North-Holland Publishing Company
Microprocessing and Microprogramming 8 (1981) 167–178

FREM: Firmware Requirements Engineering Methodology

M. Tsuchiya
TRW, One Space Park, Redondo Beach, CA 90278, U.S.A.

Considerations for the design of Firmware Requirements Engineering Methodology (FREM) are investigated for practical use in firmware development. The design of FREM is based on the existing Software Requirements Engineering Methodology (SREM), an automated requirements specification and analysis system, which has recently received increased acceptance. SREM and its two components, the Requirements Statement Language (RSL) and the Requirements Engineering and Validation System (REVS), are briefly reviewed. The minimum extensions necessary to implement FREM are discussed. The firmware requirements specification procedure is described with an example to illustrate the use of FREM, and further improvements for requirements analysis and simulation are suggested. The results show that FREM works well for firmware engineering and that it could be a valuable tool as the firmware development activities increase.

Keywords: Firmware Engineering, Microprogramming, Requirements Specification, Requirements Engineering Methodology, Validation.

1. Introduction

The recent resurgence of interest in microprogramming has created a new discipline: firmware engineering [10]. This may be an indication that firmware is in increased use and, as with software, is in need of better and more systematic development procedures or methodologies. To find solutions, as history repeats, firmware turned once again to the software disciplines for directions. This paper looks at the viability of extending a software engineering methodology to firmware engineering. In particular, it examines the first phase of the development procedure, the requirements engineering methodology, for producing more error-free, efficient firmware.

A decade ago user microprogramming was advocated [e.g. 22]. The idea was to tailor the computer architecture to suit a particular user application by way of firmware definition. It was an intriguing idea, but was a very difficult proposition because it was never clear how and when a computer is said to be tailores to a class of applications. This was not because the computer lacked this capability but because we were uncertain about the fundamental requirements of the applications and hence we did not know how to satisfy them. This was and still is a serious obstacle toward better utilization of a potentially powerful tool. If requirements were clearly and unambiguously stated and if they were adequately satisfied by a computer architecture, then it could be safely concluded that the computer was tailored to the application. This suggests a need for a firmware requirements engineering methodology.

Microprograms, or firmware, were originally conceived as a good alternative hardware design method that was a solution to many of the problems that the computer designers were facing. Its importance was recognized early by many innovative researchers in the field who subsequently formed a new workshop, the Annual Workshop on Microprogramming, as a ground for exchanging information on the up-to-date progress. Microprogramming once understood is a simple idea, however, and the basic research activities soon saturated the subject leaving more practical problems unresolved. Subsequently, with the advent of microprocessors, these practical problems became more important. Although many microprocessors were not necessarily firmware controll-

ed, the proliferation of firmware concept laid foundation for the ready acceptance of the amazing new product of the integrated hardware technology. The introduction of firmware-driven microprocessors in turn provided an ideal, low-cost opportunity to develop firmware for experimenting new ideas and for exploring new firmware applications. The increasing filmware development activities suggest a need for a firmware requirements engineering methodology.

The current software crisis provided an opportunity to re-assess the value of firmware. Presently a general belief is that the software complexity would be alleviated by increased hardware complexity. Past experiences suggest that this high expectation can be a fallacy [7, 14]. Indeed, "Two complexities don't make things simpler." [8]. A good solution (or compromise) appears to be the 'right' mixture of the three ingredients: hardware, firmware, and software. Undoubtedly, firmware will play a major role for alleviating the software crisis. In turn, this suggests a need for a firmware requirements engineering methodology.

Having established the need for a firmware requirements, this paper examines the applicability of software requirements engineering methodology to firmware. While software and firmware are inherently different, they share many common characteristics. Each has its own unique set of problems, but they often share many similar problems. After all, microprogramming, simply stated, is an adaptation of programming techniques to implementation of hardware control sequencing. Many techniques and solutions developed for software have been adapted to firmware in the past. If a requirements engineering methodology works well for software, it should likely be adaptable to firmware as well with adequate modification. Indeed there is no evil in borrowing and adapting a useful idea for a good cause. Maestro J.S. Bach[1] was a 'prolific borrower' of many delightful passages (including his own) and yet he is considered one of the greatest composers of all time [37].

[1] Johann Sebastian Bach, 1685–1750. A great German organist and composer.

2. Brief Survey of Requirements Specification Languages

This section briefly surveys the existing software requirements specification languages. Its scope is restricted to those that have been computer implemented or that are amenable to computer implementation. This restriction eliminated a number of graphic and flow diagram-type languages from consideration. CSC's Threads [36], NCR's ADS [21], and IBM's HIPO [29] are some of the examples. A graphic notation SADT [27] was included because of its potential use of computer database.

All the languages reviewed have been designed with specific emphasis. BDL [16], for example, is designed to express business information needs. Whereas PSL [34] and SADT are designed for practical problem solving, theoretical languages, such as HOS (Higher Order Software) [17], PRISM [19], FSM (Finite State Machine) [28], and V-graph [15] have an emphasis on correctness and consitstency verification. Theory-based languages afford preciseness for theoretical treatment of requirements such as formal proof of correctness, consistency, etc., but mapping of realistic concepts into formal notation is often Procrustean and unnatural for practical use. More extensive survey on software requirements specification languages may be found in [25, 33]. A comparative study by Davis and Vick [11] should provide more insight into the computer aided requirements analysis systems.

Of the three practical languages, SADT makes extensive use of graphic diagram notation along with footnotes [27]. It is an application of the principle of effective communication which expresses the why, what, and how of the proposed system. It allows many levels of system abstraction (or decomposition), which could easily disorient one unless he traces carefully which level of decomposition he is in. The computer implementation of diagrams and footnotes would not be straightforward. The Requirements Statement Language (RSL), on the other hand, is fully computer implemented [1, 5]. It is an outgrowth of PSL (Problem Statement Language) which was developed by Teichroew [34]. It is an English-like language that is fully supported by three software analysis

packages:

(1) production-driven RSL compiler;

(2) relational database for data extracted from RSL statements;

(3) requirement analyzer that verifies requirements consistency using the database and a simulator.

RSL is the input language for SREM, a requirements engineering system. In the sections that follow SREM and its two major components REVS and RSL will be described.

3. Software Requirements Engineering Methodology: SREM

A prerequisite to understanding the Firmware Requirements Engineering Methodology (FREM) described in this paper is the Software Requirements Engineering System (SREM) developed by TRW. The following sections summarize SREM and its components. More detailed description of SREM may be found elsewhere [1, 2, 3, 5].

3.1. SREM

SREM enables software requirements to be stated unambiguously in a concise English-like language, tested for consistency, and simulated for validity and feasibility. It was conceived as a means for improving the quality of software through better management of software design and development processes. It was developed by TRW Defense and Space Systems Group under the sponsorship of the Ballistic Missile Defense Advanced Technology Center. SREM is the name given to the total software system that consists basically of the Requirements Statement Language (RSL) which describes requirements and the Requirements Engineering and Validation System (REVS) which analyzes and validates the requirements stated in RSL, and the detailed sequence of steps and procedures for using RSL and REVS to incrementally define and verify processing requirements.

3.2. RSL

RSL is an English-like artificial language designed for stating software requirement concisely and precisely. It is a nonprocedural language with a small but useful vocabulary whose semantics are clearly defined. Limited vocabulary and usage constraints lead to unambiguous requirements representation for machine translation and minimize human misinterpretation of connotation.

Being a nonprocedural language, RSL has a simple syntax. It has a format that resembles repetitive declarations. It specifies all requirements using its four language primitives: elements, relationships, attributes, and structures. Table 1 lists the four primitives and their component concepts. The *elements* correspond closely to nouns in English. They are standard components that are used to describe requirements. The *relationships* are verbs. They associate two elements and specify the kind of association. The *attributes* are similar to adjectives in English. They are modifiers of elements and as such they define important properties of the elements. An attribute may be a name, number, or descriptive text. It may pertain to all element types or may be applicable only to a limited set of elements. The *structures* represent control flows of computation. Control flows are represented by the requirement networks, called R_NETS, which are an extension of a graph model of computation. Fig. 5 illustrates an example of an R_NET. The R_NET flow structure consists of nodes and arcs. Nodes represent processing operations including ALPHAs which specify complete functional processing steps and SUBNETs which also represent processing flows at a lower level. The nodes are connected by arcs. Except for the initial and terminal nodes, all procesing nodes have single-enty and single-exit arcs. Pseudo-nodes specify explicitly several types of fan-in and fan-out control flows. With these basic constructs, R_NETs model the stimulus-response control flows of software systems. R_NETs enable construction of more reliable and correct software by enforcing a discipline on the user and by increased testability.

An outstanding feature of RSL is its extensibility. It permits new concepts to be added to the existing set of basic concepts thereby adapting RSL to a new problem environment. The structures are not extensible, however. The judicious use of extensibility would make RSL applicable to a wide range of problems.

Table 1
f-RSL primitive language concepts

Elements	Relationships	Attributes	Structures
Existing RSL			
ALPHA	ASSOCIATES	ALTERNATIVES	R_NETS
DATA	COMPOSES	ARTIFICALITY	SUBNETS
DECISION	CONNECTS TO	BETA	VALIDATION_PATH
ENTITY_CLASS	CONSTRAINS	CHOICE	
ENTITY_TYPE	CONTAINS	COMPLETENESS	
EVENT	CREATES	DESCRIPTION	
FILE	DELAYS	ENTERED_BY	
INPUT_INTERFACE	DESTROYS	GAMMA	
MESSAGE	DOCUMENTS	INITIAL_VALUE	
ORIGINATING_ REQUIREMENT	ENABLES	LOCALITY	
	EQUATES TO	MAXIMUM_TIME	
OUTPUT INTERFACE	FORMS	MAXIMUM_VALUE	
PERFORMANCE_ REQUIREMENT	IMPLEMENTS	MINIMUM_TIME	
	INCLUDES	MINIMUM_VALUE	
R_NET	INCORPORATES	PROBLEM	
SOURCE	INPUTS	RANGE	
SUBNET	MAKES	RESOLUTION	
SUBSYSTEM	ORDERS	TEST	
SYNONYM	OUTPUTS	TYPE	
UNSTRUCTURED_ REQUIREMENT	PASSES	UNITS	
	RECORDS	USE	
VALIDATION PATH	SETS		
VALIDATION_POINT	TRACES TO		
VERSION			
Extensions			
STORAGE	SETS	DATA_SIZE	
FUNCTION_UNIT	RESETS	DATA_TYPE	
CLOCK	FETCHES	CYCLE_TIME	
INDICATOR	MULTIPLEXES	CONTROL_STORAGE	
COUNTER	TRANSFERS	REGISTER_FILE	
	PREFTCHES	MICROINSTRUCTION_SIZE	
	INCREMENTS	MICROINSTRUCTION_TYPE	
	DECREMENTS		

3.3. REVS

REVS is a set of software packages that automatically tests and analyzes requirements for completeness, consistency, and correctness. Structurally, it consists of the RSL translator, the Abstract System Semantic Model (ASSM), and the requirements analysis/validation aids (Fig. 1). The RSL translator translates requirements stated in RSL, extracts relevant information, and builds a database in ASSM. ASSM is a relational database that organizes requirements information extracted from the original statements by the translator. As a central repository for all information about requirements, it supports various analyses by providing necessary information. The production analysis/validation aids include the analysis reporting package, interactive graphic package, and simulator building package. The simulator in particular provides a dynamic environment for thorough requirements validation tests.

Currently, SREM is operational at several sites. It has been implemented on the large-scale computers such as the CDC 7600, 7700, the Cyber

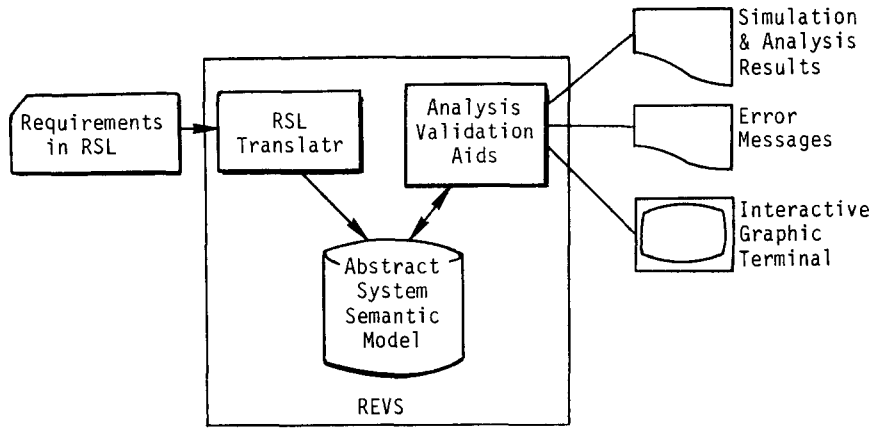

Fig. 1. The SREM Organization.

74/174, and the TI ASC [2, 3]. It has proven its practical importance for a number of software development projects and, as a result, it is being accepted for wider applications.

4. Requirements for Firmware Requirement Engineering

Presently firmware development is more an art than a science and no clear development methodology is available. The declined interest and lack of supporting tools in firmware development may be partially responsible for this. Consequently firmware requirement engineering has been mostly neglected. This is unfortunate because, when used judiciously, firmware would offer versatility and better performance. The current situations, in particular, the software crisis and proliferation of microprocessors, have created a good climate for the renewed interest in the practical applications of firmware which in turn should necessitate firmware requirement engineering.

A firmware requirement engineering methodology should provide a guideline for the development of reliable firmware. To be a practical methodology, it must incorporate tools that facilitate higher firmware productivity and testability. To develop the methodology, a set of necessary properties are established as the basic criteria. In addition to the usual criteria for requirement specification such as conciseness, unambiguity, understandability, etc., they include the following features:

(a) A means to represent the register-level hardware organization is necessary. It facilitates description of the base processor organization for which firmware is developed. A hardware description is used for consistency test of firmware which checks the firmware from assuming extraneous hardware features.

(b) The requirement specification should provide a clear guideline for the specification of system implementation and test procedures. By following the guideline, the designer is assured to satisfy the requirements.

(c) A high-level machine independent language is necessary for defining firmware procedures (i.e., ALPHA's). The firmware procedure definitions are used by the simulator for validation. Among a number of high-level microprogramming languages that have been designed and proposed, some of them may be adequate for this purpose. More widely accepted programming languages such as concurrent PASCAL may be useful also.

(d) An automatic test generator should enhance the correctness (i.e., reliability) of firmware. Generation of exhaustive test cases for firmware is a tedious and difficult task. However, the Ramamoorthy and Shankar method [24] that tests each path exhaustively should be useful if the firmware is represented by the loop-free R_NETs;

(e) A firmware simulator that simulates and

validates firmware definitions should be extremely valuable. It should verify both logical correctness and execution timing. Additionally, where feasible, it may facilitate software/firmware/hardware tradeoffs [29, 32].

This does not suggest that all the criteria be satisfied completely. Obviously, it would require massive efforts and extensive knowledge of the current state-of-the art of firmware engineering. These criteria are suggested as a measure for the completeness of methodology. Each firmware requirements engineering methodology has a different emphasis: however, to be an effective tool, it should incorporate all criteria to some degree.

5. Firmware requirements specification

FREM was conceived in anticipation of increasing firmware development activities. As it is an extension of SREM, the extended RSL will be referred to as firmware RSL or f-RSL, and the modified REVS as f-REVS in the subsequent discussions. While satisfying the criteria, efforts were made to minimize the amount of modifications. The original concepts and proposed extensions of f-RSL are listed in Table 1. The newly added concepts are primarily related to describing basic components and register organizations.

5.1. Elements

The firmware-level system constists of two basic types of components: storage and data path components. The storage components simply hold data. They include the accumulator, general-purpose registers, latches, status indicators, and other registers that are normally transparent to the programmer. In f-RSL the status indicators are distinguished from other registers since each of them convey certain designated system condition. The data paths transfer data from one storage to another and, in some cases, transform data during transition. The data paths include the adder, shifter, and other combinational networks as well as data transfer lines between registers. In order to facilitate more meaningful RSL statements, distinctions were made between combinational networks and data transfer lines. The data transfer lines are not defined explicitly as an element but their presence is implied by a relation CONNECTS_TO. FUNCTION_UNIT defines combinational networks.

5.2. Relationships

The relationship associates two elements. For firmware it specifies a mode in which data are transferred between STORAGE elements and between a STORAGE and a FUNCTION_UNIT. A pair of relationships, SETS and RESETS are used with INDICATORS whose states are determined by the values of certain specified data. FEATURES associates an element with its characteristics that are described by attributes.

5.3. Attributes

The attributes typically describe hardware characteristics, such as cycle time, number of registers, register size, and the like. BETA is a special attribute that defines precisely the operation of FUNCTION_UNIT using a programming language, PASCAL. The BETA definitions are used subsequently in system simulation.

5.4. Structures

The structure is not extensible in the sense that the other three RSL primitives are. In fact, R_NETS and ALPHAs are found to be adequately applicable to firmware requirements specification in their present forms. Although improvable, R_NETs, for example, would represent unambiguously the input-output and control relations among firmware routines that are described by ALPHAs. Each ALPHA that describes a firmware routine is in turn associated to a BETA that expresses precisely the ALPHA function using a programming language.

As stated earlier, f-RSL includes a minimum number of extentions that are considered necessary for specifying firmware requirements. By taking advantage of the RSL extensibility, other new concepts may be added as needed. Further details of f-RSL may be found in [26].

5.5. *Firmware Requirement Engineering and Validation*

The f-REVS analyzes the requirements stated in f-RSL in four steps: translation, decomposition, allocation, and analytical feasibility decomposition. The translation step insures the integrity of both the f-RSL syntax and the initial requirements specification by performing the following tasks:

(1) translation of the f-RSL-stated requirements using the productions-based compiler building system;

(2) extraction of the stated requirements;

(3) creation of a database which contains the summary of the original requirements;

(4) generation of data and structural information for requirements traceability;

(5) analysis of the requirements for consistency and completeness;

(6) generation of 'error messages', if any.

The decomposition step adds detail to the specified requirements. It would decompose a requirement into smaller, more specific ones and attempt to quantify them. As a result, more refined, quantitative requirements are generated. The allocation step performs sensitivity analyses to select the tolerance values for performance. It allocates performance values to each of the processing paths which will provide information for validation tests in a subsequent simulation. This step generates a complete set of processing requirements. The analytical feasibility demonstration step builds an analytical simulator consisting of a set of algorithm packages with input, output, and processing requirements. The simulator can be driven with realistic interface data produced by a simulation of the environment. It demonstrates analytical feasibility thereby reducing the complexity of the subsequent process design phases. Upon completion of these four steps, the f-REVS generates a set of documents that describe various requirements in detail.

6. FREM Requirements Specification Procedure: An Example

The FREM requirements specification procedure will be illustrated by the hospital patient monitoring system described in [31]. The problem is stated as follows [31]:

"A patient monitoring program is required for a hospital. Each patient is monitored by an analog device which measures factors such as pulse, temperature, blood pressure, and skin resistance. The program reads these factors on a periodic basis (specified for each patient) and stores these factors in a database. For each patient, safe ranges for each factor are specified (e.g., patient X's valid temperature range is 98 to 99.5 degrees Fahrenheit). If a factor falls outside of a patient's safe range, or if an analog device fails, the nurse's station is notified."

The hardware organization of the patient monitoring system is shown in Fig. 2. To satisfy the real-time urgency requirements, the monitoring program will be firmware implemented. The firmware is assumed to be developed for the processor with an 8080-type register organization illustrated in Fig. 3. The control storage and the firmware execution unit are not shown in the figure.

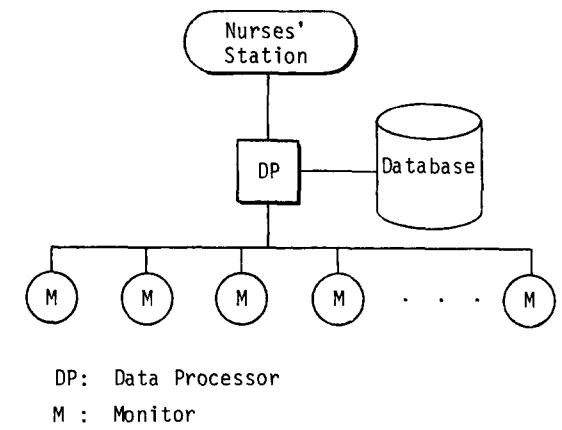

Fig. 2. The Hospital Patient Monitoring System.

The f-RSL statements for firmware requirement specification are shown in Fig. 4. The figure illustrates the use of some of the newly extended RSL primitives, e.g., FUNCTION_UNIT, STORAGE, and the message/data to be transferred between them. It should also convey some general characteristics such as preciseness,

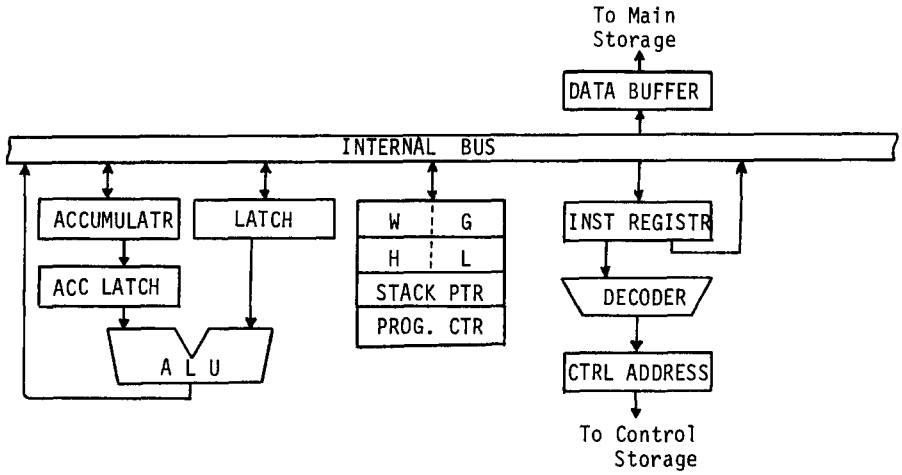

Fig. 3. Intel 8080-Type Register Organization.

```
FUNCTION_UNIT:  ALU.
   CONNECTED_TO:  STORAGE LATCH.
      INPUTS:  MESSAGE OPERANDS
         MADE_BY:  DATA DEVICE_DATA.
            INCLUDES:  DATA DEVICE_NUMBER
                       DATA DEVICE_STATUS
         MADE_BY:  DATA PATIENT_FACTOR.
            INCLUDES:  DATA PULSE
                       DATA TEMPERATURE
                       DATA BLOOD_PRESSURE
                       DATA SKIN_RESISTANCE.
STORAGE:  REGISTER FILE.
   CONNECTED_TO:  STORAGE DATA_BUFFER.
      OUTPUTS:  FILE HISTORY_FILE.
         CONTAINS:  DATA PATIENT_ID
                    DATA MEASUREMENT_TIME
                    DATA PULSE
                    DATA TEMPERATURE
                    DATA BLOOD_PRESSURE
                    DATA SKIN_RESISTANCE.
         TRACED_FROM:  SENTENCE_2
                       SENTENCE_3.
         ASSOCIATED_WITH:  ENTITY CLASS PATIENT.
   CONNECTED_TO:  STORAGE ACCUMULATOR.
      OUTPUTS:  MESSAGE OPERANDS.
            . . . .
ALPHA:  SEND_MONITOR_FAILURE_MESSAGE.
   INPUTS:  INVALID_DATA.
   OUTPUTS:  MONITOR_FAILURE_MESSAGE.
   DESCRIPTION:  "WHEN INVALID DEVICE DATA ARE FOUND, NOTIFY THE NURSE'S
                  STATION OF MONITOR FAILURE."
ALPHA:  COMPARE_PATIENT_FACTOR.
   INPUTS:  PATIENT_FACTOR.
   OUTPUTS:  OUT_OF_RANGE_MESSAGE.
   DESCRIPTION:  "COMPARE PATIENT'S FACTORS TO PRESCRIBED RANGES.
                  IF A FACTOR FALLS OUTSIDE THE RANGE, SEND A WARNING
                  MESSAGE TO THE NURSE'S STATION."
VALIDATION_PATH:  MEASUREMENT_OUT_OF_RANGE.
   PATH:  VALIDATION_POINT V1, VALIDATION_POINT V5.
```

Fig. 4. RSL Partial Definition for the Patient Monitor System.

repetitiveness, and certain awkwardness of f-RSL. The statements define the connections between FUNCTION_UNITs and STORAGEs as well as the message/data that are transferred between them. Although not shown, functions for FUNCTION_UNITs and STORAGEs can be defined explicitly. An ALPHA is a processing element and defines firmware functional requirements: its input and output data and a functional description in plain English. In the figure, ALPHAs for SEND_MONITOR_FAILURE_MESSAGE and COMPARE_PATIENT_FACTOR are defined as an example. Their names correspond to those in the R_NET.

A graphical representation of the R_NET for the hospital patient monitoring system is shown in Fig. 5. The R_NET identifies individual processing elements (i.e., ALPHAs), the data/control flow between them, and a series of consistency and completeness validation points. It permits static tests for consistency of functional requirements. The dynamic consistency of functional requirements is verified by simulations that are generated by the f-REVS. When all paths in the R_NET have been validated, performance requirements such as accuracy, execution time, etc. are derived. The feasibility of the performance requirements may be supported by the availability of hardware technology, algorithms, etc.

Note that the R_NET in Fig. 5 represent a macro-level view of control flows among firmware procedures. Although it has not been fully tested yet, the R_NET may facilitate a machine-independent representation of micro-level control flows of individual ALPHAs. That is, a sub-R_NET may be embedded in the macro-level R_NETs. At the micro-level, an ALPHA in the sub-R_NET would represent a mircooperation and the sub-R_NET may be considered to represent a maximum parallelism [23] of a microprogram.

The processing steps of ALPHA are defined by a BETA definition using the PASCAL programming language. The input/output specifications in the R_NET must be strictly adhered in the BETA definition. The BETA definitions in PASCAL

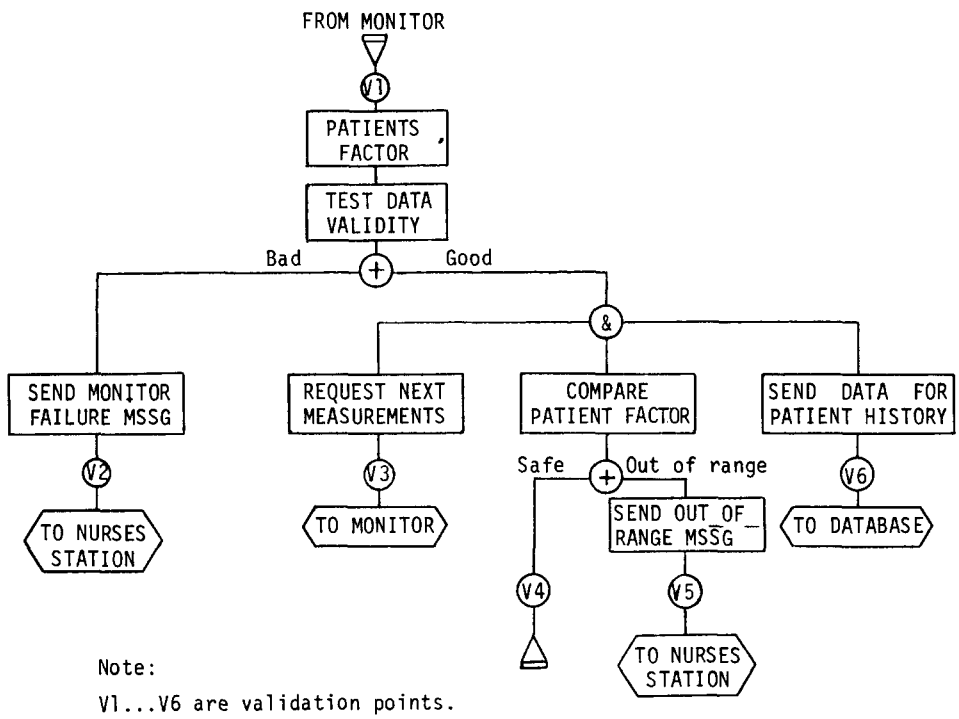

Fig. 5. R_NET for the Patient Monitor Program.

enables the f-REVS to simulate ALPHAs and R_NETs for validation of completeness, consistency and correctness. The f-REVS analyses would generate several useful reports including a list of requirements information cross referenced for ALPHAs, DATA types, their value ranges, performance requirements, validation point, etc. as well as error messages, if any. Upon comletion of requirement validation, ALPHAs and hence R_NETs are ready for design and implementation. This is where FREM's domain ends. The design and implementation are machine dependent parts of firmware developments and related decisions such as choice of register organization, horizontal vs. vertical microcode, ROM or writable control store, implementation methods, etc. belong to the designer and engineers. With clearly defined, well-documented requirements at hand, sound decisions should be made for the choice.

FREM has not proven its practical value, but is satisfies to a large extent all of the requirements stated in Section 4. Its usefulness may be demonstrated relatively easily by extending the existing SREM and by specifying practical firmware development requirements.

7. Future Improvements

Improvements are considered primarily for f-REVS. One major drawback of the current FREM is that f-REVS lacks facility for analyzing software-firmware and hardware-firmware tradeoffs. The tradeoff decisions would likely be made at the time of design and implementation. However, the f-REVS simulator should provide relevant information about the tradeoff effects. Such information should be valuable for the subsequent system development phases. The performance evaluation models for vertical migration described in [29, 32] could be made practical for the hardware/firmware/software tradeoff study.

The BETA definitions for ALPHAs may be specified in concurrent PASCAL. Where fine detail of firmware control simulation is necessary, PASCAL is not quite adequate. A comprehensive and concise language is necessary for expressing sufficient detail and concurrency of firmware control sequences. A good body of literature is available on microprogramming languages [4, 6, 13, 18, 20, 22]. A candidate language should permit representation of concurrent microoperation and allow f-REVS to simulate horizontally coded firmware.

f-REVS should perform some rudimentary microcode optimization. Microcode optimization is in fact a machine dependent practice performed during implementation. It is proposed here because it often reveals a match/mismatch of microcode (and hence the type of application) to the underlining hardware architecture. Although microcode optimization is an NP-complete problem as first suggested by Tsuchiya and Gonzalez [35] and later proven by DeWitt [12], a good number of heuristic algorithms are available [9].

Other improvements could be considered but FREM with these improvements should prove to be a valuable tool for firmware requirements engineering and should enhance the quality of firmware.

8. Summary

FREM, an extension of SREM, for firmware requirements engineering is presented. It was conceived on the hypothesis that software requirements specification methods should be applicable to a large extent to firmware as well. As the results have shown, FREM confirms this hypothesis.

FREM consists of f-RSL and f-REVS which are extensions of RSL and REVS, respectively. Upon a brief survey of the requirements statement languages, RSL with its extensibility was found to be most readily adaptable to the firmware environment. A minimum number of extensions have been added to the existing RSL concepts thereby turning RSL into f-RSL. The utility of f-RSL for describing firmware requirements has been shown using a hospital patient monitoring system as an example.

REVS is a powerful analysis tool that validates completeness, consistency, and correctness of software requirements. Mutation of REVS into f-REVS requires a few additional improvements.

These improvements are necessary to deal with some problems that are unique to firmware engineering. They include, among others, tradeoff analyses, a firmware representation and simulation language, and firmware optimization.

FREM with a minimum of modifications has proven to be useful for firmware requirements engineering. For wider acceptance, it has to demonstrate its practicality on an actual firmware development. It should be a valuable tool for improving the quality of firmware development, and should encourage firmware development projects. It is conceivable that, with the availability fo this powerful developmental tool, firmware development activities should increase. In turn firmware should play a larger role in replacing conventional software function. This, we hope, would alleviate many of the software problems without added complexity in hardware.

References

[1] M.W. Alford, A requirements engineering methodology for real time processing requirements, IEEE Trans. on Software Engineering SE-3 (1) (1977) 60–69.

[2] M.W. Alford, Software requirements engineering methodology (SREM) at the age of two, Proceedings COMPSAC 78 (1978) 332–339.

[3] M.W. Alford, Software requirements engineering methodology (SREM) at the age of four, Proceedings COMPSAC 80 (1980) 866–874.

[4] C.G. Bell and A. Newell, Computer structures: Readings and examples (McGraw Hill, New York, 1971).

[5] T.E. Bell et al., An extendable approach to computer aided software requirements engineering, IEEE Trans. on Software Engineering, SE-3 (1) (1977) 49–60.

[6] Y. Chu, Introduction to Computer Organization (Prentice Hall, Englewood Cliffs, NJ, 1970).

[7] H.C. Cragon, A case against high-level language computer architecture, Proc. Int. Workshop on High-Level Language Computer Arch. (1980) pp. 88–91.

[8] H.C. Cragon, A comment made at the International Workshop on High-Level Language Computer Arch., Ft. Lauderdale, FL (May 1980).

[9] S. Dasgupta, The organization of microprogram stores, ACM Computing Survey 11 (1) (1979) 39–65.

[10] S. Davidson and B.D. Shriver, An overview of firmware engineering, Computer 11 (5) (1978) 21–33.

[11] C.G. Davis and C.R. Vick, The software development system, IEEE Trans. on Software Engineering, SE-3 (1) (1977) 69–84.

[12] D.J. DeWitt, A machine independent approach to the production of optimal horizontal microcode, Dissertation, Univ. of Michigan, Ann Arbor, MI (1976).

[13] D. Dietmeyer, Logical Design of Digital Systems (Allyn & Bacon, Boston, IL, 1971).

[14] D.R. Ditzel and W.A. Kwinn, Reflections on a high-level language computer system or parting thoughts on the SYMBOL project, Proc. Int. Workshop on High-Level Language Computer Arch. (1980) pp. 80–87.

[15] J. Earley, Toward an understanding of data structures, Comm. Assoc. Comput. Mach. 14 (10) (1971) 617–627.

[16] P.C. Goldberg, Structured programming for non-programmer, IBM Report RC-5318 (March 1975).

[17] M. Hamilton and S. Zeldin, Higher order software – A methodology for defining software, IEEE Trans. on Software Engineering SE-2 (1) (1976) 9–32.

[18] F. Hill and G. Peterson, Digital Systems: Hardware Organization and Design (Wiley, Englewood Cliffs, NJ, 1973).

[19] T.I.M. Ho and J.F. Nunamaker Jr., Requirements statement language principles for automatic programming, Proc. ACM National Conf. (1974) pp. 279–288.

[20] K.E. Iverson, A Programming Language (Wiley, New York, 1962).

[21] H.J. Lynch, ADS: A technique in system documentation, Database 1 (1) (Spring 1969) 6–18.

[22] C.V. Ramamoorthy and M. Tsuchiya, A study of user microprogrammable computers, in: 1970 Spring Joint Computer Conf., AFIPS Conf. Proc. (1970) pp. 165–181.

[23] C.V. Ramamoorthy and M. Tsuchiya, A high-level language for horizontal microprogramming, IEEE Trans. on Computers C-23 (8) (1974) 791–801.

[24] C.V. Ramamoorthy and K.S. Shankar, Automatic testing for the correctness and equivalence of loop-free microprograms, IEEE Trans. on Computers C-23 (8) (1974) 768–783.

[25] C.V. Ramamoorthy and H.H. So, Software requirements and specifications: Status and perspectives, in: M.P. Mariani and D.F. Palmer (eds.), Tutorial: Distributed System Design (IEEE Computer Society Publ., Long Beach, Ca, 1979).

[26] Requirements engineering for distributed data processing systems, Final Report, TRW (June 1980).

[27] D.T. Ross, Structured analysis (SA): A language for communicating ideas, IEEE Trans. on Software Engineering, SE-3 (1) (1977) 16–34.

[28] K. Salter, A methodology for decomposing system requirements into data processing requirements, Proc. Second Int. Conf. on Software Eng. (October, 1976).

[29] J.A. Stankovic, The types and interactions of vertical migrations of functions in a multilevel interpretive system, IEEE Trans. on Computers C-30 (7) (1981) 505–513.

[30] J.F. Stay, HIPO and integrated program design, IBM Systems Journal 15 (2) (1976) 143–154.

[31] W.P. Stevens, G.F. Myers and L.C. Constantine, Structured design, IBM Systems Journal 13 (2) (1974) 115–139.

[32] J. Stockenberg and A. van Dam, Vertical migration for performance enhancement in layered hardware/firmware/

software systems, Computer 11 (5) (1978) 35–50.

[33] D. Teichroew, A survey of languages for stating requirements for computer-based information systems, in: 1972 Fall Joint Computer Conf., AFIPS Conf. Proc. 41 (1972) 1203–1224.

[34] D. Teichroew and E.A. Hershey III, PSL/PSA: A computer-aided technique for structured documentation and analysis of information processing systems, IEEE Trans. on Software Eng. SE-3 (1) (1977) 41–48.

[35] M. Tsuchiya and M.J. Gonzalez, Toward optimization of horizontal microprogramming, IEEE Trans. on Computers C-23 (8) (1974) 791–801.

[36] A users guide to the threads management system, Computer Science Corp. (November 1973).

[37] N. Carrell, Bach the Borrower (George Allen & Unwin, London, 1967)

M. Tsuchiya received the B.S. degree in management information systems from Konan University, Kobe, Japan, and the Ph.D. degree in computer sciences from the University of Texas at Austin, Texas.

He is currently with TRW Inc., Redondo Beach, California. Prior to joining TRW, he was a faculty member at the University of Hawaii at Manoa, the University of California, Irvine, and Northwestern University, Evanston, Illinois. In 1975, he was a visiting computer scientist at Datalogisk Afdeling, Aarhus Universitet, Aarhus, Denmark, and, in 1972, he was a visiting lecturer ar Konan University, Konan, Japan. His research interests include computer architecture, distributed processing systems, and database systems.