THE VALIDATION OF EMBEDDED SOFTWARE

Thomas Davidtz

A dissertation submitted to the Faculty of Engineering, University of the Witwatersrand, Johannesburg, in fulfilment of the requirements for the degree of Master of Science in Engineering.

Johannesburg 1986

DECLARATION

I declare that this dissertation is my own, unaided work. It is being submitted for the degree of Master of Science in Engineering at the University of the Witwatersrand and has not been submitted before for any degree or examination at any other University.

THOMAS DAVIDTZ

TWELFTH day of AUGUST 19 86

ABSTRACT

The use of embedded computers in Railway Signalling systems and other highly-critical monitoring and control applications has led to a demand for an effective method of validation of the software within such systems. An important aspect of validation is proving a computer programme to be consistent with its specification.

This dissertation proposes a pragmatic method of proving a machine-code programme to be consistent with its high-level programme specification. A disassembly of the machine-code programme is obtained and automatically analysed in terms of control-flow and data-flow. By using information from the data-declaration portion of the specification, the disassembly listing is translated to a level corresponding to that of the high-level specification. Consistency between the translated programme and the original high-level specification is proved by direct comparison.

The dissertation suggests the validity of the above approach and shows by example, how such an approach may be implemented.

ACKNOWLEDGEMENTS

I wish to acknowledge my debt to:

The SOUTH AFRICAN TRANSPORT SERVICES for sponsoring the research.

Mr Louis Potgieter, Senior District Engineer, SOUTH AFRICAN TRANSPORT SERVICES, for his advice, guidance and enthusiasm.

Professor M.G. Rodd, University of the Witwatersrand, for his advice and support throughout the duration of the research project.

CONTENTS **Page**

LIST OF FIGURES

CHAPTER 1 INTRODUCTION

1.1 Background

In railway signalling, an "interlocking system" is a control system which ensures the safe operation of trains. Until very recently, all interlocking systems were fail-safe, relay-based control systems. These relay-based interlocking systems have evolved to a point where they display extremely high degrees of reliability. However, many of the relays used in relay-based interlocking systems are specialised items which are expensive to manufacture and require routine maintenance.

The interlocking function is essentially the logical manipulation of an input state to produce an output state and is thus ideally suited to implementation by a computer-based system. With the cost of computer hardware continually decreasing, computer-based interlocking systems are becoming an increasingly attractive alternative to relay-based interlocking systems. Several electronic, computer-based interlocking systems are already in use in various countries as pilot schemes for evaluation.

The South African Transport Services, who are responsible for the national railway system, has had two computer-based, electronic interlocking systems commissioned for evaluation. In addition to evaluation of the individual interlocking systems, the South African Transport Services wishes to keep abreast of technology in the field of electronic interlocking systems.

In order to be a viable alternative to relay-based

interlocking systems, electronic interlocking systems must at least match the safety standards of relay-based interlocking systems. This high degree of safety required is normally achieved by hardware redundancy. Software output-comparison and voting are used to isolate faulty components or, in the event of multiple failure, to shut-down the entire system. Software also often does routine hardware-monitoring to check the integrity of hardware components such as RAM and PROM memories. Thus the integrity of the software is of prime importance.

Therefore, before a computer-based interlocking system can be put into use, engineers in the railway organisation must satisfy themselves as to the integrity of the embedded software. Also, if changes are to be made to the software after commissioning, engineers making the changes must be able to show that their changes have not decreased the safety of the railway system controlled by the computer-based interlocking system.

Thus a need for a method of validating software embedded in electronic interlocking systems was required by the South African Transport Services. The research described in this dissertation was sponsored by the South African Transport Services in order to develop such a validation method.

1.2 Statement of the Problem

"The computer's messed it up again!"
"It's not the computer, it's those people who work it!"

These days, most people have at some time or other encountered a computer malfunction or computer-operator

error. These errors manifest themselves in the form of exorbitant water accounts, incorrect bank balances, delayed aircraft schedules and the like. These errors are the errors which arise in "business computers". When these computers are incorrectly programmed or operated, or when they malfunction, the harm they do is to stir-up human emotions varying from irritation to frenzied anger.

There is another class of computers whose consequences of failure from malfunction or incorrect programming are far more dire. These are the "life-critical embedded systems" - the computers that steer aircraft, route trains, monitor nuclear reactors and perform a host of other life-critical functions. These computers simply must not fail. They must do exactly what their users intend them to do, even if they have been programmed to do otherwise! Therein lies the dilemma.

In the world of real-time process-control, a computer is empolyed to do a specific job and nothing else. Two grey areas immediately become apparent. How does one exactly specify the job the computer is to do and how does one precisely translate that job specification into a computer-executable programme? The extent to which these duties can be correctly performed determines the extent to which a computer will do what it is required to do.

With any method of specification and translation, one aims to ensure that the specification exactly represents the requirements and that the programme, translated from the specification, exactly represents the specification. This is the crux of software validation.

Whatever the form of the programme and however it was generated, it must be shown to meet its requirements, whatever form they too, may take. *This is the ultimate* goal of validation. In the real world, however, infinite variations of programming style and technique render *this task impossible*.

Where validation of programmes is essential, programmes *must be written* in a way which will facilitate their validation. The use of haphazard control-flow and "sneaky", elusive data manipulations *renders* a validator's task extremely difficult and eliminates the possibility of automatic analysis of programmes. An automated or *semi-automated* validation technique requires that programmes be written using only allowable constructs and forms of data-manipulation. *This places restrictions* on programmers, but in the words of C.A.R. Hoare, "...and simplicity is the unavoidable price we must pay for reliability!" (*Hoare* [1975] p. 533).

Software is generated *for many very* different applications. Each different application requires programmes to be written to suit that application.

Therefore, programmes are written in a variety of languages and using a variety of data-manipulation techniques, from low-level bit-manipulation to high-level mathematical computation. Therefore, no single validation technique can be expected to be universally applicable. For a particular application, a validation technique must be found which is most suited to the type of *software and to* the software generation process used in that application.

Software written for the control of real-time processes, such as those performed by an electronic

interlocking system, must cater for such things as bit-manipulation and critical timing. In this field, software is often produced as hand-written assembly-language code or compiled from languages allowing low-level manipulation. In railway signalling, fail-safety is of prime importance and so validation of any software for an electronic interlocking system is essential. However, no convincing techniques for validation of low-level programmes exist. The techniques normally used are those borrowed from other areas of application - techniques which were developed with different validation goals in mind.

A survey was conducted of available publications on the subject of software validation, verification and testing (Chapter 2). Almost all publications referred to static code-parsing and dynamic testing of programmes written in high-level languages. Where authors and researchers referred to validation throughout the life-cycle of software, this too, was only up to the point of high-level language generation.

No references were found to the validation of assembler language or machine-code programmes with respect to a higher-level language or specification. No references were found even to the analysis of assembler language or machine-code programmes in environments where no high-level language or specification exists.

It was therefore decided to conduct research aimed at developing a method of validating embedded software with respect to a higher-level language or specification.

1.3 Direction of Research

The aim of software validation is to demonstrate the

consistency between a computer programme and the user requirements. In the generation of software, the extremes of the generation process are marked by user requirements at one end and machine-code at the other. The translation of the user requirements into machine-code is far from standardised, although some common stepping-stones are in use.

From the requirements, some form of programme specification is usually drawn-up. This is typically a formal statement of what the programme must achieve. It may or may not include information on how the programme is to achieve its goal - the important aspect of the specification is the exact definition of the goal. In a contractual context, the programme specification is often the dotted line between a user and a supplier of software.

The specification, if it is not already in the form of a high-order-language, is translated into a high-order-language and then processed by a compiler to produce machine-code. A typical software generation procedure is shown in figure 1.1 overleaf.

In the generation of software for railway signalling and other critical fail-safe applications, the levels of high-level-language and intermediate-language are often omitted. The specification is translated by hand to assembler level and then processed by an assembler to produce machine-code. This is done to gain the advantage of bit-manipulation at the assembler level and sometimes too, for reasons of code length.

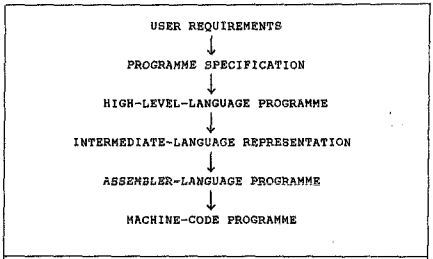


Fig. 1.1 Typical Software Generation Procedure

In order to validate a programme generated by the procedure shown in figure 1.1, a method is proposed whereby the generation procedure is reversed. The proposed procedure begins at the level of the machine-code programme. This programme is then translated backwards through the various levels shown in figure 1.1 until it is at the level of the user requirements.

The proposed validation procedure would thus have generated a set of user requirements derived from the machine-code programme itself. If the user requirements thus obtained can be shown to be consistent with the original user requirements used in the generation phase of the programme, then the programme would have been shown to correctly implement the original user requirements.

Unfortunately, user requirements are not usually formally stated. They normally take the form of

informal human language statements about what is required. The level of the programme specification is normally where formality is first encountered.

Since user requirements are normally informally stated, validation of the programme *specification* with respect to the requirements is a matter of manual interpretation, involving checks for completeness, *consistency* and unambiguity.

The research described in this dissertation was directed towards validating machine-code with respect to its high-level specification. Since the high-level specification can be formally stated, automation of the validation process is possible. One of the major aims of this research was to show how this process could be automated.

1.4 Scope of Research

Programmes and specifications take *on many forms*. In attempting to develop a validation method involving a programme and its specification, the first question must be: *what type of programme* and what type of specification?

1.4.1 Programme

A high-order-language programme suffers many manipulations and changes of appearance before it can instruct a *central processing* unit. It is compiled or interpreted; library functions and routines are called and linked; lower level representations such as P-code or *assembler* are generated and only finally is a string of executable instructions produced. To assume that a

high order language is an exact representation of the instructions which will be given to a central processor is to ignore the fallibility of these manipulators and their operators.

The validation philosophy proposed in 1.2 involves analysis and upward-translation of machine-code to prove its consistency with a high-level specification. Input to the proposed validation procedure is thus machine-code. This has the additional advantage that embedded software which was written without reference to a specification can be subjected to the same analysis and upward-translation processes. This will greatly assist understanding of such software when necessary, for example when a modification is to be made to the software.

It is also intuitively correct that the level at which a machine executes instructions should be the end of the generation phase and beginning of the validation phase of those instructions.

1.4.2 Specification

Many methods of software specification are in use, for example SPECK (Quirk [1983]), PSL/PSA (Teichroew and Hershey [1977]), SADT (Ross and Schoman [1977]) and ESPRESO (Ludewig [1981]). In the railway signalling department of the South African Transport Services, the software specification method in use is P-notation (Young [1980]).

P-notation is of a lower level than most other specification languages or methods, being roughly at the level of a high-level-language such as Pascal. This level of specification language was chosen by the

signalling department because it is used to specify programmes which are then coded directly from it as hand-written assembler.

P-notation, as presented by Young, was found by programmers in the signalling department to be inadequate in certain areas, particularly those of data-type specification and Boolean variable handling. Thus, as it is used in the signalling department, P-notation is a modified version of Young's original P-notation. A description of P-notation, as it is used in the signalling department is contained in Appendix B.

It was not within the scope of this research to assess the effectiveness of modified P-notation for application in the signalling department, nor to compare it with other specification languages in use. Since it is already in use in the department and found to be effective by programmers, modified P-notation was selected as the specification language for use in this research.

1.4.3 Automation

Programmes are often long. Humans make mistakes. In fact, the longer programmes are, the more likely are human validators to make mistakes.

Whatever guise a validation method may take, it is likely to possess the attributes of rigour and repetitiveness. This will render it tedious and time-consuming for human execution. Automation should thus be a major consideration in the development of any validation method or procedure. Errors which would inevitably arise in manual validation exercises would

also be avoided.

One of the major goals of this research was thus to automate the proposed validation procedure wherever possible, or at least to demonstrate that it could be automated.

1.4.4 Stated goal

The goal of this research was to devise a method of showing a machine-code programme, as executed by a microprocessor, to be consistent with its specification in P-notation. Maximum automation of this process was of prime importance.

1.5 Overview of Dissertation

The need for an effective method of validation of software for fail-safe, real-time process control systems was the motivation for the research presented in this dissertation.

A software validation method has been proposed which is a reverse of the typical software generation process.

The proposed method is based on the hypothesis that each stage of the generation process can be validated by translation of its product to the level of the product of the previous stage and validating by comparison. For example, a high-level-language programme can be upward-translated to the level of its specification and compared with the specification. This would validate the specification-to-high-level-language-programme translation stage of the generation process.

Since user requirements are normally informally presented, the reverse translation, or validation of a programme specification against these requirements, is also an informal process. However, since a programme specification can be formally presented in a specification language, software generation processes from that level right down to machine-code can be formally validated by the proposed method of reverse-translation.

The research presented in this dissertation was aimed specifically at software for the electronic interlocking systems used in railway signalling, where assembler language programmes are often generated directly from their high-level specification. These assembler language programmes are then processed by an assembler to produce machine-code. It was to be shown that these two processes could be validated by the proposed method of reverse-translation. Automation of this validation method was also to be investigated.

A description is given of a guinea-pig microprocessor system, the process of tracing its operation and manipulation of the resultant traces to form a complete disassembly listing of the system's embedded software.

Methods of automatic control-flow and data-flow analysis of the disassembly listing are described and their operation is demonstrated by using sample portions of code. These analyses are done in preparation for translation of the disassembly listing into P-notation.

Final formatting of the programme to P-notation format is then described. This essentially involves control-flow formatting and variable-name insertion.

Finally, an analysis of the overall effectiveness of the proposed validation method in terms of the goal of the research is given. Conclusions drawn as a result of the research are presented.

CHAPTER 2 LITERATURE SURVEY

A survey was conducted of available publications on the subject of software validation, verification and testing. Most publications present generalised approaches to software and are, as such, not specifically relevant to the reliability of software within embedded systems. However, it is precisely this shortcoming which renders these publications relevant to the history of validation.

2.1 Survey of publications

By 1975 the poor reliability and high cost of large software systems was becoming a serious problem. Formal proof of programme correctness was thought to be infeasible or at least many years away and manual testing and code inspection of large systems were, in themselves, unreliable and costly.

Some automated analysis tools and software evaluation systems were in use at the time and Ramamoorthy and Ho [1975] described these as the most effective means of improving the reliability and reducing the cost of large software systems. Automated tools were capable of checking the presence of certain software attributes such as syntactic correctness, proper control structuring and module interfacing.

"Software evaluation systems" were defined as composite systems of automated tools for the purposes of system design analysis, debugging, testing and partial validation, that being the process of demonstrating the validity of a programme to an acceptable degree of reliability and performance.

Ramamoorthy and Ho also described the software evaluation systems in use at the time as only partially fulfilling their requirements in that they analysed the source code, but generally ignored the design and specifications.

Miller [1977] proposed a method of path-based testing and showed how a test coverage measure could be used as a measure of "how far the testing process has gone".

Testing a programme by running it on sets of test data had, until 1975, not been regarded as an effective validation method, since sets of test data were generated on an ad hoc basis by analysis of the internal structure of a programme only. Goodenough and Gerhart [1975] proposed a more rigorous method of test data selection. They proposed a "condition table" method of deriving test predicates. Test predicates describe what aspects of a programme are to be tested. Derivation was done by reference to the general requirement a programme was to satisfy, the programme's specification and the general characteristics of the implementation method used.

Admitting that exhaustive testing was rendered impossible by such time-considerations as human mortality, Goodenough and Gerhart hypothesised that the input domain of a programme could be partitioned into a finite number of equivalence classes such that a representative test for each class would, by induction, test the entire class. They did, however, point out that the fundamental problem of testing was the inference from the success of one set of test data that others would also succeed and that a problem with equivalence class testing was to show that the input domain partitioning was, in fact, appropriate. They suggested that their rigorous test case generation

method led to a better approximation of exhaustive testing and, used in conjunction with programme correctness proofs, significantly decreased the likelihood of programme failure.

While Goodenough and Gerhart were testing programmes by running them on sets of test data, Allen and Cocke [1976] were proving the integrity of data-relationships within a programme without execution of the programme. Their algorithmic approach used a control-flow graph representation of the programme and information about the data items used, to determine the data-flow relationships within the programme.

King [1976] was not convinced. He considered programme testing and programme proving as extreme alternatives and introduced the concept of symbolic execution, which he regarded as a practical approach between these two extremes. He developed EFFIGY, an interactive symbolic execution system for language statements in PL/I-style syntax. In EFFIGY, a user could define arbitrary identifiers to be symbolic programme inputs in place of specific integer constants and analyse programme behaviour by inspection of the resultant expressions generated by symbolic execution.

A further practical implementation of the concept of symbolic execution was provided by Clarke [1976]. She presented an interactive system for automatic test data generation to execute a specified path of an ANSI-Fortran programme and subsequent symbolic execution of that path. Her system also provided the facility for detection of nonexecutable programme paths.

Based on King's EFFIGY, the SELECT symbolic execution system devised by Boyer, Elspas and Levitt, and Clarke's ANSI-Fortran symbolic executor, Howden [1977]

developed the DISSECT symbolic testing system. The major advantage offered by the DISSECT system over previous systems was the command-file facility whereby a user could initially set up a series of executions to be performed, some conditional on others if desired, for any specified paths and with any combination of symbolic and real input values. As with Clarke's system, DISSECT was ANSI-Fortran specific.

In a case study of the effectiveness of various analysis and testing techniques, Howden [1978] applied the techniques to six sample programmes containing "naturally" occuring errors. He found that the use of symbolic testing resulted in an increase in reliability of 10-20 percent over testing on actual data. The increase was, however, reduced to 3-4 percent if "actual data" testing was augmented with other programme analysis and testing techniques such as special values and interface analysis. He showed that in most cases, one particular analysis or testing technique was more effective than the others in pinpointing a particular type of error and his over-riding conclusion was that no single programme analysis technique or programme testing strategy should be used to the exclusion of all others.

In the midst of the massive drive to automate the validation process, work was still being carried out on the development of more reliable manual validation techniques for use in environments where limited resources were available. The coupling effect, whereby most global errors such as failure to satisfy a particular specification are seen as being coupled to simple errors such as missing control paths, was exploited by De Millo, Lipton and Sayward [1978]. They based a series of "hints on manual test data selection" on the hypothesis that test data which distinguishes

all programmes differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors. Branstad, Cherniavsky and Adrion [1980] also proposed a streamlining and improvement of manual validation techniques throughout the development life cycle by testing, code reading and inspection, and independent reviews.

While referring to the various validation tools available at the time, Meyers [1979] too, proposed rigorous manual validation techniques and testing with carefully chosen test cases as being the most effective. His choice of test cases was based mainly on boundary-value analysis and cause-effect graphing.

The selection of test data had, for the most part, always been based on the internal control structure of a programme. Howden [1980a] proposed a "black box" approach to programme testing in which the internal structure of a programme was ignored during test data selection. Tests were constructed from the functional properties of the programme that were specified in the programme's requirements. The technique was known as functional testing, as opposed to structural testing.

Howden described the disadvantage of the black box approach as the fact that it ignored important functional properties of a programme which were part of its design or implementation and which were not described in the requirements. A case study involving a collection of scientific programmes led to the predictable conclusion that structural and functional testing were complementary rather than competing techniques.

Testing was still accepted as being more effective than formal programme proof in the demonstration of

programme correctness. Formal representation of specifications was viewed as so problematical as to be of little practical value. Formal proofs could not be used with the informal specification methods in use at the time. Kopetz described the specification methods in use as "verbal specification of software systems outside the areas of logic or numerical mathematics" (Kopetz [1979]).

Deutsch [1979] was sceptical about both testing and proving of programmes as effective means of increasing their reliability. Reduction of the complexity of programmes, he believed, would increase productivity, clarity, maintainability and modifiability.

Various papers were produced on the theory of test data selection for revealing particular types of error. Weyuker and Ostrand [1980] found Goodenough and Gerhart's [1975] theory of test data selection difficult to apply in the real world and proposed certain modifications to the theory whereby they set semicorrectness-proving as their goal. Proving semi-correctness meant demonstrating the absence of certain errors rather than the ideal proof of correctness, which meant demonstrating the absence of all errors.

White and Cohen [1980] developed a method of testing specifically to pinpoint control-flow errors and the conditions under which their method was reliable were carefully specified. Gustafson [1984] proposed testing for errors whose necessary input conditions were more likely to occur and for errors whose consequences were serious. His test case selection was based on what he called the "cost of errors".

A consolidation of software analysis and testing techniques as developed up to 1980 was provided by

Howden [1980b] when he applied various existing testing and analysis methods to a package of Fortran subroutines. He divided the methods used into two distinct categories: static analysis methods and dynamic testing methods. Static analysis methods referred to methods which were performed without actual execution of the code. Dynamic methods consisted essentially of testing and were performed automatically, except for the selection of test data. Test data was selected with a view to both structural and functional testing as previously described (Howden [1980a]). Static analysis methods consisted of automatic methods such as path flow analysis and statement analysis and manual methods which mostly involved checking consistency between subroutine headers and programme or requirements content. He concluded that the methods used could discover "a large majority" of errors in programmes of the type used. He found that testing (static and dynamic) and analysis methods were equally useful, each responsible for the discovery of about half the errors found. He indicated the need for extensible static analysis systems which allowed for the addition of further static analysis rules. He also stressed the importance of the development of a method to identify and test general and detailed design functions.

Carré [1980] described the principle methods of validating programmes as flow analysis (control-flow and data-flow analysis) and semantic analysis. He described a systematic manual method of control-flow analysis involving a methodical labelling technique to show such control flow anomalies as black holes and unused labels.

In data-flow analysis, Carré's detection of undefined variables and unused definitions was based on

algorithmic processing of sets of binary vectors representing variable-definitions within the programme. His method of semantic analysis was twofold. Assertions, derived from programme specifications, were inserted into the programme and manually processed using the programme logic and computation statements. A systematic technique was presented to prove that the truth of an assertion at any point followed from the truth of assertions at previous points in the programme. The other aspect of semantic analysis was symbolic execution. He was later to automate and present these techniques as a "validation package" [SPADE, 1985].

Because of the real need to validate large software systems, various validation packages or validation environments were developed after 1980. They mostly used existing techniques such as static code-parsing and dynamic testing, each implementing the techniques slightly differently in an automated package. Important amongst these were the STRUM system [Patterson, 1981] which concentrated on programmes for microcomputers and a system presented by Benson [1981] which introduced the concept of instrumentation of a programme with "executable assertions". Executable assertions are formal assertions made about the state of the programme variables at various points in the programme. The assertions are presented in such a way that the programme statements can be applied to them to show that execution of the programme would not violate any of the assertions.

The practice of translating a high-level source language into an intermediate language more suited to validation techniques had been instituted as early as 1975 (Ramamoorthy and Ho [1975]). No further development of the technique took place until it was

again used in the IVTS system [Taylor,1983].

The IVTS system (Integrated *Varification and Testing* Sytem) was designed specifically for use on HAL/S, a language used mostly in aerospace applications. Although IVTS *used standard* established validation techniques, its advantages over other integrated validation systems were a very sophisticated user *interface, making* application of any of the techniques simple for unqualified personnel, and the incorporation of an automatic "report writer" for *documentation* enhancement. The major advantage of the use of an intermediate language is that it renders the tools used non-source-language-*specific. All source* languages are translated to the same intermediate level and are thus able to be processed by the same tools. This feature *was exploited by Carré* in his validation package [SPADE, 1985].

Software fault tree analysis, the hypothesising of a particular fault occurring and subsequent "backtracking" through the *software to discover* all possible causes of the fault, was introduced by Taylor [1982] and used in a practical implementation by *Leveson and Harvey* [1983]. Taylor presented a proposed method of automating the procedure, but to date not much interest has been shown in the analysis method. *The lack* of interest has been due to difficulty in handling loops and the size of trees generated for most hypothesised faults. The method *can be, and sometimes* is, used for analysis of some highly-critical individual possible faults, but has little general *application*.

In the early ninteen-eighties, there was an *increasing* awareness that validation of programme code with respect to its requirements was only one facet of

validation in general. It was realised that design errors discovered as late as the coding stage were expensive to correct. Thus, validators began to realise that validation techniques had to be applied throughout the life-cycle of the software (Rzevski [1981] and Howden [1982]).

Validation techniques applied to software requirements and specifications were those related to checking consistency, completeness and correctness. Howden [1982] proposed the selection of test cases throughout the software life-cycle, including the requirements and specifications definition phases.

An in-depth survey and evaluation of the existing techniques of validation, verification and testing of computer software was conducted by Adrion, Branstad and Cherniavsky [1982]. To say the least, their conclusions were controversial. Because most validation and testing techniques were applicable to the testing of actual programmes and had little other relevance through the life-cycle of the software, they concluded that traditional, manual validation methods were most effective. Such methods included walk-throughs, reviews and inspections. Traditional manual methods could be used without massive capital expenditure and had uniform applicability throughout the software life-cycle, although they required a serious commitment and disciplined application. They also concluded that most existing automated validation techniques lacked a sound theoretical basis.

Thus it was that, where validation was of critical importance in the development of real systems, validation techniques used were still essentially manually orientated, involving massive human effort (eg. Short [1983]).

In a recent publication, Gerber [1985] described the techniques which were used to validate a large, real-time process control programme. Modules were individually validated by test cases and automatic integration and module-interface (inter-module data flow) analysis were performed. Functional analysis was manually performed by cross-referencing of the documentation. Timing analysis, too, was manually performed by doing a series of time-related calculations based on the programme code and showing that the programme would always operate within its specified timing constraints.

2.2 State-of-the-art

The previous section has given a general history of developments in the field of software validation since 1975, by reference to, and résumés of, significant publications. During the period from the mid- to late-nineteen-seventies, the emphasis was on developing the ability to analyse and test high-level language programmes which performed arithmetic and logic functions. Much mental effort and practical trial went into this development and a sound basis for further development was created. Noteworthy were papers by Ramamoorthy and Ho [1975], Goodenough and Gerhart [1975], King [1976], and Howden [1978]. The original motivation for the development of this analytical and testing ability was a so-called "software crisis" brought about by the low reliability and consequent high cost of software. The objective was to improve software reliability to a level where the cost of its generation and implementation were acceptable.

Within the context of the above motivation, it can be argued that early work in the field of software

analysis and testing was extremely successful.

Semi-correctness (correctness up to an acceptable level of reliability) of programmes was achieved both by validation techniques and by *design-for-validation* techniques. Even to this day, programmers writing computational programmes can use established validation techniques to improve the quality *and* reliability of the software they produce.

After the late nineteen-seventies, however, a branch of computer usage which had been slowly developing for some years, very quickly became an important aspect of computing technology and usage. *The microprocessor* was to be used in real-time process control applications involving the risk of the loss of many human lives, such as in nuclear-reactor monitoring and transport-system control. Validation needs changed; the science of validation did not.

The goal of validation changed from "partial validation for increased reliability" to "complete validation for complete reliability". Not only did the *goal change* - applications of software also changed. Real-time applications involve aspects that were not previously considered in validation philosophies *such as stringent* timing constraints and the cyclical nature of real-time programmes. Emphasis changed from arithmetic/ computational high-level-language *programmes* to programmes involving many and varied I/O routines and bit-manipulation, often written in low-level languages.

A summary of techniques available and in use for the validation and verification of real-time software was provided by Quirk [1985]. *His book contains* a comprehensive bibliography of relevant publications.

A significant contribution to the assessment of the state-of-the-art of software validation and verification was made by the third Verification Workshop (VERkshop III [1985]), held in California.

The principal goal of VERkshop III was to review verification technology and, in particular, to identify what was being used in practice and what specific areas required additional research. The attendees included researchers who were active in the development of verification systems, theorem proving, formal language semantics and applying current verification techniques to production problems.

Although four years had passed since the previous Verification Workshop (VERkshop II), there was a consensus of opinion that only incremental progress had been made in the area of programme verification. It was agreed, however, that using existing methods and technology, significant progress had been made in the development of integrated verification systems, although the systems were still usable only by highly skilled individuals and were not in a position to be used on a production basis.

A stagnation in the abilities of validation techniques has been brought about by the application of analysis techniques to software which has different validation requirements from those ich motivated the design of the analysis techniques. All developments in validation since 1980 have essentially been refinements of the original techniques developed by validation pioneers such as Ramamoorthy and Ho [1975] and King [1976]. The fundamentals of validation as a science must be re-addressed in order to develop new, more applicable validation techniques to meet new validation demands.

Thus it was felt appropriate to undertake an exploratory project to show the validity of the validation procedure proposed in the previous chapter. *The proposed* validation procedure would meet the validation demands of a computer-based railway signalling system and *many other* applications of computer-based control systems.

CHAPTER 3 DISASSEMBLY

The validation procedure proposed in chapter one of this dissertation consists of analysing a machine-code programme executed by a microprocessor and translating the programme up to the level of its high-level specification. The translated programme is then compared with its original specification.

In this approach, no assumptions are made about the correctness of assembly-language listings supplied by a manufacturer or programmer. The machine-code programme is obtained directly from the microprocessor itself. Actual operations executed by the microprocessor, as a result of instructions fetched from PROM, are used to reconstruct the machine-code programme. Validation is thus ensured from the lowest possible level - that of the effect of the software on the microprocessor chip.

The operations of a microprocessor are traced by a logic analyser whilst external stimulus is given to the microprocessor system to force the software to traverse every one of its possible paths. All traces thus obtained are combined and edited to form a complete disassembly listing of the programme. This disassembly listing is then subjected to subsequent analysis and translation procedures.

The method of obtaining a disassembly listing of a machine-code programme was demonstrated by practical trial. The test set-up used and experimental procedure followed are described in this chapter.

3.1 Test Set-Up

3.1.1 *Guinea-pig microprocessor system*

A method of producing a disassembly listing from a microprocessor-based system has been proposed. To test the proposed method, a guinea-pig microprocessor-based system was required. One such system, the "TEM L 30 Block Instrument Controller" (TEM L 30 Block Instrument Control Unit, A Technical Description (1983]), was available in the Signalling Department of the South African Transport Services, where the research was undertaken. This system was selected so as to provide a test-bed which was a true representation of the application area being addressed in this current research investigation.

In railway signalling, a "block instrument" is an electro-mechanical device which is used to send current of forward or reverse polarity down a pair of wires to another block instrument. It also indicates the presence and polarity of any current it receives from any other block instrument. A more detailed description of a signalling block instrument can be found in Appendix A.

The "TEM L 30 Block Instrument Controller" is a microprocessor-based control-unit designed to facilitate the operation of block instruments over radio links, instead of over wire-pairs. The task performed by the TEM L 30 controller is the bi-directional, fail-safe transmission of information, between two block instruments, over a radio link. A more detailed description of the TEM L 30 controller can be found in Appendix A.

3.1.2 Tracing the microprocessor operations

Tracing was done with the State-6 Analyser option of an HP64000 measurement system (HP64000 Logic Development System, System Overview [1982]). The system offers instant disassembly, mass storage of traces on its associated disc, sophisticated triggering and storage specification facilities, printing of traces for documentation purposes and uploading of traces to a minicomputer for manipulation.

3.1.3 Stimulus of the device

The TEM L 30 was required to traverse every possible path of its programme while its operations were traced by a logic analyser. In order to ensure that all possible paths had been traversed, the contents of the programme-PROM were listed. A check was made to ensure that there were no programmed PROM locations which did not appear in any of the logic analyser traces.

Certain paths of the TEM L 30 software are not traversed during normal power-up, quiescent operation or power-down. They are the paths associated with the operation of one or both of the block instruments causing message-transmission, hardware-monitoring with associated failure strategies and message-reception with subsequent output to the block instruments. In order to cause the TEM L 30 controller to traverse these paths of its programme, it was necessary to provide external stimulus to the TEM L 30.

Operation of the block instruments

Block instrument simulation was realised by means of

two block instrument simulators supplied with the controller for testing purposes. The simulators simply provide the ability to source current of either polarity to the controller by manual pushbutton operation.

Hardware-monitoring and failure strategies

The actual monitoring of the condition of the hardware is a routine operation when the controller is in a powered-up state. It was, however, necessary to simulate hardware failures to force the software along the paths of its failure-strategies. Such hardware failures were easily simulated by false feeds, component removal, supply voltage adjustment, etc.

Message reception and analysis

A radio simulator was provided with the controller for testing purposes. The simulator simply injects noise into a physical connection between the modem cards of two TEM L 30 controllers. This creates the effect of the operation of the modems over a radio link. Another controller could, therefore, have been used to send a valid message via the radio simulator to the controller under test.

The normal, operative interaction of the two controllers, however, forms a small part of their message-handling routines. To force the TEM L 30 controller to traverse all possible programme-paths related to message-analysis, it was necessary to transmit to the controller under test messages with incorrect parity, faulty Manchester II coding, less than three messages in agreement, etc. The easy

manipulation of messages to be sent was thus an important criterion. A programmable microprocessor-based data acquisition and transmission system was used for this purpose. This microprocessor-based system was developed by engineers of the South African Transport Services and is known as a Remote Data Unit (RDU).

The modem card of a second controller was used to interface, via the simulated radio link, to the controller under test. Control of the modem card, interactive message-compilation and message-transmission were realised with the RDU.

Programmes were written on the RDU to perform the various message-generation tasks and were executed by the RDU as compiled Basic programmes.

The complete test set-up which was used to obtain the traces is shown in fig. 3.1 overleaf.

3.2 *Trace-Specification*

Before any of the traces were executed, a trace-specification document was produced. This document showed the address trigger point to be set up on the HP64000, the section of code to be stored and the stimulus to be applied to the TEM L 30 for each trace to be executed and recorded. The trace-specification document was produced by reference to the TEM L 30 manufacturer's software listing, circuit diagrams and description of operation. Any errors or omissions in these manufacturer's documents would have become apparent when actual execution of the traces was attempted. A sample portion of the complete trace specification document is shown in figure 3.2.

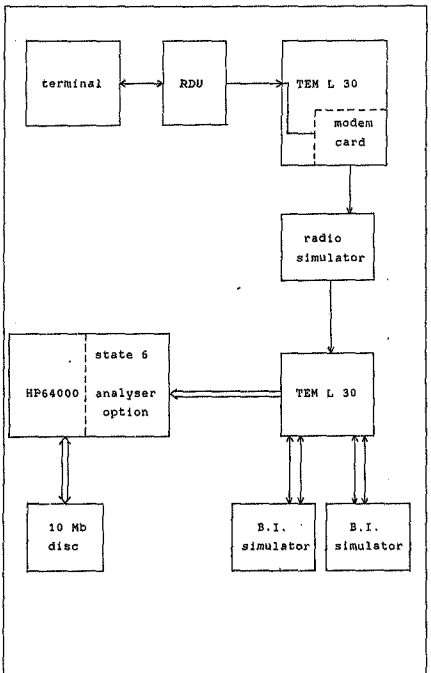


Fig. 3.1 Test set-up

TCML36 SOFTWARE TRACE SPECIFICATIONS : TS10 : 6.contd "second half bit" thru "addressed to LP01"

Trace no.	Trig. pnt ADDR. Hex	Storage Start ADDR. Hex	Storage End ADDR. Hex	Occur.	Stimulus to test rig
167	C97F	C009	C90A	2	Send valid message to TCML36
168	C982	C970	CCF0	1	Send message with any M2 failure
169	C989	C004	C00D	1	trace 168 - M2 failure in fixed part
170	C991	C0E1	C00D	1	ref. trace 169
171	C9A0	C997	C9CA	1	trace 168 - M2 failure in variable part - LP23
172	C99D	C910	C9A3	1	trace 168 - M2 failure in variable part - LP01
173	C9CC	C9CA	C009	1	ref. trace 172
174	C9D1	C0E1	C08D	1	ref. trace 172
175	C9D4	C9CA	CC3C	1	ref. trace 172
176	C9D7	C004	C80D	1	ref. trace 172
177	C9DE	C97F	C009	1	ref. trace 167
178	C9E6	C0E1	C80D	1	ref. trace 167
179	C9EF	C9DC	C009	1	ref. trace 167
180	C9F7	C0E1	C80D	1	ref. trace 167
181	C9FA	C9ED	CA7F	1	ref. trace 167 - to LP23
182	C9FE	C9FC	C009	1	Send message with incorrect parity
183	CA09	C0E1	C80D	1	Send message with incorrect parity
184	CA30	CA17	CA40	2	ref. trace 167 - to LP01
185	CA1F	CA10	CC3C	1	Send mess. to LP23 then to LP01
186	CA22	C004	C00D	1	Send mess. to LP23 then to LP01
187	CA37	CA35	C009	1	Send another message during decode - LP01
188	CA3F	C0E1	C00D	1	Send another message during decode - LP01
189	CA4F	CA40	CC3C	1	Send 1st 2 mess. with different polarity
190	CA52	C004	C00D	1	ref. trace 189
191	CA5A	CA4D	C009	1	ref. trace 167 - LP01
192	CA7C	C0E1	C00D	1	ref. trace 167 - LP01
193	CA7F	CA7F	CA89	1	ref. trace 167 - to LP23

Fig. 3.2 Portion of trace specification document

3.2.1 Trace-Specification Document

Trigger-occurrence

It will be seen that the *trace-specification* document (*figure* 3.2) contains a column headed "occur.". This specifies the occurrence of the trigger-point address on which the analyser must trigger: first or second. Triggering on the second occurrence was specified in order to trace through a section of code ended by a branch instruction back to a point within that section. This was preferable to triggering on the branch instruction itself, since it showed that the code was actually taking that particular branch of the decision.

Trigger-on-opcode

What is not apparent from the trace-specification document (figure 3.2) is that triggering was not initiated on the occurrence of a particular address alone. The TEM L 30 does regular inter-processor PROM comparisons and so a particular address may appear on the address bus for the purpose of a data-read from the PROM. Triggering was desired only in the case of the correct address being present and an opcode being fetched. This was specified for all triggering conditions.

3.3 Production of the Disassembly Listing

Production of the disassembly listing consisted of two phases.

Firstly, each of the test runs specified on the trace specification document was executed. All the trace listings obtained were stored as files on the HP64000 measurement system's local disc.

Secondly, all files thus obtained were uploaded to an HP1000 minicomputer, where they were edited and sorted by absolute address to form the complete disassembly listing.

3.3.1 Obtaining the Traces

Providing the stimulus

The stimulus to be applied to the TEM L 30 controller to obtain each trace was determined from the trace-specification document. Some stimuli consisted of

initial conditions to be set up (eg. removal of a fuse), while others were actions to be taken during operation of the device (eg. operation of one block instrument during servicing of the other).

Tracing the test runs

Triggering and storage conditions for each test run were set-up on the measurement system. Once the trigger had been enabled and the measurement system was waiting to trigger, the TEM L 30 controller was powered-up as specified in the trace-specification document. On completion of storage, the trace obtained was stored on the measurement system's disc.

3.3.2 Manipulation of the Traces

Uploading the files to a minicomputer

All the traces which were stored as files on the measurement system's disc were uploaded to an HP1000 minicomputer for editing, sorting and analysis.

Since all files were to be concatenated for sorting and analysis, the large virtual-RAM capacity of the minicomputer was required for this purpose.

Editing and sorting the traces

Pascal programmes were written and run on the HP1000 minicomputer to edit and sort the traces into a complete disassembly listing. A sample trace-file is shown in figure 3.3.

```
64620 State Analyzer                    Mon, 10 Jun 1985, 11:38

                Trace List                    State 6, 60 channel, 6800/02 interface
Label:  ADDRESS      6800/6802 Mnemonic
Base:       hex                  hex
Map:    ADDR_MAP     ADDR_MAP
~002    abs  CC25 BNE    CC2A
~001    abs  CC27    53 operand fetch
trigger abs  CC28 LDAA 2002
*001    abs  CC2B    20 operand fetch
*002    abs  CC2C    02 operand fetch
*003    abs  2002    10 memory read
*004    abs  CC2D BITA #02
*005    abs  CC2E    02 operand fetch
*006    abs  CC2F BNE  CC34
*007    abs  CC30    03 operand fetch
*008    abs  CC31 ANDA #F9
*009    abs  CC32    F9 operand fetch
*010    abs  CC33 STAA 1002
*011    abs  CC35    10 operand fetch
*012    abs  CC36    02 operand fetch
*013    abs  1002    C0 memory write
*014    abs  CC39 PULA
*015    abs  007C    09 stack read
*016    abs  CC3A PULB
*017    abs  007D    90 stack read
*018    abs  CC3B RTS
*019    abs  007E    C1 stack read
*020    abs  007F    2F stack read
```

Fig. 3.3 Sample trace file

```
AC14 LDAB F003
AC17 EORB #F007
AC1A STAB F003
AC19 STAA X
AC1A CMPA #F00B
AC1B BGT  AC20
AC1C LDAA F001
AC1D ORAA F009
AC1E STAA F009
AC1F BRA  AC24
AC20 LDAA F001
AC21 COMA
AC22 ANDA F009
AC23 STAA F009
AC24 LDAA F000
AC25 LDAB #F00A
AC26 CMPA F001
AC27 BEQ  AC2E
AC28 BLT  AC2A
AC29 LSRB
AC2A LDAA F002
AC2B ANDA #F00B
AC2C ABA
AC2D STAA F002
AC2E BRA  AC01
```

Fig. 3.4 Sample disassembly listing

These trace-files were first individually edited to remove the unnecessary text at the head, triggering information and data-reads and writes. All the resultant edited trace-files were then concatenated to form a single file. All duplicated statements were removed and the file was sorted by absolute programme address to form a complete disassembly listing. A sample fr the resultant listing is shown in figure 3.4.

3.4 Result of disassembly

A method of deriving a complete disassembly listing of a programme executed by a microprocessor has been described.

The method essentially involves tracing the operations of the microprocessor with a logic analyser while the microprocessor system is externally stimulated to execute every path of the machine-code programme. Resultant traces are manipulated to form the complete disassembly listing.

The actual disassembly of machine-code instructions is performed by the logic analyser. If required, the integrity of the disassembler can be demonstrated by re-assembling the resultant disassembly listing and comparing the result with the original machine-code.

The analysis and translation techniques described in further chapters are aimed at demonstrating the consistency between a machine-code programme and its P-notation specification. Since the machine-code programme of the TEM L 30 was not written from a P-notation specification, the programme will not be subjected to such analysis and translation.

Assembler listings of the same format as the TEM L 30 disassembly listing which were written from *P-notation* specifications will be used to demonstrate these techniques. It will, however, be shown how the techniques developed can greatly assist the readability and understandability of a disassembly listing such as that obtained from the TEM L 30.

Further chapters describe how a disassembly listing can be analysed and translated to the level of its P-notation specification. This process begins with control-flow analysis, described in the following chapter.

CHAPTER 4 CONTROL-FLOW ANALYSIS

The previous chapter described a method of *deriving* a complete disassembly listing of a machine-code programme. In order to translate the listing thus obtained into P-notation, it *was necessary* to analyse the listing in terms of control and data-flow. A method of analysing the control-flow of a disassembly *listing in terms* of standard P-notation constructs is presented in this chapter.

4.1 Constructs in P-notation

P-notation supports constructs in the three broad categories of **sequence**, **selection** and **iteration**.

4.1.1 Sequence

Sequence refers to the top-down sequential execution of programme statements. If a statement *does not* explicitly transfer control to some other part of the programme, then the statement below it is the one which will be executed *next*.

4.1.2 Selection

P-notation supports two types of **selection** *construct*: the **case** statement and the **if** statement. Definitions of these statements can be found in Appendix B.

4.1.3 Iteration

P-notation supports three iterative constructs: the

repeat statement, the **while** statement and the **for** statement. Definitions of these statements can be found in Appendix B.

4.2 Construct Recognition and Labelling

4.2.1 Input and storage of *disassembly listing*

The control-flow analysis programme provides the facility for the *input* of any user-specified file containing a disassembly listing of the form obtained from the TEM L 30 (Chapter 3). The listing is stored in a record structure in RAM. Fields of each record contain absolute programme address, opcode, and where applicable, operand.

4.2.2 *Processor-specific information*

A data file containing information specific to the Motorola 6802 microprocessor is referenced by the analysis programme. The data file contains information about whether a particular opcode is a conditional branch statement, an unconditional branch statement or neither. For each statement read from the disassembly listing file, *branch information* is read from the data file and added to the record of that particular statement.

Together with the absolute addresses in the disassembly listing, the branch *information from* the data file is sufficient to facilitate automatic control-flow analysis of the disassembly listing.

4.2.3 First pass: **if-then-else** and **loop** recognition

Recognised constructs are numbered sequentially in order of *recognition*. An *internal* labelling system is used in labelling recognised constructs. The various elements of a construct are labelled with a *character-string* indicating their significance.

So if recognised construct number # is an **if-then-else** construct, the statement beginning the **if** portion is *labelled* **if__#**, the statement beginning the **else** portion is labelled **else#** and the end of the construct is labelled **comp#**. In the *simplified case* of an **if-then** construct, the **else#** is omitted.

If recognised construct number # is a **while do** loop construct, then the beginning and end of the loop are *labelled* **whil#** *and* **endw#** respectively. In the case of a **repeat until** loop, the beginning and end of the loop are labelled **rept#** and **until#** *respectively*.

Distinguishing between a **for** loop and a **while do** loop is difficult and in any case not always possible. **For** loops are recognised as **while do** loops. Standard coding of **for** loops for the purpose of recognition could be added as a refinement of the loop recognition process.

If a branch and its destination are not *recogniseable* in the context of any standard construct, then they are labelled **unkn#** and **endu#** respectively.
Using the abbreviations UCB for unconditional branch and CB for conditional branch, generalised *implementations* of the P-notation standard constructs are given in figure 4.1 overleaf.

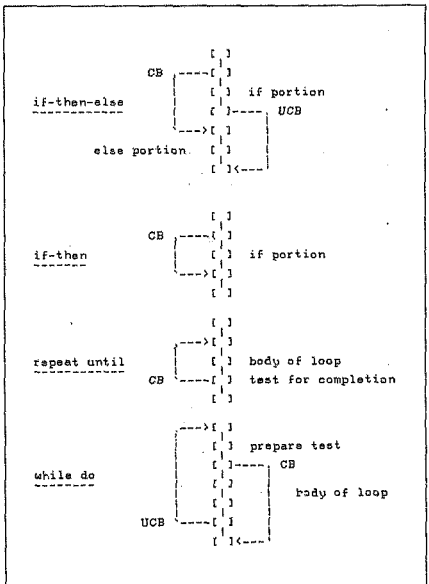


Fig. 4.1 Generalised implementations of P-notation *standard constructs*

The algorithm used to recognise the *above constructs* and label the relevant statements accordingly is shown in Appendix C. The algorithm was implemented as a Pascal programme for automatic *construct recognition*. A sample printout after the first pass of the analyser is shown in figure 4.2 below.

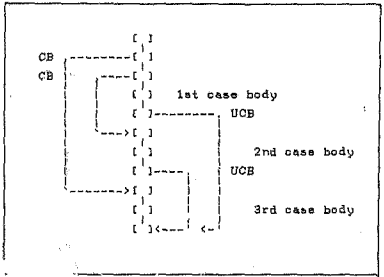
```
            C1AE LDAA 00,X
if__1       C1B0 BEQ  C1BC
            C1B2 LDAA 135C
            C1B5 LDX  #C0
proc        C1B8 JSR  04,X
            C1BA BRA  C1D1
            C1BC CPX  #0000
if__2       C1BF BNE  C1C5
            C1C1 CLR  0003
            C1C4 LDAA #01
comp2       C1C6 STAA 0003
            C1C8 LDAA #01
            C1CA STAA 0003
            C1CC STAA 1002
comp1       C1D1 NOP
```

Fig. 4.2 Printout after pass one of the analyser

4.2.4 Second pass: **case** recognition

If recognised *construct* number '#' is a **case** construct, then the statement beginning the construct will be labelled **case#**. The statement beginning each of the separate *cases within the construct* will be labelled **of__#**.

An implementation of the P-notation **case** construct could take the form shown in figure 4.3 overleaf.



.3 An implementation of the P-notation **case** construct

Branching need not have occurred in the order shown. The first conditional branch could have branched to the second **case** body and the *second* conditional branch to the third **case** body, or any other order.

Whatever the order of branching, the whole *construct* would have been analysed in the first pass of the analyser as a series of overlapping **if-then-else** constructs. This fact is exploited in the recognition of **case** constructs by the *second pass* of the analyser.

The analyser does a second pass of the listing, searching for statements which were marked in the first pass as multiple **comp** *statements*. All such statements could form the end of **case** constructs. The algorithm shown in Appendix C is used to determine whether a particular multiple **comp** statement *does in fact form*

the end of a **case** construct. The algorithm also marks the relevant statements accordingly if a **case** construct is recognised.

The number of a recognised **case** construct is the number of the lowest numbered overlapping **if-then-else** construct forming part of the **case** construct.

The algorithm shown in Appendix C was implemented as a Pascal programme. This programme was used to perform automatic case construct recognition.

Figure 4.4 shows a printout after pass two of the analyser. The portion of code which was analysed is seen to contain an **if-then** construct (construct number 2) nested within the first **case** of a *case construct* (construct number 1).

```
                C1C0 LDAA  00,X
                C1C2 CMPA  #01
case1           C1C4 BEQ   C1D0
                C1C6 CMPA  #02
                C1C8 BEQ   C1DB
                C1CA CMPA  #03
                C1CC BEQ   C1DD
                C1CE BRA   C1E3
of__1           C1D0 LDAA  2004
if__2           C1D3 BNE   C1D6
                C1D5 DECA
comp2           C1D6 BRA   C1E3
of__1           C1D8 INC   4140
                C1DB BRA   C1E3
of__1           C1DD DEC   4ACF
                C1E0 DEC   4ACF
comp1           C1E3 NOP
                C1E4 NOP
comp1           C1E3 NOP
```

Fig. 4.4 Printout after pass two of the analyser

4.2.5 Overlapping and unrecogniseable constructs

Programmes which have been incorrectly coded from their P-notation specifications or which were not written from P-notation specifications, will often have unrecogniseable structure. Such programmes may contain branch statements which do not form part of standard P-notation constructs.

Once a particular construct has been recognised by the analyser, inadvertent branching into or out of that construct is disallowed. Such a branch statement is not recogniseable in terms of that construct.

Branches which are not recogniseable in terms of standard P-notation constructs are marked as such by the first pass of the analyser.

Automatic detection of overlapping constructs is also possible. A third pass of the analysed listing is performed, checking for overlapping constructs. For each loop and each section of an **if-then-else** or **case** construct, a check is made that all constructs nested within the loop or section are complete. This is done by checking that within the loop or construct section:

each whil# has a corresponding endw# and vice versa
each rept# has a corresponding untl# and vice versa
each if__# has a corresponding comp#
each case# has a corresponding comp#
each comp# has a corresponding if__# or case#
each else# has a corresponding if__#
each of__# has a corresponding case#

Violations of these completeness criteria are flagged by being printed during this third pass of the analyser.

Figure 4.5 shows a sample analyser output *together* with its corresponding overlapping construct printout. The **else** portion of an **if-then-else** construct is seen to overlap the end of a **repeat until** *loop*.

```
rept2   2000 LDAA C000
        2003 ANDA #FE
        2005 STAA C000
        2008 ANDA #01
if__1   200A BNE  2015
        200C LDAB C001
        200F ABA
        2010 STAA C001
        2013 BRA  2022
else1   2015 LDAB C008
        2018 ABA
until2  2019 BNE  2000
        201B STAA C008
        201E CLRA
        201F STAA C004
compl   2022 NOP

Overlapping constructs:

NO   compl   FOR   if__1   IN   rept until   NUMBER   2
NO   rept2   FOR   until2  IN   else         NUMBER   1
```

Fig. 4.5 Overlapping construct *detection*

Overlapping constructs are not *representable* in P-notation. For translation of disassembly listings containing overlapping constructs into P-notation, it is necessary to *modify* the *original* machine-code programmes to contain no overlapping constructs.

When a printout *from the analyser* shows no unrecogniseable branches and no overlapping constructs in a disassembly listing, then the structure of the listing is sound and it may be translated into P-notation format.

4.3 Results of Control-Flow Analysis

The control-flow analyser described in this chapter

analyses a disassembly listing of the form *obtained* in the previous chapter. Analysis is in terms of standard P-notation constructs.

The analyser indicates branch-relevant statements which are *not part of standard* P-notation constructs. Overlapping constructs are also detected and indicated. For the analyser to fit all *branch-relevant statements* into standard, non-overlapping P-notation constructs is a necessary and sufficient condition for the translation of the control-flow framework of the disassembly listing into P-notation.

The control-flow analyser is no more than its name implies. It *analyses only* the control-flow possibilities within a disassembly listing. Test predicates which determine along which path execution of a programme will occur at run-time are ignored.

Once the control-flow framework of the disassembly listing has been analysed, only sequential portions of code remain to be *analysed*. The next chapter describes how the remaining sequential code portions are analysed in terms of data-flow. It also describes the *analysis* of test predicates, where possible, for insertion into the final P-notation control-flow framework.

The following chapter describes how information from both the *control-flow* analyser and the data-flow analyser can be used to translate a disassembly listing into P-notation.

CHAPTER 5 DATA-FLOW ANALYSIS

The previous chapter described a method of analysing the control-flow framework of a disassembly listing. Once this control-flow framework has been extracted from the disassembly listing, only sequential portions of code remain.

To enable translation of a disassembly listing into P-notation, the remaining sequential portions of code must be analysed in terms of their memory-location manipulation. Test predicates (data-preparations for conditional branch instructions) must also be analysed for insertion into the control-flow framework. Analysis of code in terms of its memory-location manipulation and test predicates is referred to here as "Data-Flow-Analysis".

This chapter presents a method of automatic data-flow analysis. The abilities and limitations of this method are demonstrated by applying it to portions of assembler code. These portions of code are of the same format as the TEM L 30 disassembly listing obtained as described in Chapter 3.

5.1 Data-Types in P-notation

P-notation in its original form (Young[1980]) supports only two predefined simple data-types called **8bit** and **16bit**. All other simple types must be user-defined in terms of these two predefined types. The type indicates the size of the data object, thus the minimum size of a data object is eight bits.

The predefined structured type **record**, however, may contain entries of type less than eight bits. Even in

this case, however, the type declaration declares only the size of a data object. No facility is available in P-notation for specifying the position of a data object of less than eight bits within an eight bit word. Assignation of absolute memory addresses to bytes is also not possible in P-notation. Having only two predefined simple types was also found to be a shortcoming of standard P-notation. P-notation, as it is used in the Signalling Department of the South African Transport Services, where this research was conducted, has been modified to overcome these shortcomings.

Modified P-notation has three predefined data-types which are shown below, together with their memory requirements.

integer : 8 bits (signed 2's complement)
po'nter : 16 bits
Boolean : 1 bit

The above list of predefined types could be expanded to suit a particular application.

A facility for positioning a data object of less than eight bits within an eight bit word has also been added to P-notation. The eight bit word is declared as a **record** and the positions of its entries are indicated by binary values, as in the example of figure 5.1 overleaf.

```
storbyte = record
  flag1 : Boolean (%0000 0001);
  flag2 : Boolean (%0000 0010);
  flag3 : Boolean (%1000 0000);
end;
```

Fig. 5.1 Data object positioning within a record

Here storbyte has been declared as a byte containing packed bits flag1, flag2 and flag3 in the positions indicated.

Another addition to P-notation is the facility to give a data object an absolute addresss in memory. This is necessary, for example, when hardware is designed before its embedded software is designed. In such cases, address decoding predetermines the addresses of memory and I/O devices.

For example, to read from or write to a single line of an I/O port, the port is declared as a **record** containing a Boolean variable in the position of the I/O line. The record is then assigned an absolute address in memory.

If the I/O line of the above example is in bit position zero of an I/O port at absolute address 1000H, then the P-notation data declaration would be as shown in figure 5.2.

In this case, outport has been declared as a variable at absolute address 1000H containing a single I/O line called 'xmit' in bit position zero. In effect, this amounts to a declaration of the absolute address of Boolean variable 'xmit'.

```
type
  r = record
    xmit : Boolean (%0000 0001);
  end;
var
  outport:r absolute:1000H;
```

Fig. 5.2 Absolute address declaration

Further references to P-notation must be taken to imply *modified* P-notation, that is, P-notation with the above additions.

5.1.1 Formulation of a data-table

A disassembly listing of a programme references variables only by their absolute addresses in memory. If such a disassembly listing is to be translated to the level of and compared with its P-notation specification, the original variable names from the specification would have to be added to the listing. An automatic method of translating a disassembly listing into P-notation would thus require information regarding the correlation between variable names in the P-notation specification and absolute addresses in the disassembly listing.

If the data declaration portion of the P-notation specification contains absolute address and optional bit-within-byte position declarations, then these declarations give direct correlation between variable names and their absolute addresses. For variables not declared at absolute addresses, variable name/absolute

address correlation is determined by the way in which assembler language code is written from the P-notation specification.

By inspection of the data declaration portion of the P-notation specification and the declaration/equate portion of the assembler language code, all variable name/absolute address correlations can be determined. Together with information about the **types** of the variables, these correlations are presented in a fixed-format tabular fashion. An example of such a table is shown in figure 5.3 below.

```
[0000] lines[1]    integer
[0001] lines[2]    integer
[0002] storbyte    record
  0       flag1      Boolean
  1       flag2      Boolean
  2       flag3      Boolean
[1000] outport     record
  0       xmit       Boolean
```

Fig. 5.3 Standard format of data-table

A data-table such as the one shown above includes data-type and absolute address information for all the variables appearing in the data declaration portion of a P-notation specification.

If a byte consists of packed variables of less than eight bits, then the bit positions of such variables within the byte are indicated below the absolute address of the byte.

Actual insertion of the variable names into the

disassembly listing is performed by an automatic programme translator, described in the next *chapter*. Data-flow analysis, as described in this chapter, involves the derivation of expressions representing data manipulations within the disassembly listing and verifying their type-consistency. Thus, once the data-flow analyser indicates no type-inconsistencies within expressions, the programme translator can simply insert variable names in place of absolute addresses, according to the variable name/*absolute address* correlation table.

5.2 Effect of Data-Type on Data Manipulations

Data objects, depending of their declared type, are either whole bytes (eg. **integer**), combinations of bytes (eg. **pointer**) or portions of a byte (eg. **Boolean**). So the nature of manipulations which are performed on a particular byte of memory depends on the data-type of the byte, or of data objects within the byte.

In the case of **integer** and **pointer** variables, only whole-byte manipulations may be performed. A bit or bits within a byte may not be selectively manipulated. Typical whole-byte manipulations would be to clear a byte, to add a value to a byte, to decrement a byte etc. Such manipulations are clearly of an arithmetic nature.

In the case of Boolean variables, arithmetic-type manipulations of bytes containing such variables constitute data-type violations. Boolean variable manipulation consists of logical operations on individual bits within bytes. Such manipulations make use of the operations of loading, masking, shifting,

operating (logically) and storing of bytes.

So the type of manipulation performed on a particular byte of memory depends on the data-type of the variable of which the byte forms a part, or which forms part of the byte. This fact is exploited in the development of a method of automatic type-consistency checking during data-flow analysis.

5.3 Analysis of Data Manipulations

As stated earlier, two distinct types of manipulation are used to manipulate data objects of P-notation predefined type. These are bit-wise, logical manipulation and whole-byte, arithmetic manipulation. The manipulation method used depends on the type of the data object being manipulated.

5.3.1 General strategy

Data-flow analysis consists of two stages. Firstly, expressions representing data manipulations are generated and then type-consistency within such expressions is confirmed. The two processes work hand-in-hand. An expression containing a type-consistency violation will not be printed - the appropriate section of code will be flagged as containing illegal operations.

In cases where disassembly listings do not have corresponding P-notation specifications, no data-type table exists. Partial data-flow analysis can still be applied to such listings to aid manual analysis of the programme. In such cases, type-consistency checking is disabled and expressions representing data

manipulations are generated in all cases of data object manipulation.

The general strategy of the analyser is to parse the disassembly listing (including information from the control-flow analyser) from beginning to end, searching for conditional branch statements and statements which affect the contents of memory-locations.

If a statement affects a memory-location 'immediately' (independently of any other statement), as in the case of memory-location clear, memory-location increment etc., then an expression of the operation is derived as described in 5.3.2 below.

If a register storage statement is encountered (effect on memory-location dependent on contents of register) then the analyser works backwards through the code, generating an expression as described in 5.3.3 below.

In the case of a conditional branch statement, the analyser again works backwards through the code, this time generating an expression representing the condition under which branching will occur, as described in 5.3.4 below.

For manual inspection of type-inconsistencies and for analysis of code where no P-notation specification exists, an intermediate data-flow analysis result may be produced. This result consists of a printout of the listing, together with generated expressions inserted in the appropriate places within the listing.

All expressions representing data manipulations are stored in a text file, together with the addresses of the statements which caused their generation. This file is then referenced by the programme translator as

described in Chapter 6.

5.3.2 Immediate manipulation

Certain instructions in the Motorola 6802 instruction-set operate directly on memory-locations and are independent of the contents of the processor's registers. Some examples are shown in figure 5.4.

```
LSR 1000 -  Do a logical shift right of data in
            memory-location 1000H
CLR  000F - Clear memory-location 000FH
DEC  C126 - Decrement data in memory-location C126H
```

Fig. 5.4 Immediate memory-location manipulation

With the exception of the CLR (clear) statement, all such immediate statements have an implicit arithmetic or logical connotation. Thus the analyser, before generating an expression representing the operation, checks that the variable being manipulated is of the appropriate type. If not, it generates a type-inconsistency message.

In the case of a packed Boolean type, a string of expressions is generated, showing the effect of the operation on each of the Boolean variables in the byte. This is achieved by reference to the data-table.

In the case of the CLR statement, the analyser cannot check for type-inconsistency, since the operation is legal for all data-types. So the analyser uses information from the data-table to determine what type of expression should be generated, bit-wise or whole-

byte. Examples are shown in figure 5.5.

disassembly listing statement	data-table entry	analyser output
CLR 1000	[1000] integer	[1000]:=0
CLR F000	[F000] record	
	0 Boolean	[F000]0:=0
	4 Boolean	[F000]4:=0

Fig. 5.5 CLR statement type-determination

Intentional misuse by programmers of the natural connotation of immediate instruction opcodes will prevent analysis of the code because of type-inconsistencies. Typical of such misuse is the incrementation of a record which is known to contain a Boolean variable in bit position zero, in order to complement that Boolean variable.

5.3.3 Register storage

If a store-register instruction is encountered, the analyser has to work backwards from the instruction to determine what the contents of the register would have been at the time of the store operation. However, register contents are not always completely determineable. Where a register emerges from a previous construct to be manipulated and stored before being redefined, its contents are not completely determineable. The origin, and hence contents, of the register are unknown at the time of emergence from a previous construct.

The analyser is able to generate a complete expression representing the effect of a store instruction when all registers affecting the data to be stored are defined prior to the store instruction and in the same sequential portion of code as the store instruction.

A whole-byte, arithmetic expression can be generated to represent any store instruction. Even when a bit-wise operation is performed, this can be represented as a whole-byte expression, as shown in figure 5.6 below.

```
C100 LDAA 0002
C102 EORA #20
C104 STAA 0002

[0002]:=[0002] eor 20
```

Fig. 5.6 Whole-byte representation of bit-wise operation

The intention in the above example was clearly to complement bit 5 of memory-location 0002H and the correct representation for this would be as shown in figure 5.7.

```
[0002]5:=not([0002]5)
```

Fig. 5.7 Correct representation of bit-complement

Before the analyser can properly analyse a portion of code, therefore, it must know whether a bit-wise or a whole-byte operation is being performed. This information is given by the data-type of the operand of the store instruction. If the operand is of type

integer or **pointer**, the analyser uses a routine to perform arithmetic expression generation. If the operand is of type **record** (containing Boolean bits), the analyser uses a routine to perform bit-wise analysis and expression generation.

The bit-wise analysis routine considers only a small subset of the processor's instruction set as valid for bit-wise manipulation. These are the instructions related to loading, masking, shifting, rotating, clearing, storing and logically operating on data objects. If the analysis routine encounters an instruction outside of this subset, it is unable to continue bit-wise analysis of the portion of code containing that instruction. The bit-wise analysis routine then indicates that an arithmetic-style operation has been attempted on a Boolean variable.

If, during its analysis, the bit-wise analyser encounters a variable of type other than Boolean, it terminates analysis of that portion of code and indicates that a bit-wise operation has been attempted on an illegal variable.

Similarly, for the purpose of type-checking, the arithmetic expression generator excludes certain opcodes which are inherently of a bit-wise operative nature (eg. rotate, logical and, exclusive or). The expression generator too, indicates the attempted use of these excluded opcodes on non-Boolean variables. The appearance of bytes containing Boolean variables in arithmetic expressions is also prohibited and flagged as a type-violation.

Where no P-notation specification of a programme exists, the data-flow analyser can still be used as an aid to manual analysis of the programme. In such

cases, no information is available concerning intended types of data objects within the programme. All type-checking is thus disabled. The bit-wise analysis routine attempts to analyse all store operations in terms of bit manipulation. When it encounters an unknown opcode, it terminates analysis of that portion of code and continues with the following store instruction.

When bit-wise analysis is complete, the arithmetic expression generator parses the listing, generating expressions for all remaining, unanalysed store operations. When doing so, the expression generator does not prohibit the use of any of the processor's opcodes. Thus expressions are generated representing all data manipulations. Where such manipulations are obviously of a bit-wise nature, bit-wise expressions are generated. This greatly assists in the manual analysis of a disassembly listing.

Bit-wise analyser

The bit-wise analyser uses character-strings to represent what each of the register bits would have contained if normal execution of a portion of code had occurred. It thus starts from an undefined register bit and determines an expression for the contents of the bit to be stored by working backwards through the instructions. Once all register bits appearing in the expression have been defined (eg. by load, clear), then the expression is complete. The process is best demonstrated by example:

When a store register instruction is encountered, the analyser initiates all its character-strings to represent the undefined register bits as in figure 5.8.

instruction	character-strings							
	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
STAB 9000	B7	B6	B5	B4	B3	B2	B1	B0

Fig. 5.8 Character-string initialisation

Thus after the store, bit 7 of location 9000H would contain bit 7 of register B, bit 6 of location 9000H would contain bit 6 of register B, etc.

Now suppose the previous instruction had been to mask certain bits of register B. Working backwards, the analyser would have modified the character-strings as shown in figure 5.9 below.

ANDB #0F	0	0	0	0	B3	B2	B1	B0
STAB 9000	B7	B6	B5	B4	B3	B2	B1	B0

Fig. 5.9 *Character*-string modification after bit-mask operation

After each *modification of the* character-strings, the analyser checks to see if there are any remaining undefined *register bits. In this case* there clearly are, so analysis continues. Note that the analyser will never continue *back through the end of a* previous construct. A check is thus also made on whether the next statement to be *considered forms the end* of a previous construct. This check is carried out by using information derived by the *control-flow analyser.*

In the example under consideration, *suppose that* the

previous statement had caused register B to be loaded from a memory-location. The analyser would have modified the character-strings as shown in figure 5.10.

LDAB 0C00	|0|0|0|0|[0C00]3|[0C00]2|[0C00]1|[0C00]0|				
ANDB #0F	|0|0|0|0|	B3	B2	B1	B0
STAB 9000	|B7|..|B4|	B3	B2	B1	B0

Fig. 5.10 Character-string modification after register-load operation

The analyser has replaced each undefined register bit (in this case, bits 0 through 3) with the corresponding bit of memory-location 0C00H. A check shows no remaining undefined register bits, so the bit-wise expression-generation procedure is terminated.

Each character-string is an expression which is assigned to the corresponding bit of the store instruction operand. The leftmost string is assigned to bit 7 of the operand, the one to the right to bit 6 of the operand, etc. The only exception to this rule is when a character-string is identical to its corresponding operand and bit number. In such cases, the expression is not generated. The reason for this is that the expression would represent a bit which was left unchanged by a bit-wise operation. Before generation of each expression, type consistency checking is performed.

If any of the bits in a potential expression are part of an integer byte or pointer double-byte, an error message is generated. If all of the bits in a particular expression are of type Boolean, the expression is generated. The above example would produce the expressions in figure 5.11 below.

```
[9000]0:=[0C00]0
[9000]1:=[0C00]1
[9000]2:=[0C00]2
[9000]3:=[0C00]3
[9000]4:=0
[9000]5:=0
[9000]6:=0
[9000]7:=0
```

Fig. 5.11 *Expressions generated* for bit-wise operation

```
|[C000]7|[C000]3|[0001]5|[0001]4|...
                ...|[0001]3|[0001]2|[0001]1| 0 |
```

Fig. 5.12 Character-strings for store-operation *with operand 0001H*

If a store operation with operand 0001H generated the character-strings in figure 5.12 above and the data-table confirmed variables *[C000]7, [C000]3* and *[0001]0* to be of type Boolean, then the series of expressions generated would be as shown in figure 5.13 below.

```
[0001]0:=0
[0001]6:=[C000]3
[0001]7:=[C000]7
```

Fig. 5.13 Expressions generated for bit-copy operation

All other bits of memory-location 0001H were unchanged.

The following examples are from actual outputs of the automatic bit-wise data-flow analyser.

In the first example, the data-table contained the entry: '[0002] integer'.

The analyser detected a type-violation in the portion of code shown in figure 5.14.

```
C100 LDAA 0002
C102 EORA #20
C104 STAA 0002
                    ILLEGAL eor  ON TYPE integer
```

Fig. 5.14 Data type-violation detection

With the relevant bit properly declared as [illegible], the analyser was able to analyse the operati[illegible] was intended - a Boolean-bit complement. This [illegible]n in figure 5.15.

```
data-table:  [0002]  record
                5        Boolean


C100 LDAA 0002
C102 EORA #20
C104 STAA 0002
                    [0002]5:=not([0002]5)
```

Fig. 5.15 Recognition of Boolean-bit complement operation

Note that with type-checking disabled (no *data-table* available), the *above expression* would be the one to be generated.

Bit-copies within or between bytes are conceptually simple operations, but their implementation invariably results in several machine code instructions. It is a laborious and error-prone task to analyse such operations manually from a disassembly listing. Even where a disassembly listing was not to be translated into P-notation, the data-flow analyser was found extremely useful in analysis of such portions of code. The examples of bit-copy operations shown in figures 5.16 and 5.17 were taken directly from the TEM L 30 disassembly listing (Chapter 3).

```
C106 LDAA 1002
C109 ANDA #FD
C10B LDAB 1002
C10E ANDB #10
C110 LSRB
C111 LSRB
C112 LSRB
C113 ABA
C114 STAA 1002
                    [100211]:=[100214
```

Fig. 5.16 *Bit-copy within byte*

```
C146 LDAA 1002
C149 ANDA #FB
C14B LDAB 0002
C14D ANDB #40
C14F LSRB
C150 LSRB
C151 LSRB
C152 LSRB
C153 ABA
C154 STAA 1002
                    [100212]:=[000216
```

Fig. 5.17 *Bit-copy between bytes*

In addition to eight character-strings representing eight bits to be stored, the analyser keeps a ninth, "hidden" string, to keep track of the contents of the carry bit. This is done because many of the shift and

rotate instructions make use of the carry bit. The example of figure 5.18 overleaf shows how the analyser keeps track of the carry bit.

```
A100 LDAA E000
A103 TAB
A104 ROLA
A105 ROLA
A106 ANDA #01
A108 ANDB #FE
A10A ABA
A10B STAA E000
                    {E000}:={E000}7
```

Fig. 5.18 Analysis of manipulation using carry bit.

Arithmetic expression generator

When a 'store-register instruction has an operand of type **integer** or **pointer**, the analyser uses an expression generation routine to translate the relevant portion of the disassembly listing into an arithmetic expression format.

As with the bit-wise analyser, the expression generator starts at a store instruction with an internal representation of an undefined register to be stored. It then works backwards through the listing, generating an arithmetic expression, until all registers appearing in the expression have been defined (eg. by loading, clearing etc.). Again, as with the bit-wise analyser, the expression generator will not continue working backwards past the end of a previous construct. The process is best demonstrated by example.

When the expression generator encounters a 'store

instruction, it initiates a character-string as the name of the register to be stored. This is shown in figure 5.19 below.

processor instruction	character-string
STAA C000	A

Fig. 5.19 Character-string initialisation according to register-name.

Suppose that the previous statement had been to add the processor's registers. The character-string (expression) would have been modified as shown in figure 5.20.

ABA	A+B
STAA C000	A

Fig 5.20 Character-string modification after register-addition

A check for undefined registers shows two undefined registers, so the analyser continues. If the previous two statements had defined the registers by loading them (one from memory, the other immediately), the expression would have been modified as shown in figure 5.21 overleaf.

```
LDAA 2002       [2002]+0E
LDAB #0E ,      A+0E
ABA             A+B
STAA C000       A
```

Fig. 5.21 Character-string modification after register-loading

A check shows no undefined registers, so analysis is complete. Figure 5.22 shows the resultant expression.

```
[C000]:=[2002]+0E
```

Fig. 5.22 Addition-operation representation

Attempted loading of a register from a memory-location containing Boolean bits generates a type-violation message.

This process is similar for all opcodes of the processor's instruction-set. A slight problem occurs with the insertion of parentheses. When the expression is modified by insertion of a character-string in place of a register name, parentheses are placed around the character-string to maintain the sense of the expression. This is shown in figure 5.23 overleaf.

```
LDA 1000        asl([1000]+1)
INCA            asl(A+1)
ASLA            aslA
STAA 2000       A

[2000]:=asl([1000]+1)
```

Fig. 5.23 Insertion of parentheses

This can unfortunately lead to redundant parentheses, as in figure 5.24.

```
LDAA 1000       asl(([1000]+1)+1)
INCA            asl((A+1)+1)
INCA            asl(A+1)
ASLA            aslA
STAA 2000       A

[2000]:=asl(([1000]+1)+1)
```

Fig. 5.24 Insertion of redundant parentheses

Removal of redundant parentheses is proposed as a refinement of the arithmetic-expression generator.

5.3.4 Conditional branches

In the analysis of the code preceding a store instruction, the operand of the store instruction (data destination) is used to determine whether a bit-wise or whole-byte operation is being performed. In the case of branches, however, the operand of the branch

instructic~ is the destination address of the branch. No data element is available to determine the type of analysis to be done on the code immediately *preceding a* conditional branch instruction.

The strutegy employed in th*is case*, therefore, is the same as for the analysis of memory-location manipulation with type-checking disabled. Bit-wise *analysis* is attempted for each conditional branch instruction and if this fails, the expression generator is used to generate an *arithmetic-style expression.* Failure of bit-wise analysis is caused by an irrelevant opcode or a data object of type other than Boolean.

The major differences between conditional branch analysis and store instruction analysis are the type of *expression* to be formed and the initialisation of character-strings.

In the case of a conditional branch instruction, the analyser must produce an expression of condition, not an *expression of assignment.* The conditional branch instruction represents the final test to be performed after the necessary data manipulation. Both in the *case* of bit-wise expression generation and arithmetic expression generation, the analyser forms the final expression by adding a *textual representation* of the branch instruction to the relevant character-string.

Before *forming* a textual representation of a conditional branch instruction however, the logic of the test causing the branch is inverted. *This is* because of the way in which conditional branches are used. For example, in an **if-then** construct, the body of the construct *will be executed* if the condition of the branch is <u>not</u> met. So the textual representation of a test is not a representation of the branch

condition, but a representation of the construct-body-execution condition.

Some examples of textual representations of conditional branch instructions are shown in figure 5.25 below.

The instruction immediately preceding a conditional branch instruction, together with the branch instruction itself, is used for initialisation of the character-strings for both arithmetic and bit-wise

processor instruction	textual representation (inverted logic)
BEQ	<>0
BGT	<=0
BNE	=0

Fig. 5.25 Textual representations of conditional branch instructions

expression generation. This is because the instruction immediately preceding a conditional branch instruction sets up the condition for the branch. Examples are shown in figure 5.26 below.

```
bit-wise analysis
     CMPA #03   |A7-0|A6-0|A5-0|A4-0|...
                           ...|A3-0|A2-0|A1-1|A0-1|
     BNE  2000

arithmetic expression generation
     CMPA #03    A-03
     BNE  2000
```

Fig. 5.26 Initialisation of character-strings before conditional branch instruction

Analysis continues in both cases using the *same routines as* for register-store instruction analysis.

When no registers remain undefined, the analyser forms a final expression or series of expressions by addition of the inverted-logic textual text representation.

Sample analyser outputs for some conditional branches follow:

```
003E LDAA 4000
0092 ANDA #06
0094 BEQ  F0E0
                    [4000]1<>0
                    [4000]2<>0
```

Fig. 5.27 Successful bit-wise analysis

In the above example, bit-wise analysis was successful.

```
009E LDAA 4020
00A1 INCA
00A2 BNE  F0A6
                    [4020]+1=0
```

Fig. 5.28 Unsuccessful bit-wise analysis

In the above example, bit-wise analysis was unsuccessful - an arithmetic expression was generated.

```
0073 LDAA 4018
0076 INCA
0077 ABA
0078 BLT  F0A6
                    ((4018]+1)+B>=0
```

Fig. 5.29 Premature termination of analysis

In the above example, analysis was terminated by attempted analysis past the end of a previous construct. The state of the expression at the time of termination is printed, showing undefined registers.

5.4 Results of Data-Flow Analysis

A method of automatic data-flow analysis has been described in this chapter. Sample outputs from an implementation of the method have been shown to demonstrate the effectiveness of the analysis method. By reference to a data-table derived from the P-notation specification of a programme, an automatic data-flow analyser is capable of flagging data-type violations. Data-type violations consist both of attempting to combine incompatible data-types within an expression, and of attempting to operate on a data-type with an operator incompatible with that data-type.

Data-type checking can be disabled when the analyser is used as an aid to manual analysis of programmes which do not have P-notation specifications.

Expressions generated by the analyser are either of a whole-byte, arithmetic nature or of a bit-wise, logical nature, depending on the data-types of the variables in the expressions. With data-type checking disabled, the nature of generated expressions is at the discretion of the analyser.

Between them, the proposed automatic methods of control-flow and data-flow analysis and a manually derived data-table provide sufficient information for the translation of a disassembly listing into P-

notation.

The following chapter describes a proposed method of using this information to translate automatically a disassembly listing into P-notation.

CHAPTER 6 PROGRAMME TRANSLATION

The previous two chapters have described methods of analysing a disassembly listing in terms of its control-flow and data-flow. The purpose of these flow-analysis techniques is to provide information for the translation of a disassembly listing into P-notation.

This chapter describes a method of automatically translating a disassembly listing into P-notation. The method uses information from the automatic flow-analysers, together with information from the manually-derived data table described in Chapter 5. The translator parses the disassembly listing, checking whether each address represents either a control-flow node or a data-flow expression. Although control and data-flow translation occur in a single pass of the listing, they are described separately for clarity.

6.1 Structure Translation

The control-flow analyser described in Chapter 4 identifies loop and selection constructs in a disassembly listing. The analyser creates a file containing the memory locations of the nodes of all identified constructs. This information is used by an automatic programme translator which translates the control-flow framework of the disassembly listing into P-notation format.

6.1.1 Formatting of constructs

The selection constructs in P-notation are the **case** construct and the **if-then-else** construct. The **if-then** construct is a simpler, single-bodied version of the

if-then-else *construct.*

The translator uses a character string to represent the line of indentation of the P-notation programme at any point in the programme. The character string contains blanks and, where relevant, key words such as "if", "case", etc. This character string is updated whenever a structure node is encountered, as determined by the output file of the control-flow analyser.

Successful control-flow analysis is a pre-requisite for translation of the control-flow framework of a disassembly listing into P-notation (4.3). . The translator, therefore, does no checking on correspondence between key words. It simply translates according to information in the output file of the control-flow analyser. For example, if the control-flow analyser has shown a particular address in the disassembly listing to correspond to **else3**, the translator determines the indentation level of the corresponding **if__3**. It then modifies its indentation character string to contain the word "else" in the position of the "if" of **if__3**. Thus the "else" of **else3** will be printed vertically below the "if" of **if__3**. The modified indentation character string ensures that until the next control-flow node, data manipulation expressions will be printed directly below each other in the correct horizontal position.

An example of control-flow translation is shown in figure 6.1 overleaf. Data manipulations have been omitted.

if__1	if
rept2	rept
untl2	untl
if__2	if
else2	else
comp2	end
else1	else
comp1	end

Fig. 6.1 *Control-flow translation*

A further example, *containing* a **case** construct, is shown in figure 6.3.

case1	case
of__1	of
if__2	if
else2	else
comp2	end
of__1	of
of__1	of
comp1	end

Fig. 6.2 Control-flow translation including **case** construct

6.2 Data-Flow Translation

All data type-checking and expression generation is performed by the data-flow analyser (Chapter 5). The *translator* has only to insert variable names in place of memory locations in expressions and insert the expressions into the control-flow framework. *Data-flow*

translation makes use of the manually derived data table and the output file of the data-flow analyser.

As described in Chapter 5 (Data-Flow Analysis), conditional branch statements generate expressions *representing construct* body execution conditions. These conditions are used to complete the test predicates of the control-flow framework. In the *case* of an **if** statement, the word "then" is added to the derived condition. An example is shown below.

```
data table entry:  [2000]  counter   integer

1000 LDAA 2000
1003 CMPA #55
1005 BNE  100C        if counter-55=0 then
1007 CLR  2000          counter:=0
100A BRA  100F        else
100C INC  2000          counter:=counter+1
100F NOP              end
```

Fig. 6.3 Translation of **if-then-else** construct

A further example, containing a **repeat until** loop, is shown in *figure 6.4 overleaf*. In this example, the loop index is held in a memory location. Where this is not done (loop index held in a register), the *data-flow* analyser would have been unable to analyse the code. The register name would appear in the test predicate, as in figure 6.5 overleaf. Further analysis would have to be manually performed.

```
data table entry:  [20F0]  counter   integer

1040                       repeat
 -                          -
 -                          -
1080 DEC  20F0              counter:=counter-1
 -                          -
 -                          -
1090 LDAA 20F0
1093 BNE  1040             until counter=0
1095 NOP
```

Fig. 6.4 *Translation of* **repeat-until** construct

```
      C000 LDAA 1000
      C003                 repeat
       -
       -
      C020 DECA
       -
       -
      C050 CMPA #04
      C052 BNE  C003       until A-04=0
      C054 NOP
```

Fig. 6.5 Register-name appearing in test-predicate

Difficulty of test predicate insertion arises with the **case** construct. Each conditional branch to a ***case*** body causes generation of a test predicate. This predicate is inserted at the head of each **case** body, preceded by the word "of", as shown in figure 6.6 overleaf.

original P-notation	programme		translated programme
case count of	0100 LDAA 2000		case
	0103 CMPA #01		
	0105 BEQ 0111		
	0107 CMPA #02		
	0109 BEQ 0120		
	010B CMPA #03		
	010D BEQ 0130		
	010F BRA 0150		
1:-	0111 -		of count-1=0: -
-	-		-
	BRA 0150		
2:-	0120 -		of count-2=0: -
-	-		-
	BRA 0150		
3:-	0130 -		of count-3=0: -
-	-		-
end	0150		end

Fig. 5.6 Translation of **case** construct

The *first* **case** body is often executed by default - none of the other **case** tests resulted in a branch. In such cases, no test predicate is generated for that ***case*** body. Further analysis must be manually performed.

So translated **case** constructs are not fully authentic. It is felt, however, that the **case** construct translation procedure described above generates an easily readable, high-level *version of a* **case** construct, albeit slightly different from the standard P-notation **case** construct definition.

6.3 Results of Programme Translation

The automatic programme translator described in this chapter translates a disassembly listing of the form obtained from the TEM L 30 (Chapter 3) into P-notation.

The translator uses results from the control and data-flow analysers of the previous two chapters, together with a manually derived data table described in the previous chapter, to perform its task.

Translation is performed in a single pass of the disassembly listing. All disassembly listing statements referenced in the control and/or data-flow analyser output files have control and/or data-flow relevance. They initiate the generation of appropriate P-notation statements.

The overall effectiveness of the proposed methods of disassembly, analysis and translation of a machine code programme into P-notation is assessed in the next chapter.

CHAPTER 7 CONCLUSIONS

Previous chapters have discussed how a machine-code programme can be shown to be consistent with its high-level specification. This process essentially consists of four phases.

A series of traces of a microprocessor running the actual machine code programme is obtained. These individual traces are processed to form one complete disassembly listing of the programme.

The disassembly listing is analysed in terms of its control flow. Standard constructs are identified and unrecogniseable constructs are flagged.

The disassembly listing, together with information generated by the control-flow analyser, is then analysed in terms of data-flow. Information from the specification's data declaration is used to authorise memory location manipulations and flag attempted data-type violations.

Finally, information from the control-flow analyser, the data-flow analyser and the specification's data declaration is used to translate the disassembly listing into P-notation.

Demonstration of consistency between a machine-code programme and its high-level specification is then by direct comparison of the two. This comparison is at present a manual task, but has the potential to be automated.

7.1 Techniques Developed

7.1.1 Features

Operations of the *microprocessor* are traced by a logic analyser as the microprocessor is forced by external stimulus to traverse each path of its programme. Thus actual *code, as executed,* is used as input to the validation system. If desired, this code can be compared with its equivalent PROM listing to indicate "dead" or *unreachable* code in the PROM.

The validation techniques need not necessarily operate from *processor traces.* Code at higher levels can be used instead. For example, portions of assembler-code can be checked against their P-notation equivalents before *the entire program* is assembled and run.

Even where no P-notation specification exists, the *validation techniques* will do a complete control-flow analysis and translation of the code to a register-independent level. This greatly assists *readability and analysis* of code where little or no documentation is available.

If a programme is inconsistent with its specification, this will be shown in one of two ways. If there are unrecognizeable constructs or data-type *inconsistencies,* these will be flagged by the analysers. If there are not, the resultant P-notation representation will be seen to *differ from* its *specification* by inspection.

7.1.2 Limitations

The analysis techniques *presented* will not

automatically analyse all machine-code programmes. Programmes *must* be coded according to certain conventions in order to be analysed automatically. In some cases this is desireable; in others, unfortunate.

Limitations of control-flow analysis

Programmes whose structure exactly mirrors that of their P-notation specification will always be *analyseable* in terms of control-flow. This is necessary and sufficient for control-flow validation of such programmes. Where a programmer has inadvertently or intentionally deviated from the P-notation structure, the structure of the resultant programme may or may not be analyseable. If it is not, the *validator knows immediately* that the programme contains unsound structures. If it is, it will be seen by inspection to differ from its specification.

Programmes without P-notation specifications can still be analysed in terms of their control-flow. If the control-flow analyser finds no unknown structures in a programme, the programme has been shown to contain only sound structures. If the analyser *finds* unknown structures, an operator may indicate to the analyser that such structures are to be ignored if he finds them, by manual inspection, to be acceptable.

A limitation of the control-flow analyser, albeit relatively minor, is its *all-or-nothing recognition* of a particular structure. If a structure contains any irregularity, the analyser is of no assistance to the validator - code must be manually inspected.

Limitations of data-flow-analysis

Every instance of *data-manipulation* will generate either an arithmetic-type representation, or a logical (bit-manipulation) representation. If a programme *exactly mirrors* its P-notation specification, then all such representations will be valid. Limitations of analysis are epitomised by *processor registers* appearing in such representations. This occurs when a programmer carries registers through structure end-boundaries. The *data-flow analyser* is then unsure of the origin and thus the contents of such registers and can represent them only by their register names. It is obvious that this limitation can be minimised by use of an appropriate coding technique.

Another limitation occurs with indexed addressing. The data-flow analyser is unaware of the contents of the *index register* and can thus represent the absolute address only as index-register plus offset.

Where programmes were not written from P-notation specifications, the data-flow-analyser will still generate arithmetic or *logical representations*, but these cannot be expected always to be valid, since the analyser has no information about the types of the data items involved. It will not, for example, notice if an arithmetic operation is performed on a Boolean variable. This is not a serious *limitation since*, used in this mode, the analyser is essentially an aid to manual analysis, rather than an automatic validation tool.

General limitations

The analysers are *unavoidably processor-specific*. In

the case of the control-flow analyser and expression generator, the operating programmes are non-processor-specific, working from data bases containing processor data. It is, in the case of these two analysers, a simple task to adapt them to other processors by changing their data bases.

In the case of the bit-manipulation analyser, however, the operating programme has to simulate the operation of the processor and is thus, in itself, processor-specific. So to adapt it to another processor would involve changes which, although simple to perform, would be substantial.

7.1.3 Recommended refinements

Control-flow-analysis refinements

The all-or-nothing recognition of individual structures is a shortcoming of the control-flow-analyser. It is not a serious shortcoming, since properly structured code will always be analyseable (compiler-generated code, for example, will always have proper structure). It is only improperly structured code which will need manual analysis.

However, certain bad coding practices lead to common forms of improper structure. The overlapping of structures is the only form of improper structure detected by the control-flow analyser. A list of overlapping structures is printed during the third pass of the control-flow analyser.

Research could be done to identify and appropriately treat other common forms of bad structure. This would greatly reduce any manual code inspection which might

otherwise have been necessary.

Data-flow-analysis refinements

A limitation of the data-flow-analyser is its inability to take into account the structure of the programme. When data-flow is heavily dependent on the structures within a programme, it is not practical to do data-flow analysis across such structures. Generated expressions become multiple expressions, selection of a particular expression being dependent on the data active in previous constructs. In such cases it is better to admit defeat, since translated code becomes even less readable than the code from which it was generated!

When too many previous constructs affect the contents of a register at a given position in the programme, it is clearer to generate an expression involving the register name than to try to indicate the possible contents of the register.

However, in certain simple cases, where for example, the body of an if-then construct does not affect a register, analysis could be continued above the body of the construct. So a proposed refinement would be to identify instances where registers are carried "around" simple constructs and to continue analysis above such constructs.

General refinements

Manual intervention is required in formulating a data table from the P-notation variable specification. In the case of larger programmes, this process could entail a substantial amount of work, not to mention of

course, the unfavourable human trait of *inadvertent error-seeding*! So automation of this process is a proposed refinement. It would have to be determined whether there would always be *sufficient formal* information within the P-notation variable declaration and assembler declaration for automation of the above process.

7.2 Conclusions

The absence of effective methods of validating real-time process control software was the motivation for the research described in this dissertation.

Real-time process control *software has* attributes such as stringent timing constraints, cyclic programmes and low-level bit-manipulation, which are not present in many other software *applications*. Therefore, established validation techniques for other software applications have very limited effectiveness in the validation of real-time *process control* software. But such software is being used increasingly in the control of life-critical systems. It simply has to be validated.

A major aspect of validation is the proof of consistency *between a programme* and its specification. The goal of the research described in this dissertation was to show how such a method could be developed, the *programme and specification* being in the forms of machine code and P-notation, respectively. Automation of the method was also of prime concern.

A method of deriving a complete disassembly listing of a machine code programme has been developed. The method *consists* of tracing the operations of a

microprocessor as it executes the machine code in question. The microprocessor system is stimulated to cause the microprocessor to execute all paths of the programme. All traces thus obtained are edited and sorted to produce the complete disassembly listing.

Methods of automatic control-flow and data-flow analysis of a disassembly listing have also been developed. These methods have been shown to be effective in all cases where assembler programmes have been directly and formally derived from their specifications. The analysis methods have also been shown to be effective in pinpointing inconsistencies between programmes and their specifications.

It has been shown how, by use of control-flow and data-flow analysis and use of information from the specification's data declaration, an assembler programme can be translated into P-notation. It has also been shown how this translation process can be automated.

The methods of control-flow and data-flow analysis have been shown to be useful also in the analysis of assembly language programmes and their translation to a register-independent level where no P-notation specification exists.

The above analysis and translation techniques could be integrated into an interactive validation environment for validating machine code programmes with respect to their high-level specifications.

REFERENCES

Adrion, W.R., Branstad, M.A. and Cherniavsky, J.C. (1982) **Validation, Verification and Testing of Computer Software**, Computing Surveys, vol. 14, no. 2, Jun 1982, pp. 159-192.

Allen, F.E. and Cocke, J. (1976) **A Program Data Flow Analysis Procedure**, Communications of the A.C.M., vol.19, no. 3, Mar 1976, pp. 137-147.

Benson, J. (1981) **A Preliminary Experiment in Automated Software Testing**, ACM Sigsoft, Software Engineering Notes, vol. 6, no. 3, Jul 1981, pp. 68-75.

Branstad, M.A., Cherniavsky, J.C. and Adrion, W.R. (1980) **Validation, Verification and Testing for the Individual Programmer**, Computer, Dec 1980, pp. 24-30.

Carré, B.A. (1980) Software Validation, Microprocessors and Microsystems, vol. 4, no. 10, Dec 1980, pp. 395-406.

Clarke, L.A. (1976) **A System to Generate Test Data and Symbolically Execute Programs**, I.E.E.E. Transactions on Software Engineering, vol. SE-2, no. 3, Sept 1976, pp. 215-222.

De Millo, R.A., Lipton, R.J. and Sayward, F.G. (1978) **Hints on Test Data Selection : Help for the Practicing Programmer**, Computer, Apr 1978, pp. 34-41.

Deutsch, M.S. (1979) **Verification and Validation**, in Jensen, R.W. and Tonies, C.C. eds., "Software Engineering", Prentice-Hall Inc., New Jersey, 1979.

Gerber (1985) **Generation, Documentation and Validation of Software for the Siemens Electronic Interlocking**, Siemens LTD., Department T/SI-ST (Railway Signalling), 1985.

Goodenough, J. B. and Gerhart, S.L. (1975) **Toward a Theory of Test Data Selection**, I.E.E.E. Transactions on Software Engineering, vol. SE-1, no. 2, Jun 1975, pp. 156-173.

Gustafson, D.A. (1984) **Guidance for Test Selection Based on the Cost of Errors**, Proceedings AFIPS National Computer Conference, 1984, pp. 425-429.

Hoare, C.A.R., (1975) **Data Reliability**, Proceedings of the International Conference on Reliable Software, Los Angeles, 1975, pp. 528-533.

Howden, W.E. (1977) **Symbolic Testing and the DISSECT Symbolic Evaluation System**, I.E.E.E. Transactions on Software Engineering, vol. SE-3, no. 4, Jul 1977, pp. 266-278.

Howden, W.E. (1978) **An Evaluation of the Effectiveness of Symbolic Testing**, Software - Practice and Experience, vol. 8, 1978, pp. 381-397.

Howden, W.E. (1980a) **Functional Program Testing**, I.E.E.E. Transactions on Software Engineering, vol. SE-6, no. 2, Mar 1980, pp. 162-169.

Howden, W.E. (1980b) **Applicability of Software Validation Techniques to Scientific Programs**, ACM Transactions on Programming Languages and Systems, vol. 2, no. 3, Jul 1980, pp. 307-320.

Howden, W.E. (1982) **Life-Cycle Software Validation**, Computer, Feb 1982, pp. 71-78.

HP64000 Logic Development System, System Overview, Hewlett Packard Company/Logic Systems Division, Colorado, USA, 1982.

King, J.C. (1976) **Symbolic Execution and Program Testing**, Communications of the ACM, vol. 19, no. 7, Jul 1976, pp. 385-394.

Kopetz, H. (1979) **Software Reliability**, The Macmillan Press Ltd., 1979.

Leveson, N.G. and Harvey, P.R. (1983) **Analyzing Software Safety**, I.E.E.E. Transactions on Software Engineering, vol. SE-9, no. 5, Sept 1983, pp. 569-579.

Ludewig, J.L. (1981) **Specification of a Specification Language**, paper for presentation at IFAC/IFIP Workshop on Real-Time Programming, Kyoto, Japan, 1981.

Meyers, G.J. (1979) **The Art of Software Testing**, John Wiley and Sons, New York, 1979.

Miller, E.F. (1977) **Program Testing: Art Meets Theory**, Computer, Jul 1977, pp. 42-51.

Patterson, D.A. (1981) **An Experiment in High Level Language Microprogramming and Verification**, Communications of the ACM, vol, 24, no. 10, Oct 1981, pp. 699-709.

Quirk, W.J. (1983) **Recent Developments in the SPECK Specification System**, HARWELL Report CSS.146, 1983.

Quirk, W.J. (1985) ed. **Verification and Validation of Real-Time Software**, Springer-Verlag, Berlin, 1985.

Ramamoorthy, C.V. and Ho, S.F. (1975) ***Testing Large Software with Automated Software Evaluation Systems***, I.E.E.E. Transactions on Software Engineering, vol. SE-1, no. 1, Mar 1975, pp. 46-58.

Ross, D.T. and Schoman, K.E., Jr. (1977) **Structured Analysis for Requirements *Definition***, I.E.E.E. Transactions on Software Engineering, vol. SE-3, no. 1, Jan 1977, pp. 6-15.

Rzevski, G. (1981) **Recent Advances in Software Reliability Methods**, Quality Assurance, vol. 7, no. 3, Sept 1981, pp. 80-87.

Short, R.C. (1983) **Software Validation for a Railway Signalling System**, publ. *IFAC Safecomp* '83, Cambridge, 1983.

SPADE, Program Validation *Limited*, Southampton, 1985.

Taylor, J.R. (1982) **Fault Tree and Cause Consequence Analysis for Control *Software* Validation**, Riso National Laboratory, Roskilde, Denmark, Jan 1982.

Taylor, R.N. (1983) **An Integrated Verification and Testing Environment**, Software - Practice and Experience, vol. 13, 1983, pp. 697-713.

Teichroew, D. and Hershey, E.A. III (1977) **PSL/PSA: A Computer-Aided Technique for Structured Documentation and *Analysis* of Information Processing Systems**, I.E.E.E. Transactions on Software Engineering, vol. SE-3, no. 1, Jan 1977, pp. 41-48.

TEM L 30 Block Instrument Controller, A *Technical Description*, Issue 1, M.L. Engineering (Plymouth) Ltd., Plymouth, 1983.

VERkshop III proceedings, A.C.M. Sigsoft Software Engineering Notes, vol. 10, no. 4, Aug 1985, pp. i-v, 1-100.

Weyuker, E.J. and Ostrand, T.J. (1980) **Theory of Program Testing and the Application of Revealing Subdomains**, I.E.E.E. Transactions on Software Engineering, vol. SE-6, no. 3, May 1980, pp. 236-246.

White, L.J. and Cohen, E.I. (1980) **A Domain Strategy for Computer Program Testing**, I.E.E.E. Transactions on Software Engineering, vol. SE-6, no. 3, May 1980, pp. 247-257.

Young, S. (1980) **P-notation: High Level description language for software design**, Microprocessors and Microsystems, vol. 4, no. 7, Sept 1980, pp. 267-272, no. 8, Oct 1980, pp. 307-311, no. 9, Nov 1980, pp. 363-369, no. 10, Dec 1980, 411-419.

APPENDIX A TEM L 30

A.1 Introduction

The "TEM L 30 Block Instrument Controller" was used as a guinea-pig microprocessor system for the production of a disassembly listing (Chapter 3). This appendix contains a brief description of the operation and features of the TEM L 30.

A.2 Overview of Operation

In railway signalling, a "block instrument" is a device which sends signalling information over a pair of wires to another block instrument. The information is sent in the form of manually-pulsed current of pre-defined negative or positive polarity with respect to the wire-pair. Thus the information that can be transmitted over the wire-pair is current in one direction, current in the other direction or the absence of current. The receiving block-instrument indicates to its operator the presence or absence of current and its polarity.

Wire-pairs longer than even a few hundred metres are expensive and time-consuming to install, so it is clearly desireable to replace them with radio links. Since the information transmitted between block instruments, despite its simplicity, is of critical importance, any radio link replacing a wire-pair would have to transmit this information in a fail-safe manner. One such radio link is the "TEM L 30 Block Instrument Controller", manufactured by M.L. Engineering (Plymouth) Limited. One TEM L 30 is connected to two block instruments at the same end of adjacent loops, as shown in figure A1.

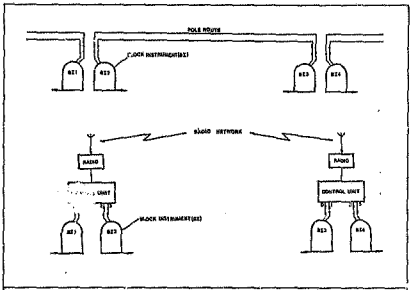


Fig. A1 Replacement of wire pair by radio link

A.3 System Operation

The control unit consists of three basic parts:

A data modulation/demodulation system designed to interface directly with a radio set.

Power supply, control and input/output interface circuitry designed to simulate the characteristics of a wired connection between block instruments.

A duplicated microprocessor-controlled logic system and message store (figure A2).

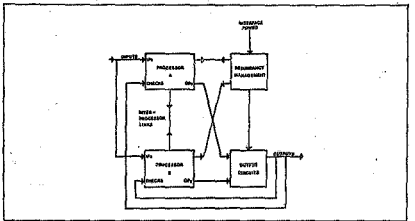


Fig. A2 Duplicated fail-safe microprocessor-based control system

Information is sent from one control unit to the next in the form of a coded message consisting of a fixed length portion followed by a variable length portion. The operation of the control unit is best studied by briefly considering the sequence of events which occurs when a message is sent from one unit to another.

A.3.1 Operation of the block instrument - input

Operation of the block instrument in the normal way alerts the control unit and powers-up the system. Messages may exist on either of the two instruments connected to the controller independently. They consist of a series of pulses of current of either polarity. They are coded by sampling at the block message sample frequency of 15 Hz.

A.3.2 *Encoding*

The fixed length portion of the message holds the unit

address, parity and a synchronisation sequence. The variable length portion contains the signalling information and is assembled as follows. The variable length sequence record is opened whenever an input line becomes active, whereafter the input is sampled at the block message sampling frequency until the input has remained zero long enough for the message to have been judged to have finished. When encoding, the polarity is judged by the first current pulse and thereafter the processor provides the ability to sink current of only that polarity.

A.3.3 Data transmission

There is an internal message store for messages awaiting transmission, validation or output to one of the block instruments. It is divided into equal parts, one for each block instrument. Each part can be used for either incoming or outgoing messages. For security and availability reasons, messages are triplicated sequentially and two of the three messages are required to be identical before the receiving control unit will output the signal information to the block instrument. A message is transmitted only after a check has been made to ensure that no other radio is transmitting.

A.3.4 Data receipt

All control units receive and decode all data messages, but messages are stored only if the message address applies to the particular location address and the appropriate store is available. The first two messages are stored and compared with each other and with the third message, which is not stored. If any two messages agree with each other, then one of them is

sent out to the relevant block instrument.

A.3.5 Operation of the block instrument - output

Messages are output to the instruments at the block message sample frequency, sequentially, bit by bit until completion. For each logical one that was received, one line is driven high and the other is held low, according to the received polarity. For each logical zero that was received, neither line is driven.

A.4 Safety Features

The main safety feature is a symmetrical microprocessor board, on which two independent processors perform the same function and constantly check each others actions. Any disagreements between the processors cause them to blow a power interface fuse, thus isolating the system outputs and preventing any faulty messages from being sent out. The various checking procedures of the processors include:

Independent checking of the state of the source and sink lines.

Independent "watchdog" circuits which require continual refresh.

Continual self-checking by each processor of its ability to read from and write to its own RAM.

Continual inter-checking between the two processors of the contents of the PROM memories.

Frequent testing by each processor of its ability to blow the fuse.

Low supply voltage detection on the power board.

Continual self-checking of transmitted data.

In addition to these safety checks, the radio messages are protected from the effects of noise by a triple-layer system: a Hamming coding; a Manchester II coding and the requirement that two identical messages must be received out of three transmitted.

APPENDIX B P-NOTATION SYNTAX

This appendix presents P-notation syntax in Backus-Naur Form (BNF).

Words belonging to P-notation are *printed in* **boldface**, e.g.:**type**,**repeat**. The following symbols belong to BNF and are not part of the P-notation *syntax:*

::= means "is defined as".
| means "or".
{} indicate items which may be repeated zero or more times.

All other symbols are part of the P-notation syntax.

actual parameter::= expression|variable
adding-operator::= +|-|**or**|**eor**
array-type::= **array**[digit-sequence]**of** component-type
array-variable::= variable
assignment-statement::= variable:=expression
binary-value::= 0|1{0|1}
block::= declaration-part statement part
case-element::= case-list:statement
case-list::= case-list-element{,case-list-element}
case-list-element::= constant|constant..constant
case-statement::= **case** expression **of** case-element {;case-element} **end**|**case** expression **of** case-element {;case-element} **else** statement{;statement} **end**
complemented-factor::= signed-factor|**not** signed-factor
component-type::= type
component-variable::= indexed-variable|field-designator
compound-statement::= **begin** statement{;statement} **end**
conditional-statement::= if-statement|case-statement
constant::= unsigned-integer|sign unsigned-integer|*constant-identifier*|*sign* constant-identifier|string

```
constant-definition-part::= const constant-definition
   {;constant-definition}
constant-definition::= identifier=constant
constant-identifier::= identifier
control-variable::= variable-identifier
declaration-part::= {declaration-section}
declaration-section::= constant-definition-part|
   type-definition-part|variable-declaration-part|
   procedure-declaration-part
digit::= 0|1|2|3|4|5|6|7|8|9
digit-sequence::= digit{digit}
empty::=
expression::= simple-expression{relational-operator
   simple-expression}
factor::= variable|unsigned-constant|(expression)
field-identifier::= identifier
field-list::= empty|field-identifier
   {,field-identifier}:type|field-identifier
   {,field-identifier}:type{%binary-value}
final-value::= expression
for-list::= initial-value to final-value|
   initial-value downto final-value
for-statement::= for control-variable:=for-list
   do statement
formal-parameter-section::= parameter-group|
   var parameter-group
hexdigit::= digit|A|B|C|D|E|F
hexdigit-sequence::= hexdigit{hexdigit}
identifier::= letter{letter-or-digit}
identifier-list::= identifier{,identifier}
if-statement::= if expression then statement
   {else statement}
indexed-variable::= array-variable[simple-expression
   {,simple-expression}]
initial-value::= simple-expression
```

```
letter::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|
   W|X|Y|Z|a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|
   v|w|x|y|z|_
letter-or-digit::= letter|digit
multiplying-operator::= *|/|div|mod|and|shl|shr|shra|
   rotr|rotl
parameter-group::= identifier-list:type-identifier
pointer-type::= identifier
procedure-declaration-part::= {procedure-declaration}
procedure-heading::= procedure identifier;|procedure
   identifier(formal-parameter-section
   {;formal-parameter-section});
procedure-statement::= procedure-identifier|
   procedure-identifier(actual-parameter
   {,actual-parameter})
programme-heading::= programme programme-identifier
programme::= programme-heading block
programme-identifier::= identifier
record-type::= record field-list end
record-variable::= variable
record-variable-list::= record-variable
   {,record-variable}
repeat-statement::= repeat statement{;statement}
   until expression
repetitive-statement::= while-statement|
   repeat-statement|for-statement
scalar-type::= (identifier{,identifier})
sign::= +|-
simple-expression::= term{adding-operator term}
simple-statement::= assignment-statement|
   procedure-statement
simple-type::= scalar-type|subrange-type|
   type-identifier
statement::= simple-statement|structured-statement
statement-part::= compound-statement
string::= '{character}'
string-type::= string[constant]
```

```
structured-statement::= compound-statement|
   conditional-statement|repetitive-statement|
   with-statement
structured-type::= string-type|array-type|record-type
subrange-type::= constant..constant
term::= complemented-factor{multiplying-operator
   complemented-factor}|indirection-operator
   identifier{multiplying-operator complemented-factor}
type-definition::= identifier=type
type-definition-part::= type type-definition
   {;type-definition}
type-identifier::= identifier
type::= simple-type|structured-type|pointer-type
unsigned-integer::= digit-sequence|$hexdigit-sequence
variable::= variable-identifier|component-variable|
   referenced-variable
variable-declaration::= identifier-list:type|
   identifier-list:type absolute hexdigit
variable-declaration-part::= var variable-declaration
   {;variable-declaration}
variable-identifier::= identifier
while-statement::= while expression do statement
with-statement::= with record-variable-list
   do statement
```

APPENDIX C CONTROL-FLOW ANALYSIS ALGORITHMS

This appendix contains algorithms which were used to identify and label **if-then-else**, **loop** and **case** constructs within a disassembly listing.

CB is an abbreviation for conditional branch.
UCB is an abbreviation for unconditional branch.

A statement which **precedes** another is lower in absolute programme address. A statement preceding another is its **predecessor**.

A statement which **succeeds** another is higher in absolute programme address. A statement succeeding another is its **successor**.

The **destination** of a branch statement is the statement to which it branches.

C.1 Algorithm for case Identification

```
start at beginning of listing
while not end of listing do
  move to next statement
    if statement labelled with multiple comp labels then
      if each comp has corresponding else and if labels then
        label if corresponding to lowest-numbered-comp 'case#'
        (where # is number of lowest-numbered-comp)
        label else's corresponding to all other comp's 'of__#'
        label multiple comp statement 'comp#'
      endif
    endif
endwhile
```

C.2 Algorithm for **if-then-else** and **loop** Identification

```
set # to 1 and start at beginning of listing
while not end of listing do
  move to next statement
  if CB forwards then
    if CB destination predecessor is UCB forwards then
      label CB 'if__#'
      label CB destination 'else#'
      label UCB destination 'comp#'
      increment #
    else
      if CB destination predecessor is UCB backwards then
        if UCB destination precedes CB then
          label CB 'whil#'
          label UCB 'endw#'
          increment #
        endif
      else
        label CB 'if__#'
        label CB destination 'comp#'
        increment #
      endif
    endif
  else
    if CB backwards then
      label CB 'until#'
      label CB destination 'rept#'
      increment #
    else
      if (CB or UCB) and unlabelled then
        label CB or UCB 'unkn#'
        label CB or UCB destination 'endu#'
        increment #
      endif
    endif
  endif
endwhile
```