving unpredictable objects. Nevertheless, this method was used in order to achieve real-time image processing.

This paper examines the processing of compressed images in real-time using the Run Length Encoding (RLE) algorithm. The algorithm is not computationally expensive. It allows an image to be encoded with just a single access to each pixel. The RLE algorithm was applied to the problem of finding the difference between two binary images in [3]. It was shown that an algorithm could be efficiently implemented using systolic architecture. Such an implementation has the advantage that the computational complexity (and the execution time) increases just linearly with the size of the image.

The RLE algorithm can be implemented on commodity hardware in the robotic application domain (for example, in robot soccer visual systems) [4-5]. This paper examines the *full* image processing of Run Length Encoded 24-bit RGB images using commodity hardware. It shows that once the compression phase is completed, the algorithm is essentially linear in the number of objects and size of the image. The implementation discussed in section V is more than four times faster than that reported in [5].

The RLE encoding algorithm preserves the position, size and color of the objects in the image. The compressed image is processed to identify the size and positions of the objects in the image in real-time. RLE is a loss-less compression technique if the exact pixel values are preserved. The RLE compression used in this paper produces significant losses since a range of RGB values are represented by a single color value. The fact that the original image cannot be recovered by the RLE compression is not a problem in this application since the compressed image is not required for any post-processing other than object recognition. It is shown that the performance of the image processing on the compressed image is significantly faster than standard threshold based techniques when used for robotic vision applications.

II. RUN LENGTH ENCODING

The RLE algorithm takes a line of pixels in the image and identifies the matching color identifier for each pixel. The color identifiers represent a set of RGB values that form a convex subspace of the 24-bit RGB color space. A covex partition of the RGB color space must be provided for each color identifier before this can be achieved (see Figure 1). This convex partition is specified by a range of RGB values namely, R_{min} - R_{max}, G_{min} - G_{max}, B_{min} - B_{max}.

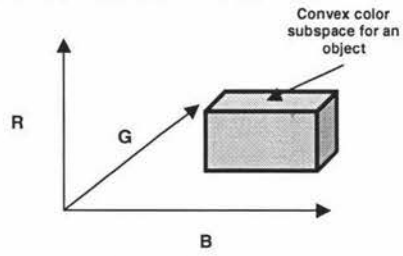


Figure 1. Convex color subspace

The matching color partition for each color identifier is created based on the expected color of the object in the image, which is dependent on the actual color of the objects and the lighting conditions. Given a fixed set of objects and controlled uniform lighting, a constant set of convex color partitions can be defined.

Once each pixel has been classified as being part of a particular object's color partition, a line of pixels is run length encoded by identifying the start and end position of each object (Figure 2). The start and end position of the line of pixels that are from a single object and the color identifier of the object forms the components of the run length element given in Equation 1.

Run Length Element = (ColorID, StartIndex, EndIndex) (1)

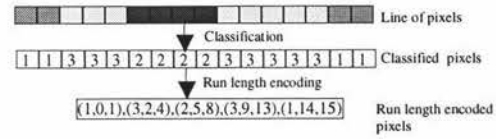


Figure 2. Run Length Encoding of a line of pixels

For objects of less than 3 pixels in size the compression algorithm is very inefficient. This is because the RLE algorithm requires 3 integers to code each object. However the existence of small objects does not necessarily make the algorithm inefficient for the whole image since large objects (larger than 20 pixels in width), including the background, have compression ratios in excess of 75%. If the image has only small objects the compression ratio of the objects' pixels will be small, however the background will be very large in such images producing a large overall compression ratio. If the background is not of interest for the image processing algorithm, the compression phase can skip over the background pixels thus increasing the compression ratio even further.

The compressed image is then processed to identify the objects in the image, their sizes and their positions. The run length encoded lines are compared to identify the adjacent elements with identical color classifications (Figure 3). These run length elements are tagged with object identifiers (Equation 2) that are unique for each element.

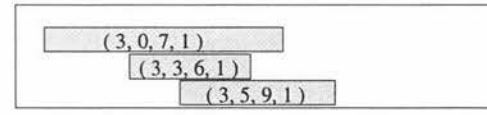


Figure 3. Grouping of objects in compressed image

Tagged Run Length Element =

(ColorID, StartIndex, EndIndex, ObjectID) (2)

The run length elements that have no neighbors of the same color are allocated a new object identifier (ObjectID). If the run length element is adjacent to a run length element of the same color it is assigned the same object identifier as the neighbor. This approach guarantees that each object of a particular color is tagged with a unique object identifier.

Starting from the top of the image the object identifiers can be consistently assigned. However when a run length element has more than one adjacent element in the previous row (Figure 4) two object identifiers can exist for a single object. This

is resolved by re-labeling the second neighboring element with the correct object identifier.

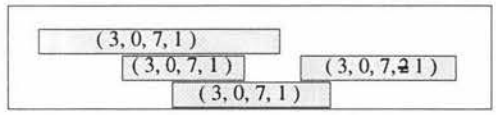


Figure 4. Reassigning object identifiers

Once the elements have been correctly classified with unique object identifiers, they are grouped together by totaling up the run length elements that share the same object identifier. This provides the size of each object. The position of each object is determined using a center of gravity calculation based on the size and position of the run length elements. The size and position of the run length elements are given by the start and end position of the run lengths and the line in the image in which they occur.

III. THE ALGORITHM

The image analysis algorithm has been implemented using two data structures, 'RLE_element' and 'Neighbors'. Each instance of the first structure, 'RLE_element', holds the information for a single sequence of pixels of the same color. If a run of pixels of a particular color occurs, an instance of the 'RLE_element' structure stores the location and length of the run of pixels, the object ID number, the color partition the object falls into, and whether it has any neighbors of the same classification (Figure 5).

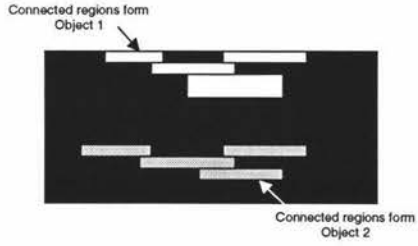


Figure 5. Neighboring RLE elements forming objects

An instance of the second structure, 'Neighbors', is used to store for each RLE_element, the locations of all neighboring elements falling into the same color partition. Thus, when a run of pixels is recorded in a RLE_element structure, the region around the run of pixels is also checked for other runs of pixels, with the same color categorization. Every run that falls into the same color category is stored in the current object's corresponding 'Neighbors' structure. The structure holds all the neighboring objects' locations in an array (Figure 6).

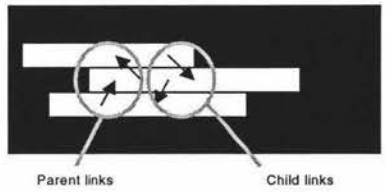


Figure 6. Links to neighboring RLE elements of identical color

The whole program passes through the image twice: firstly, to compress the image, and secondly, to process the encoded image. The first pass analyses each row of pixels, until the end of the captured image is reached. In each row, the runs of pixels are classified into their respective color partitions.

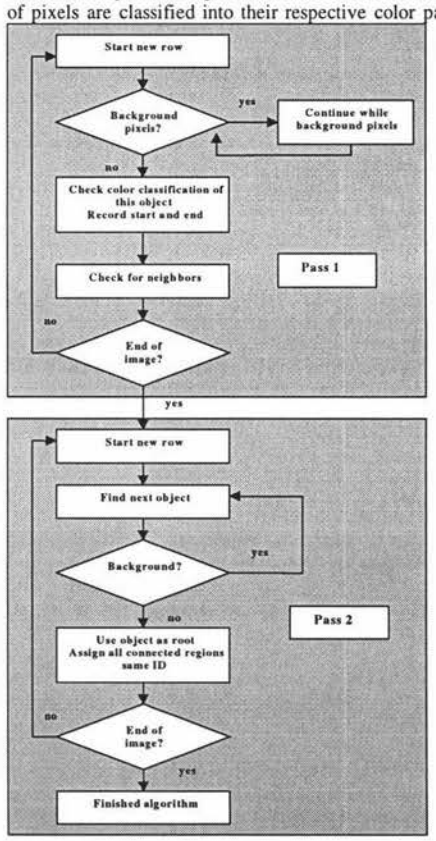


Figure 7. Image-processing flow chart

When an object other than the background is encountered, two processes are executed. Firstly, the object is assigned the next available object ID. Next, the surrounding region of the current object is checked for other objects of the same color category. If any exist, they are considered connected, and belong to be the same object. The connected region is recorded as a neighbor of the current object, and vice versa. The background, or playing field, is not recorded with the 'neighbors' field.

The second pass through the image selects each object, and uses it as the root of a tree. From this root, a recursive depth-first traversal is performed on all the connected regions that have been recorded as 'Neighbors' in the first pass. In this process, all connected regions will be assigned the same object ID, as the tree root. If the connected regions already have been assigned an object ID, the tree overwrites its object ID. The result of this second pass through the image is that all connected portions are identified with the same object. The flow chart in Figure 7 illustrates the processes in the algorithm.

IV. ALGORITHM OPTIMIZATION

The RLE compression phase examines each pixel and identifies the corresponding object color based on the predefined convex color space (Figure 1). This calculation can be achieved by 6 comparisons of the pixel's RGB value with the defined range of the convex color space. An optimization that has been applied [5] is to use a lookup table to extract the required color ID. The tradeoff of this approach is that the improved processing speed comes at the expense of memory required. For a 24 bit RGB system a lookup table of size 16MB is required (256*256*256 bytes). This lookup table must be created off-line as the time taken to create such a large lookup table is significant. A further disadvantage with the lookup table method is that an adaptive object color partition is difficult to implement. (An adaptive object color partition may be required in systems where the lighting conditions change over time).

A method to overcome the large space requirement of an RGB lookup table was originally proposed by [5]. This required three lookup tables, one for each R, G and B component. These three lookup tables are projections of the 3 dimensional RGB lookup table into the R, G, and B axes. Given a particular pixel value, each lookup table will return the set of color identifiers that match that pixel value. The three returned sets can then be intersected to identify which actual object color corresponds to the given RGB value.

The sets can be implemented by representing each object color with a binary identifier (1, 2, 4 etc) and returning a set of possible object colors by bit-wise OR of the required object IDs. For example, if the R component value of 55 corresponded to object color 1 and 4 then the R lookup table will return 5 (which is 1 bit-wise OR 4).

The set intersections can then be implemented with the bit-wise AND of the results from the three (R, G and B) lookup tables (Equation 3).

$$Color\ ID = R[r]\ bit\text{-}wise\ AND\ G[g]\ bit\text{-}wise\ AND\ B[b] \quad (3)$$

Continuing the example from above, if a G component value of 100 corresponds to object color 1 and 2 while, a B component value of 125 corresponds to object 1 and 8, then the RGB value of (55,100,125) will correspond to object 1.

The size of each R, G and B lookup table depends on the number of object colors required. One byte can accommodate 8 object colors. A system with 8 object colors will require each lookup table to be 256 bytes, a total of 768 bytes for the three R, G and B lookup tables. The 16MB RGB lookup table and the three 256 byte R, G and B lookup tables were tested in software on the current system. As expected there was not any significant difference in time performance between the two methods, but there was a significant improvement in the memory space required.

V. PROTOTYPE APPLICATION

Robotic soccer has been chosen as the test bed for the proposed algorithm and its software implementation using the commodity hardware. In brief, the robot team plays against the opponent team on a specified size field. All the team robots are equipped with wireless communication facilities to receive the instructions from the central computer. The computer controls the robots by transmitting information to them regarding the location of the ball, where the robot must go next, where its opponents are, and where its teammates are. The robots operate using a common vision system, which views the playing field from above (Figure 8). The identified colored objects are grouped together to produce positions and orientation of the robots and obstacles in the image.

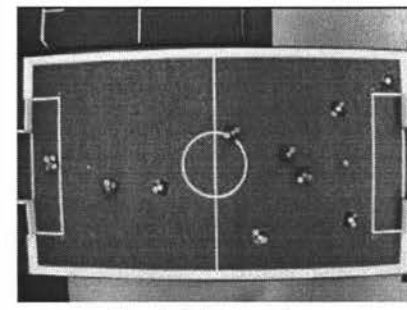


Figure 8. Robot Soccer Image

Classifying the objects in the image into opponent robots, own robots, or the ball is done by recognizing colored patterns on the upper faces of the robots. This is the final process in the overall image processing algorithm, following the first phase – the object location. The robot controller then makes decisions on its game strategy based on the positions of all the players and the ball and transmits the required information to the mobile robots.

VI. RESULTS

The proposed algorithm was tested on commodity hardware: a Flashbus Pro image capture card [6], 1GHz Pentium III with 256MB RAM. The signal was an interlaced NTSC video, each half frame delivered at 60Hz. Using an image of 240 x 320 pixels the system was used to identify the size and position of 32 colored objects. The processing consumed 5 ms of CPU time a significant improvement on the 33 ms reported in [5].

The technique presented here compares favorably to the color threshold based approaches discussed in [7], which take more than 33ms on the hardware above. The time performance of the RLE image processing algorithm compares favorably with the incremental tracking, real-time approach presented in [2]. However, the main advantage over incremental tracking is that RLE image processing is a full image scanning technique and therefore does not suffer the problem of losing track of fast moving and unpredictable objects.

The RLE algorithm's time complexity is linear in height and linear in width of the image since each pixel is processed once. The processing of the compressed image is linear in the number of lines occupied by each object and the number of objects in view. This is because the background is skipped over and the grouping process acts on the run length elements of the objects in view. The image processing phase of the algorithm scales extremely well. If the number of objects in the image is fixed, increasing the resolution of the image will only incur a linear cost in image processing time.

For a small number of small objects in a high-resolution image the RLE phase represents the most time consuming part of the algorithm. This can be seen from the example of 11 objects each occupying 16 lines of a 960 x 1280 pixel image. The 1,228,800 pixels are processed by the RLE phase while only 176 (which is 11 objects * 16 lines) run length elements are processed by the grouping phase of the image processing algorithm. Since each operation in the grouping phase is not significantly more computationally expensive than that of the RLE phase, the difference in the number of operations means that the RLE phase dominates the processing time as compared to the image processing phase.

VII. CONCLUSIONS

An application of run length encoding (RLE) to real time image processing was presented. It was shown that the time complexity of this algorithm is very good, particularly in the case of robotics applications where there are a small number of small objects. The algorithm can process a 240 x 320 image within 5 ms which, with the linear increase in processing time with respect to the size of image, we expect the algorithm to process a 480 x 640 resolution image within the 16.67 ms sample time of an interlaced NTSC system.

Further performance improvements can be achieved with the RLE algorithm by modifying the image capture process (as discussed in section IV). If the image's RGB components are fed through look-up-tables that classify the component into the possible objects in view then the RLE algorithm is reduced to just two bit-wise AND operators for each pixel. Implementing this operation in hardware is expected to contribute significantly to the scalability of the RLE image processing system, which will be investigated in future work.

REFERENCES

[1] R. Templer, H. Nicholls and T. Nicolle, "Robotics for meat processing - from research to commercialisation", Industrial Robot, Vol 26, Number 4, 1999.

[2] C.H.Messom, G.S. Gupta and H.L. Sng, "Distributed Real-time Image Processing for a Dual Camera System", CIRAS 2001, Singapore, 2001 pp 53-59.

[3] F. Ercal, M. Allen, and F. Hao, "A Systolic Image Difference Algorithmfor RLE-Compressed Images", IEEE Transactions on Parallel and Distributed Systems, Vol. 11, No. 5, May 2000.

[4] H. Kitano, M. Asada, Y. Kuniyoshi, I. Noda, E. Osawa and H. Matsubara, "RoboCup: A Challenge Problem for AI", RoboCup-97: Robot Soccer World Cup I, Springer Verlag, London, 1998.

[5] J. Bruce, T. Balch and M. Veloso, "Fast and Inexpensive Color Image Segmentation for Interactive Robots", IROS 2000, San Francisco, 2000.

[6] http://www.integraltech.com/Products/FrameGrabbers.html

[7] J. Baltes. "Practical camera and colour calibration for large rooms". In Manuela Veloso, Enrico Pagello, and Hiroaki Kitano, editors, RoboCup-99: Robot Soccer World Cup III, pages 148-161, New York, 2000. Springer.