Functional Documents for Computer Systems

David Lorge Parnas
Telecommunications Research Institute of Ontario (TRIO)
Department of Electrical and Computer Engineering
McMaster University, Hamilton, Ontario, Canada L8S 4K1

Jan Madey
Institute of Informatics, Warsaw University
Banacha 2, 02-097 Warsaw, Poland

Abstract

Although software documentation standards often go into great detail about the format of documents, describing such details as paragraph numbering and section headings, they fail to give precise descriptions of the information to be contained in the documents. This paper does the opposite; it defines the contents of documents without specifying their format or the notation to be used in them.

We describe documents such as the "System Requirements Document", the "System Design Document", the "Software Requirements Document", the "Software Behaviour Specification", the "Module Interface Specification", and the "Module Internal Design Document" as representations of one or more mathematical relations. By describing those relations, we specify what information should be contained in each document.

1 Introduction

Engineers are expected to make disciplined use of science, mathematics and technology to build useful products. Those who construct computer systems are clearly Engineers in that sense. However, the design process used for computer systems is very different from that used in developing other engineering products. Professional Engineers make extensive use of mathematics to provide precise descriptions of their products. In contrast, computer systems are usually described, inaccurately and imprecisely, using anthropomorphic analogies and intuitive language. When Engineers assemble a system, they recognise the necessity for precise interface specifications for each of the components. In contrast, computer system designers, particularly programmers, rarely write such specifications; instead they rely on intuitive descriptions. The result is a "cut-and-try" process in which substantial redesign must be done during "system integration".

This paper advocates making computer systems design more like other types of engineering. Applying a mix of standard engineering and mathematical concepts, we show how essential properties of computer systems, and their components, can be described as a set of mathematical relations. By describing these relations, computer systems designers can document their designs systematically, and use that documentation to conduct thorough reviews. A project that involved the use of such documents in formal reviews is described in [38, 39, 40].

This paper discusses a limited set of professional reference documents. It does not discuss many useful types of documents such as user-manuals or queuing-theory models.

Accepted for publication in *Science of Computer Programming* (Elesevier) 1995.

Our goal, in this paper, is to describe the contents of key computer systems documents – not their form. To do this we define documents in terms of the functions that they must describe, but we will not say much about the way those functions should be represented. We consider it important to agree on content before discussing the notation to be used. This is in sharp contrast to other papers on “formal methods”, which provide precise specifications of syntax but give short shrift to the question of what information should be provided using that syntax. Some companion reports discuss notations that can be used to provide more easily read representations of the functions [17, 30, 36, 43, 44].

In this paper, we discuss documentation at a high level of abstraction, dealing uniformly with many types of systems and documents. For a specific application, one must apply these concepts with a specific notation (as was done in [7]). Because of the breadth of this paper, discussions of case studies has been relegated to other papers cited throughout the text. Comparison with other notations, intended for more specialised applications, is best done in papers with a more limited scope.

This paper is addressed to a mixed audience. We want it to be understood by Engineers educated in such fields as Chemical Engineering or Mechanical Engineering as well as by Computer Scientists who are interested in providing tools for Engineers. The first group may have no familiarity with concepts and notation that are standard in abstract mathematics. The second group will find approaches based on control theory or circuit theory to be new. To make sure that everyone can stay with us, we include some basic definitions that will be familiar to many.

2 What should be the role of documentation in computer system design?

The production of design documents plays a key role in engineering practice. Professional Engineers do not build a product without putting detailed plans on paper. Documents are prepared, and thoroughly analysed, before construction is begun and used as references throughout further design and construction.

While one cannot prescribe procedures for designing ([42]), we can establish criteria to be used when evaluating designs. The validation of design documents, using preestablished criteria, is a major part of engineering. The inspiration that leads to a design may occur in a flash, but the analysis required to confirm the workability and safety of the approach takes much longer. That analysis is the difference between professional engineering and amateurish invention.

Design validation is a technical task that can only be carried out if the design documentation is precise enough to permit systematic analysis. Conventional engineering documents are sufficiently detailed that one can calculate pressures, derive differential equations, compute load factors, and find resonant frequencies from the information that they contain.

The way that computer systems, particularly the software parts, are documented is very different. Many of the documents produced in advance of implementation are not technical ones; they are narratives explaining the role of the system, scenarios describing how it might be used, or glowing descriptions of the attractive qualities it will have. Little technical evaluation can be done on the basis of such documents. Other documents provide precise analyses of abstract algorithms but ignore much of the actual code. Real evaluation of the overall design must wait until the implementation is nearly complete. At that point, corrections are much more difficult (and expensive) than they would have been at an earlier date.

In many software development organisations, documentation is not viewed as part of the design activity but as an additional, somewhat distasteful, task that must be completed because of bureaucratic regulations. Often the programs are written before the documentation; frequently the documentation is written by a separate group, one that does not include the designers. Usually, programmers consider the pre-implementation documentation vague and nearly useless. Consequently, computer system design documents are usually inaccurate when first delivered and are rarely kept up-to-date. The exceptions are, almost always, hardware documents. Hardware is designed by Engineers, who have been taught the importance of technical documentation.

Inadequate documentation causes software quality to degrade over time because the changes are inconsistent with the original (undocumented) design concept; such changes result in unnecessarily complex programs [31].

We believe that standard engineering practice can be applied in all phases of computer system design. Documentation can be used both as a design medium and as the input to subsequent analysis and testing activities [12]. We view the documentation as being (at least) as important as the product itself; if there is good documentation, a software product can be revised or replaced relatively quickly; without good documentation, software products are of questionable long-term value.

We do not suggest that the documents be completed in the order presented here. All documents should be revised repeatedly as the project progresses and understanding deepens. It is vital that documentation be kept "alive", that is up-to-date and consistent with the current state of the product itself.

3 What do we mean by "functional"?

In this paper we describe an approach that we call "functional", although "relational" might be more accurate[1]. It is important to note that we are not using "functional" in its vernacular sense, but with its standard mathematical meaning. In the vernacular "function" often means purpose, role, or activity, as in: "The function of this system is to control the level of water in the tank". In mathematics, *function* means a mapping between two sets of elements (called *domain* and *range*, respectively) such that every element of the domain is mapped to exactly one element in the range. If the latter condition is not satisfied, the mapping is called a (binary) *relation*[2]. In certain applications, it is useful to combine a relation with an additional set, a subset of the relation's domain, known as the *competence set*, to form what we call a *limited domain relation* (in short: LD-relation) [27, 43, 44]. We use the term *functional* to denote approaches based on any of these concepts.

The conventions below will be used inn the sequel.

Let R be a (binary) relation, then:

(1) "R" denotes the set of ordered pairs that constitutes the relation,

(2) "domain(R)" denotes the set of values that appear as the left element of a pair in R,

(3) "range(R)" denotes the set of values that appear as the right element of a pair in R,

(4) "$R(x,y)$" denotes a predicate, the *characteristic predicate* of the set R; $R(a,b)$ is true if and only if $(a,$b) is a member of R,

(5) when R is a function, "$R(x)$" denotes y such that $R(x,y)$.

It is important to note that the elements in the range and domain of a relation need not be scalars. The universes from which these sets are drawn may include vectors of scalar values, functions, and vectors of

[1] We avoid using "relational" to prevent confusion with the data base use of that term.

[2] We treat a function as a special case of a relation. Alternatively, a relation can be treated as a function mapping elements of the relation's domain to subsets of the relation's range.

functions. Of particular interest are *time-functions*, functions whose domain is a set of real numbers interpreted as representing time. By using time-functions in the range and domain of our functions, it is possible to specify critical characteristics of real-time systems.

Specifying real-time properties by means of relations between time-functions is the traditional approach to dealing with time and has long been used, routinely, by Engineers who design electronic circuits, control systems, etc. We believe that those who argue that we need new kinds of logic to deal with real-time systems, should be explaining why they cannot use an approach that has been taught to Engineers for many years. As we show in sections 4.2 - 4.5, these "classic" approaches allow us to talk about real-time properties in a precise way.

We can illustrate the generality of our approach using the familiar area of program semantics. As Mills [20] and others have pointed out, the semantics of a deterministic sequential program can be described by a single function. In that paper, Mills says little about the representation of those functions, but elsewhere (e.g. [21]) he suggests that we describe them using concurrent-assignment statements. However, there many other ways to represent that function. Dijkstra's weakest precondition predicate transformer [6], the precondition-postcondition pairs used by many other authors, and the notation introduced by Hehner [9] all provide the necessary information. Each of these notations has advantages and disadvantages; there is no single representation that is ideal for the broad class of functions involved. The first step towards precise practical documentation must be the identification of the relations that should be described in those documents. Notation for the description of those relations can be expected to develop further as our profession matures.

4 What information should be provided in computer system documentation?

Because computer systems are complex products, they require a great deal of documentation. In Section 4.1 we describe the purpose of the most important documents; subsequent sections provide more detailed descriptions. The documents we discuss are the ones mentioned in most company and industry standards. The division into documents is intended to provide "separation of concerns". Each document is aimed at a different audience, i.e. Engineers with specialised interests. Our purpose is to replace the usual vague, "common sense", descriptions of these documents by precise definitions.

4.1 What documents are needed? - an overview

The *System Requirements Document* treats the complete computer system (the computers and all associated peripheral devices) as a "black-box". It must include a description of the environment that identifies a set of quantities of concern to the system's users and associates each one with a mathematical variable. It must describe the relationships between the values of these quantities that result from physical (or other) constraints, as well as the additional constraints on the values of the environmental quantities that are to be enforced by the new system. Because this is a "black-box" description, any method used to document the system requirements should be as applicable to systems built of analogue components and relays as to systems using digital computers. The analysis of a network comprising computers and other components is much easier if the same notation and concepts are used throughout (cf Section 4.2).

The *System Design Document* identifies the computers within the system and describes how they communicate; it must also include a precise description of the relevant properties of the peripheral devices. The values in each computer's input and output registers are denoted by mathematical variables; the system design document defines the relationship between these values and the values of the environmental quantities identified in the systems requirements document (cf Section 4.3).

The *System Requirements Document* and the *System Design Document* determine the software requirements. Together, these two documents may serve as the *Software Requirements Document* as described in [47, 48].

It is often the case that the requirements do not fully determine the software behaviour. There may be many externally distinguishable software products that would satisfy the system requirements. In such a situation, many project managers may request an additional document, the *Software Behaviour Specification*, which records additional design decisions, and provides a description of the actual behaviour of the software. Sections 4.4 and 4.5 discuss these issues in more detail. See [10] and [11] for examples and another discussion of these documents.

Software projects are usually organised as a set of work assignments. Each assignment is to produce a group of programs, which we call a *module*. The *Software Module Guide* is an informal document that describes the division of the software into modules by stating the responsibilities of each. Together, the modules described in the module guide should satisfy the stated requirements. Software module guides are described at length in [4, 41].

For each module listed in the software module guide, there should be a *Module Interface Specification*. This should treat the module as a black-box, identifying those programs that can be invoked from outside the module, which we call the *access-programs*, and describing the externally-visible effects of using them (cf Section 4.6).

Software products should also be described by a *Uses-relation Document*. The range and domain of the “uses” relation are subsets of the set of access-programs of the modules. A pair (P,Q) is in the relation if program P uses program Q. This document, which often consists of a binary matrix, constrains the work of the programmers and determines the viable subsets of the software. This document has been discussed in [25, 26].

For each implementation of a module specification there should be a *Module Internal Design Document*. This document must be sufficiently precise that one can use it, together with the module interface specification, to verify the workability of the design. It should describe the module’s data structure, state the intended interpretation of that data structure (in terms of the external interface), and specify the effect of each access-program on the module’s data structure (cf Sections 4.7 and 4.8). This is sometimes called a “clear-box” description [22].

A different overview of a system can be obtained by documenting the “data flow” between variables or between communicating sequential processes. This can be included in *a Data-flow Document* (cf Section 4.9).

Communication between computers requires the establishment of communications protocols. It is absolutely essential that the communications services provided by these protocols be completely defined and that the protocols themselves are documented with a precision that allows testing and verification. This can be done with two documents, the *Service Specification* and the *Protocol Design Document* (cf Section 4.10).

At the lowest level of a computer system we find the hardware chips. Unlike the simple devices used in early computers, modern chips can include substantial amounts of memory and complex decision making logic. A black-box description of the behaviour of these chips, the *Chip Behaviour Specification*, would serve to define the interface for both the chip designers and those who integrate the chip into the rest of the system (cf Section 4.11).

Earlier discussions of these documents were contained in [33, 34, 35].

4.2 How can we document system requirements?

A critical step in documenting the requirements of a computer system is the identification of the environmental quantities to be measured or controlled and the representation of those quantities by mathematical variables. The environmental quantities include: physical properties (such as temperatures and pressures), the readings on user-visible displays, administrative information, (such as the number of people assigned to a given task), and even the wishes of a human user. These quantities must be denoted by mathematical variables and, as is usual in engineering, that association must be carefully defined, coordinate systems, signs etc. must be unambiguously stated. Often diagrams are essential to clarify the correspondence between physical quantities and mathematical variables.

It is useful to characterise each environmental quantity as either monitored, controlled, or both. *Monitored* quantities are those that the user wants the system to measure. *Controlled* quantities are those whose values the system is intended to control[3]. For real-time systems, time can be treated as a monitored quantity. In the sequel, we will use “m_1”, “m_2”, …, “m_p” to denote the monitored quantities, and “c_1”, “c_2”, …, “c_q” to denote the controlled ones. If the same quantity is to be both monitored and controlled, the corresponding values must be specified to be equal (by relation NAT, cf Section 4.2.1 below). Note too that some systems are expected to monitor the status of their own (internal) hardware components. Since these are often determined later in the design process, we may add to the set of monitored variables long after design has begun.

Each of these environmental quantities has a value that can be recorded as a function of time. When we denote a given environmental quantity by “v”, we will denote the time-function describing its value by “v^t”. Note that v^t is a mathematical function whose domain consists of real numbers; its value at time t is denoted by “v^t(t)”.

The vector of time-function $(m_1^t, m_2^t, ..., m_p^t)$ containing one element for each of the monitored quantities, will be denoted by “$\underset{\sim}{m}^t$”; similarly $(c_1^t, c_2^t, ..., c_q^t)$ will be denoted by “$\underset{\sim}{c}^t$”.

4.2.1 The Relation NAT

The environment, i.e. nature and previously installed systems, place constraints on the values of environmental quantities. These restrictions must be documented in the Requirements Document and may be described by means of a relation, which we call NAT (for nature), defined as follows:

— domain(NAT) is a set of vectors of time-functions containing exactly the instances of $\underset{\sim}{m}^t$ allowed by the environmental constraints,

— range(NAT) is a set of vectors of time-functions containing exactly the instances of $\underset{\sim}{c}^t$ allowed by the environmental constraints,

— $(\underset{\sim}{m}^t, \underset{\sim}{c}^t) \in$ NAT if and only if the environmental constraints allow the controlled quantities to take on the values described by $\underset{\sim}{c}^t$, if the values of the monitored quantities are described by $\underset{\sim}{m}^t$.

[3] Frequently, it is not possible to monitor or control exactly the variables of interest to the user. Instead one must monitor or control other variables whose values are related to the variables of real interest. Usually, one obtains the clearest and simplest documents by writing them in terms of the variables of interest to the user in spite of the fact that the system will monitor other variables in order to determine the value of those mentioned in the document.

NAT is not usually a function; if NAT were a function the computer system would not be able to vary the values of the controlled quantities without effecting changes in the monitored quantities. Note that if any values of $\underset{\sim}{m}^t$ are not included in the domain of NAT, the system designers may assume that these values will never occur.

4.2.2 The Relation REQ

The computer system is expected to impose further constraints on the environmental quantities. The permitted behaviour may be documented by describing a relation, which we call REQ. REQ is defined as follows:

— domain(REQ) is a set of vectors of time-functions containing those instances of $\underset{\sim}{m}^t$ allowed by environmental constraints,

— range(REQ) is a set of vectors of time-functions containing only those instances of $\underset{\sim}{c}^t$ considered permissible, i.e. values that would be allowed by a correctly functioning system.

— $(\underset{\sim}{m}^t, \underset{\sim}{c}^t) \in$ REQ if and only if the computer system should permit the controlled quantities to take on the values described by $\underset{\sim}{c}^t$ when the values of the monitored quantities are described by $\underset{\sim}{m}^t$.

REQ is usually not a function because one can tolerate “small” errors in the values of controlled quantities.

4.2.3 Requirements feasibility

Because the requirements should specify behaviour for all cases that can arise, it should be true that,

— domain(REQ) $\supseteq$ domain(NAT),(1)

The relation REQ can be considered *feasible with respect to NAT* if (1) holds and

— domain(REQ $\cap$ NAT) = (domain(REQ) $\cap$ domain(NAT)).(2)

Feasibility, in the above sense, means that nature (as described by NAT) will allow the required behaviour (as described by REQ); it does not mean that the functions involved are computable or that an implementation is practical.

Note that (1) and (2) imply that,

— domain(REQ $\cap$ NAT) = domain(NAT).(3)

Discussions of the use of this model in practice can be found in [1, 10, 11, 12, 38, 39, 40]. Further examples and discussion can be found in [47, 48].

4.3 How can we document system design?

During the system design two additional sets of variables are introduced: one represents the *inputs*, the values actually stored in the input registers of the computers in the system; the other represents the *outputs*, the contents of the output registers of those computers. Their values will also be described by time-functions. Note that $\underset{\sim}{m}^t$ and $\underset{\sim}{c}^t$ are defined as in Section 4.2. Below we describe how to document the meaning of these new variables by giving the relation between their values and those previously introduced.

4.3.1 The Relation IN

Let “$\underset{\sim}{i}^t$” denote the vector $(i_1^t, i_2^t, ..., i_r^t)$ containing one element for each of the input registers. The physical interpretation of the inputs can be described by a relation, IN, defined as follows:

— domain(IN) is a set of vectors of time-functions containing the possible instances of $\underset{\sim}{m}^t$; it must include domain(NAT).

— range(IN) is a set of vectors of time-functions containing possible instances of $\underset{\sim}{i}^t$,

— $(\underset{\sim}{m}^t, \underset{\sim}{i}^t) \in$ IN if and only if $\underset{\sim}{i}^t$ describes values of the input registers that are possible if $\underset{\sim}{m}^t$ describes the values of the monitored quantities.

IN describes the behaviour of the input devices. IN is a relation rather than a function as a result of imprecision in the measurement and transducer devices.

4.3.2 The Relation OUT

Let “$\underset{\sim}{o}^t$” denote the vector $(o_1^t, o_2^t, ..., o_s^t)$ containing one element for each of the output registers. The effects of the output devices can be described by a relation, OUT, defined as follows:

— domain(OUT) is a set of vectors of time-functions containing the possible instances of $\underset{\sim}{o}^t$,

— range(OUT) is a set of vectors of time-functions containing possible instances of $\underset{\sim}{c}^t$,

— $(\underset{\sim}{o}^t, \underset{\sim}{c}^t) \in$ OUT if and only if $\underset{\sim}{c}^t$ describes values of the controlled quantities that are possible when $\underset{\sim}{o}^t$ describes the values of the output quantities when the output device(s) are functioning correctly.

OUT describes the behaviour of the output devices. It is a relation rather than a function because of unavoidable device imperfections.

4.4 How can we document software requirements?

The software requirements are determined by the System Requirements Document and System Design Documents. As mentioned earlier, the *Software Requirements Document* can be seen as a combination of those two documents. It must describe the relations NAT, REQ, IN, and OUT.

In the sequel we assume that REQ is feasible with respect to NAT.

4.4.1 The Relation SOF

The software will provide a system with input-output behaviour that can be described by a relation, which we call SOF. It is defined as follows:

— domain(SOF) is a set of vectors of time-functions containing all possible instances of $\underset{\sim}{i}^t$,

— range(SOF) is a set of vectors of time-functions containing possible instances of $\underset{\sim}{o}^t$,

— $(\underset{\sim}{i}^t, \underset{\sim}{o}^t) \in$ SOF if and only if the software could produce values described by $\underset{\sim}{o}^t$ when the inputs are described by $\underset{\sim}{i}^t$.

SOF will be a function if the software is deterministic.

4.4.2 Software acceptability

For the software to be acceptable, SOF must satisfy[4]:

$$\text{—}\quad \forall \underset{\sim}{m}^t\ \forall \underset{\sim}{i}^t\ \forall \underset{\sim}{o}^t\ \forall \underset{\sim}{c}^t[\mathrm{IN}(\underset{\sim}{m}^t, \underset{\sim}{i}^t) \wedge \mathrm{SOF}(\underset{\sim}{i}^t, \underset{\sim}{o}^t) \wedge \mathrm{OUT}(\underset{\sim}{o}^t, \underset{\sim}{c}^t) \wedge \mathrm{NAT}(\underset{\sim}{m}^t, \underset{\sim}{c}^t) \Rightarrow \mathrm{REQ}(\underset{\sim}{m}^t, \underset{\sim}{c}^t)] \quad (4)$$

Note, that whenever one (or more) of the predicates $\mathrm{IN}(\underset{\sim}{m}^t, \underset{\sim}{i}^t)$, $\mathrm{OUT}(\underset{\sim}{o}^t, \underset{\sim}{c}^t)$, or $\mathrm{NAT}(\underset{\sim}{m}^t, \underset{\sim}{c}^t)$ is false, any software behaviour will be considered acceptable. For example, were a given value of $\underset{\sim}{m}^t$ to be outside the domain of IN, the behaviour of acceptable software would not be constrained by the above. Whenever the relations describing device behaviour do not hold, it means that a device is broken and the software cannot be required to satisfy system requirements under such conditions. However, it is possible to provide several different versions of the relations in order to describe “fail-soft” behaviour. The “fail soft” specifications would have weaker predicates, and the software would be expected to satisfy the conjunction of all requirements expressed in the form above. For example, one of these “fail soft” specifications might describe a broken device using weaker IN or OUT predicates and a weaker REQ.

If we treat relations REQ, IN, OUT, and SOF as functions, we can use functional notation to rewrite the implication above as follows:

$$\text{—}\quad \forall \underset{\sim}{m}^t[\underset{\sim}{m}^t \in \mathrm{domain(NAT)} \Rightarrow (\mathrm{REQ}(\underset{\sim}{m}^t) \ = \ \mathrm{OUT(SOF(IN}(\underset{\sim}{m}^t))))] \qquad (4a)$$

Using relational composition[5], and recalling that the domain of SOF will include all possible values of $\underset{\sim}{i}^t$, the above formula can be expressed in a more concise way:

$$\text{—}\quad (\mathrm{NAT} \cap (\mathrm{IN} \bullet \mathrm{SOF} \bullet \mathrm{OUT})) \subseteq \mathrm{REQ} \qquad (4b)$$

The authors of the requirements document must describe the relations NAT, REQ, IN, OUT. The software designers determine SOF and verify (4), (4a), or (4b).

A document of this type may require natural language and physical diagrams for the description of the environmental quantities, but can otherwise be precise and mathematical. The use of natural language in the definition of the physical interpretation of mathematical variables both unavoidable and usual in engineering.

4.5 How can we document software behaviour?

Even when the software requirements document fully represents the requirements that the software must meet, it may allow observable differences in behaviour. Frequently designers chose to implement a subset of the behaviours[6] that are allowed by the requirements document. In this way designers will make some decisions that might otherwise have been left for the programmers. The relation SOF, describing the behaviour of the actual implementation, can be described in a separate document known as the *Software Behaviour Specification.* This document is especially important for multiple-computer systems because it will define the allocation of tasks to the individual computers in the system. For computer networks, or multi-processor architectures one may see a hierarchy of software behaviour specifications with an upper level document assigning duties to a group of computers, and the lower level documents detailing the responsibilities of individual computers or groups of. computers.

[4] In the following the universes from which $\underset{\sim}{m}^t$, $\underset{\sim}{c}^t$, $\underset{\sim}{i}^t$ and $\underset{\sim}{o}^t$ are drawn can be assumed to include vectors of time-functions with the appropriate types in the ranges of those functions.

[5] Given two relations $R \subseteq A \times B$, and $S \subseteq B \times C$, *relational composition*, $R \bullet S$, can be defined by: $R \bullet S = \{(a,c) \in A \times C \mid \exists b \in B\ [R(a,b) \wedge S(b,c)]\}$.

[6] “Behaviours” are $(\underset{\sim}{m}^t, \underset{\sim}{c}^t)$ pairs.

4.6 How can we document black-box module interfaces?

Most computer systems require software that cannot be completed by a single person in a few weeks. For such products, it is desirable to decompose the software construction task into a set of smaller programming assignments. Each assignment is to produce a group of programs (cf. Section 4.8), which we call a *module*. In this section, we assume that the modules have been designed using the information hiding principle [23, 24]. The division of the software into modules is described informally in a *Software Module Guide*, which states the responsibilities of each module [4, 41]. However, one must specify the behaviour of these modules precisely to allow the module implementors to work independently with a reasonable likelihood that the separately written modules will function correctly when combined.

We view each module as implementing one or more finite state machines, frequently called *objects* or *variables*. A description of the module interface is a black-box description of these objects. Every program in the system belongs to exactly one module. These programs use the objects created by other modules as components of their data structure.

Writing software module interface specifications is, in principal, similar to documenting software requirements but some simplifications are possible. Many software modules are entirely internal; there are no environmental quantities to monitor or control and all communication can be performed through external invocation of the module's programs. Moreover, the state set of a software module is finite, and state transitions can be treated as discrete events. For most such modules, real-time can be neglected because only the sequence of events matters. This allows us to replace the concept of time-function by a sequence describing the history in terms of discrete events; we call these sequences *traces*.

We identify a finite subset of the (infinite) set of possible traces as *canonical traces*. Every trace is equivalent[7] to exactly one canonical trace. *Trace assertion specifications* comprise three groups of relations:

(1) Functions, whose domain is a set of pairs (canonical trace, event) and whose range is a set of canonical traces. The pair $((T_1,e), T_2)$ is in the function if and only if the canonical trace T_2 is equivalent to the (canonical) trace T_1 extended by the event e. These functions are known as *trace extension functions*[8].

(2) Relations, whose domain contains all the canonical traces and associate each canonical trace with a set of values of output variables.

(3) Functions, whose domain is the set of values of the output variables and whose values define the information returned by the module to the user of the module.

Sometimes a single module will be used to implement many objects of a given class or type. For example, a single module can be used to implement many stacks. In this case, we may choose to design our module so that the objects are completely independent and then describe the behaviour of a typical object of the class.

Reports [17, 18, 36, 49] describe this method in more detail and show how the set of functions can provide a precise but readable description of the externally visible behaviour of a module. Older discussions of this approach can be found in [2, 14, 15].

[7] Two traces are *equivalent* if they have the same effect on future behaviour of the object. More precise definitions can be found in [2, 17, 18, 36, 37, 49].

[8] A trace extension function is sometimes called a *reduction* function. An alternate model, using an extension relation, is also possible.

4.7 How can we document internal module design?

Each module has a private data structure and one or more programs. We propose to document the design sufficiently precisely that its correctness can be verified without reference to the code. The internal documentation of a module should contain three types of information:

(1) A complete description of the data structure, which may include objects (variables) implemented by other modules.

(2) A function, known as the *abstraction function* ([13, 21][9]), whose domain is a set of pairs (object name, data state), and whose range is the set of canonical traces for objects created by the module. The pair ((o, d), T) is included in this function if and only if one of the traces equivalent to T describes a sequence of events affecting the object named o that could have resulted in the data state d.

(3) An LD-relation, often referred to as the *program function* ([8, 21]), specifying the behaviour of each of the module's programs in terms of mappings from data states before the program execution to data states after the execution (cf Section 4.8).

As has been shown by Hoare [13] and others, (e.g. [8, 21]), this information allows the design to be verified and may subsequently be used to check on the implementation of the module interface. If we view the module as creating a single object, (which simplifies the discussion by eliminating object names) and consider only deterministic programs, design verification is illustrated by the diagram below. The lower level represents changes in the module's data structure caused by a program invocation (or other event). The upper level represents the external view of those changes. A canonical trace, T_1, extended by a single event, e, is mapped by the trace extension function (cf Section 4.6, item (1)) to a canonical trace, T_2. The program function for the event e maps the old data state, ds_1, to a new data state, ds_2. The abstraction function maps data states to canonical traces. If the design is correct, the diagram "commutes" for all possible events.

$$\begin{tikzcd}
T_1 \arrow[r, "\text{trace extension function for event } e"] & T_2 \\
ds_1 \arrow[u, "\text{abstraction function}"] \arrow[r, "\text{program function for event } e"'] & ds_2 \arrow[u, "\text{abstraction function}"']
\end{tikzcd}$$

If we view the module as producing a set of (named) objects, the diagram becomes slightly more complex (because the domains of the functions must be modified to include the names of the objects), but the principle remains the same. If the program is non-deterministic, the program function would be an LD-relation, (cf Section 4.8); all of the possible next data states must be mapped to the same canonical trace by the abstraction function.

4.8 How can we document the effect of individual programs?

We use the term *program* to denote a text describing a set of state sequences in a digital (finite state) machine. Each of those state sequences will be called an *execution* of the program.

[9] Mills et al [21] use the term "representation mapping" where we use "abstraction function" (p. 502).

When an execution begins, the machine is in a particular state, called the *starting state*. A sequence of state changes, which may or may not terminate, will then commence. If the sequence does terminate, its last state is called the *final state* and we say that the program's execution *terminates* or, more simply, that the program terminates.

In many situations, we do not want to document the intermediate states in the sequence. For each starting state, *s*, we want to know only:

(1) is termination possible, i.e. are there finite executions beginning in *s*?

(2) is termination guaranteed, i.e. are all executions that begin in *s* finite?

(3) if termination is possible, what are the possible final states?

All of this information can be described by an LD-relation ([27, 43, 44]). An LD-relation comprises a relation and a subset of the domain of that relation, called the *competence set*. In a description of a program, the set of starting states for which termination is guaranteed is the competence set. The set of starting states for which termination is possible is the domain of the relation. An ordered pair (*x,y*) is in the relation if it is possible that the program's execution would terminate in state *y* after being started in state *x*.

We can document the effects of executing a program by describing an LD-relation[10]. If the competence set is identical to the domain of the relation, we, by convention, omit it. If the program is deterministic this yields exactly the function used by Mills [20]. The fact that the LD-relation can be reduced to Mills' function (when the program is deterministic) means that these concepts provide a set of upward compatible notations; one may omit information when it is redundant, and mix the notations, because all are special cases of the most general concept.

LD-relations can be used either as specifications for programs (stating the behaviour required of those programs) or as complete descriptions of the actual behaviour of a program. Longer programs should be presented as a self-documenting set of understandable short programs with each program presented in a *display*. A display comprises (a) the specification of the program, (b) the program itself (possibly containing names of other programs), and (c) specifications of those named programs. It should be possible to understand and verify each display using only the information in the display, supplemented by the dictionary definitions, without any reference to other displays. This aspect of documentation is discussed in more detail in [43, 44].

4.9 How can we document data flow?

There are two interpretations of "data flow" in computer systems. One is a description of the way that information "flows" from one variable to another. After each execution of a program, or after each execution of the outer loop of a program that controls a periodic process, there will be constraints relating the values of the variables in the programs. These constraints can be described by relations. There will be one relation for each variable in the program. The range will be the set of possible values for that variable. The domain of each relation will be the set of possible values for the other variables in the program. Such documentation often helps when debugging a program. It describes constraints that can be checked at run-time.

[10] The texts called "procedures with parameters" are not programs (according to the definition given in this section). Their behaviour can be described by a more general concept, the so called "parameterised program functions" (cf [17]).

A second interpretation of data flow is particularly convenient for concurrent real-time programs. We often view software as consisting of a set of processes, each of which is considered to be a device that transforms a sequence of input values to a sequence of output values. The data flow between these transducers can be described using the same approach that we propose for the description of system requirements. If real-time is not a concern, the trace assertion method can then be applied to document the behaviour of each process.

We have no experience with either approach but suggest that they are worthy of further study.

4.10 How can we document communication services and protocols?

Modern communications systems are often implemented as a hierarchy of services, each service using the one below it in the hierarchy. Each level in a hierarchical communications system can be viewed as a module; the service offered can be specified by means of the trace assertion method. This document corresponds to what is usually called a *service specification*. An implementation of a given service by the use of lower level services and local data structures can be (partially) described using the program functions and abstraction functions mentioned in Section 4.7. These functions correspond to what is often called a *protocol design*. This is discussed in more detail in [3, 5, 14, 15, 28]. Most communication protocols have a relatively small number of states. Consequently, enumerative analysis techniques, which would not be practical in many other applications, can be used in verification of protocols. In the past decade, work on the use of “formal methods” for protocol description and analysis has diverged from work on verification of other types of programs. We believe that, in spite of the fact that different analysis techniques are available, the notation used in protocol work can be the same as that used elsewhere in computer systems. We would hope to see a merging of the work from these two communities by using the concepts applied in this paper.

4.11 How can we document chip design?

Current technology allows us to fabricate chips that perform tasks so complex that they would have been implemented in software a few years ago. The externally visible behaviour of these chips can be described using the techniques from Sections 4.2 and 4.6. A specification written in this way could be called a *Chip Behaviour Specification*. If we make the usual discrete state assumptions, their behaviour can be approximated using the trace assertion method. It may also prove useful to document the internal structure of the chips by means of abstraction functions and functions that describe the change in the internal state that result from external events.

Except for memories (which can be described simply because of their regularity), digital hardware components usually have much smaller state spaces than software systems. Often the state spaces are small enough that enumerative design, and design-validation techniques can be used. We believe that the concepts discussed in this paper are applicable to hardware as well as to software, and that the small state space makes it possible to perform some analyses that would not be practical for software systems.

5 How can we represent functions in computer systems engineering?

While the above definitions are precise, they are not sufficient to allow practical work, because we have not agreed upon a notation that can be used to provide definitions of the functions and relations involved. Turning the abstract ideas into a method that can be applied in a project, requires that we define a syntax for expressions and explain how those expressions are to be interpreted as functions or relations.

A trap, into which some “formal methods” groups have fallen is to attempt to define a universal notation for the definition of functions. It is tempting to define languages such as VDM or Z, in the hope that doing so would give us a universal specification language. Unfortunately, as centuries of research in applied mathematics has shown, there is no one representation that is best for all classes of functions. For example, there are functions that are easily expressed as polynomials, but many functions that can only be approximated, often in a clumsy way, by that class of expressions. If history is a good predictor of the future, we will continue to invent improved forms of expressions as we discover new classes of functions to be of interest.

Our experience with these methods has revealed that the relations of interest in computer system design are often well represented by tables whose entries are expressions, possibly other such tables. In practice, the functions that we want to describe have a great many points of discontinuity and the conditional expressions that describe those functions are lengthy and difficult to parse. A tabular form of expression parses the information for the reader and factors out many of the common subexpressions. We define the semantics of tables by giving a translation of a table to a boolean expression, which can be interpreted as the characteristic predicate of the relation viewed as a set of ordered pairs. A companion paper gives the semantics of a variety of table formats [30]. A more recent paper [19] provides a systematic definition of a much larger class of table formats. The interpretation of predicate expressions that we use in these tables is discussed in [29].

6 Final remarks

This section presents some thoughts on the soundness, practicality, and value of these ideas.

6.1 Do we have evidence that the theory is sound?

Many of the ideas presented in this paper are old by computer science standards. They are like the wheel, which is often reinvented because it is a good idea. The system requirements model is essentially that used in control theory; we have made some minor adjustments to accommodate current computer terminology. An early version of the software requirements model was first used in 1977 (to produce[10]); the approach seems quite obvious to those who have some background in control theory. The trace assertion model has been in use since 1978 [2] and is very close to certain algebraic theories. Its main advantage is that it evades several difficult issues by using functions on a domain consisting of strings rather than function compositions. Relational semantics has been thoroughly explored by many authors. An excellent text on relational methods is [46]. More discussion of our particular version can be found in [32, 44]. The internal module documentation model is more than 20 years old and has been reinvented by several researchers. Our effort has been to show how these well-understood theories can be applied to the problem of documentation. Because these approaches have been thoroughly studied by many others, we are confident that there will be few unpleasant surprises in our use of this model for documentation.

6.2 Do we have evidence that the theory scales up?

Many theories prove impractical when applied to realistic applications. However, the original aspects of the ideas presented in this paper grew out of practical applications and experience has shown them to be usable. The systems and software requirements models evolved out of work at the United States Naval Research Laboratory on the U.S. Navy's A-7 aircraft [10, 11, 41] The ideas have since been used by a variety of organisations including Bell Laboratories [12], AECL (Atomic Energy Canada Limited) [38, 40], and the U.S. Air Force. An early version of the trace model (2) has been used successfully by PROSYS GmbH in Darmstadt, Germany. The internal documentation approach was used for a disciplined verification of safety-critical software by Ontario Hydro [1, 38, 40].

All of these experiences, although they were successful, demonstrated that functional methods, using tabular representations, are useful but very time consuming. The methods demand precision and care similar to that required in programming or mathematics. On the basis of observations about how the Engineer's time was spent on the projects mentioned, we believe that the production of these documents will not be fully practical until we have new tools available. One class of tool would reduce the time and effort required to write these documents by automating some of the dreariest work. More advanced tools would increase the value of these documents by (a) checking the validity of the tables, (b) simulating designs based on specifications, (c) generating test cases from these specifications, and (d) inserting run-time checks and diagnostic programs based on the specifications. Some such work is now in progress (e.g. [16, 45]) but there is a great deal more that could be done.

6.3 What can we gain by using these concepts?

We believe that there is much to be gained by using these concepts for documentation. The primary advantage would be an increase in the quality of the documentation; mathematical documentation is far more likely to be complete and correct than informally written documentation. This, rigidly organised, documentation is advantageous as a reference work when one must find specific information. It is not particularly suitable as an introduction to the software.

We see very great advantages in having a common set of notations to be used through the computer system design process. Having a common set of concepts will allow a set of basic tools that can be used throughout the whole process, from systems design to software design, and even into chip design. Our approach, in which the interpretation of tables can be defined by translating the table into a conventional expression, is extensible. New table formats can be introduced as needed because our basic model is completely independent of the representation of the functions.

Acknowledgements

We are grateful for the comments by many people, including N.S. Erskine, D.M. Hoffman, M. Iglewski, P. Kelly, A. Malton, R. Milner, J.C. Muzio, A.P. Ravn, K.A. Schneider, M. Serra, M. Sintzoff, R. Taylor, W.M. Turski, A.J. van Schouwen, D. Weiss, Y. Wang and some of the anonymous referees.

H.D. Mills and N.G. de Bruijn, inspired the senior author to think in functional terms.

This work was supported by the Province of Ontario through Telecommunications Research Institute of Ontario (TRIO), by the Atomic Energy Control Board (AECB), by the Natural Sciences and Engineering Research Council of Canada (NSERC), by the State Committee for Scientific Research in Poland (KBN), and by Digital Equipment's External Research Programme.

References

1 G.H. Archinoff, R.J. Hohendorf, A. Wassyng, B. Quigley and M.R. Borsch, "Verification of the Shutdown System Software at the Darlington Nuclear Generating Station", in: *Proceedings of the International Conference on Control & Instrumentation in Nuclear Installations*, The Institution of Nuclear Engineers, Glasgow, United Kingdom, May 1990, No. 4.3, 23 pp.

2 W. Bartussek and D.L. Parnas, "Using Assertions About Traces To Write Abstract Specifications for Software Modules", in: *Proceedings of 2nd Conference of European Cooperation in Informatics*, Venice, 1978, Lecture Notes in Computer Science, 65, Springer-Verlag, pp. 211-236, 1978. Reprinted in:

N. Gehani, A.D. Mc Gettrick (Eds.), *Software Specification Techniques,* AT&T Bell Telephone Laboratories, 1985, pp. 111-130.

3 J. Bojanowski, M. Iglewski, J. Madey and A. Obaid: “Functional Approach to Protocols Specification”, in: *Proceedings of the 14th International IFIP Symposium on Protocol Specification, Testing & Verification (PSTV’94)*, Vancouver, B.C, Canada, June 7-10, 1994, pp. 371-378.

4 K.H. Britton and D.L. Parnas, “A-7E Software Module Guide”, *NRL Memorandum Report* 4702, United States Naval Research Laboratory, Washington D.C., December 1981, 35 pp.

5 F. Courtois, and D.L. Parnas, “Formally Specifying A Communications Protocol Using the Trace Assertion Method”, *CRL Report* 269, McMaster University, CRL, Telecommunications Research Institute of Ontario (TRIO), Hamilton, Ontario, Canada, July1993, 19 pgs.

6 E.W. Dijkstra, *Discipline of Programming*, Prentice-Hall, 1976.

7 M. Engel, M. Kubica, J. Madey, D.L. Parnas, A.P. Ravn and A.J. van Schouwen, “A Formal Approach to Computer Systems Requirements Documentation”, in: *Hybrid Systems*, R.L. Grossman, A. Nerode A.P. Ravn, H. Rischel (Eds.), Lecture Notes in Computer Science 736, Springer-Verlag, 1993, pp. 452-474.

8 J.D. Gannon, R.G. Hamlet and H.D. Mills, “Theory of Modules”, *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 7, July 1987, pp. 820-829.

9 E.C.R. Hehner, “Predicative Programming”, *Communications of the ACM*, Vol. 27, No. 2, February 1984, pp. 134-151.

10 K.L. Heninger, J. Kallander, D.L. Parnas and J.E. Shore, “Software Requirements for the A-7E Aircraft”, *NRL Memorandum Report* 3876, United States Naval Research Laboratory, Washington D.C., November 1978, 523 pp.

11 K.L. Heninger, “Specifying Software Requirements for Complex Systems: New Techniques and their Application”, *IEEE Transactions Software Engineering*, Vol. SE-6, No. 1, January 1980, pp. 2-13.

12 S.D. Hester, D.L. Parnas and D.F. Utter, “Using Documentation as a Software Design Medium”, *Bell System Technical Journal*, Vol. 60, No. 8, October 1981, pp. 1941-1977.

13 C.A.R. Hoare, “Proof of Correctness of Data Representations”, *Acta Informatica*, Vol. 1, No. 19, 1972, February, pp. 271-281.

14 D.M. Hoffman, “Trace Specification of Communications Protocols”, *Ph.D. Thesis*, University of North Carolina, Chapel Hill, 1984.

15 D.M. Hoffman, “The Trace Specification of Communications Protocols”, *IEEE Transactions on Computers*, Vol. C-34, No. 12, December 1985, pp. 1102-1113.

16 M. Iglewski, M. Kubica and J. Madey, “Editor for the Trace Assertion Method”, in: *Proceedings of the 10th International Conference of CAD/CAM, Robotics and Factories of the Future: CARs & FOF’94*, M. Zaremba (Ed.), OCRI, Ottawa, Ontario, Canada, 1994, pp. 876-881.

17 M. Iglewski, J. Madey and D.L. Parnas, “Documentation Paradigms”, *CRL Report* 270, McMaster University, CRL, Telecommunications Research Institute of Ontario (TRIO), Hamilton, Ontario, Canada, July 1993, 45 pp.

18 M. Iglewski, J. Madey and K. Stencel, “On Fundamentals of the Trace Assertion Method”, *Technical Report* TR 94-09 (198), Warsaw University, Institute of Informatics, Warsaw 1994, 8 pgs. (Also published by the University of Quebec in Hull, Department of Computer Science, Canada, as a Technical

Report RR 94/09-6.)

19 R. Janicki, “Towards a Formal Semantics of Tables”, *CRL Report* 264, McMaster University, CRL, Telecommunications Research Institute of Ontario, (TRIO), Hamilton, Ontario, Canada, September 1993, 18 pgs. (A revised version of this will appear in the proceedings of the *17th International Conference on Software Engineering*.)

20 H.D. Mills, “The New Math of Computer Programming”, *Communications of the ACM*, Vol. 18, No. 1, January 1975, pp. 43-48.

21 H.D. Mills, V.R. Basili, J.D. Gannon and R.G. Hamlet, *Principles of Computer Programming: A Mathematical Approach*, Allyn and Bacon, 1987.

22 H.D. Mills, R.C. Linger and A.R. Hevner, “Box Structured Information Systems”, *IBM Systems Journal*, Vol. 26, No. 4, 1987, pp. 395-413.

23 D.L. Parnas, “Information Distributions Aspects of Design Methodology”, in: *Proceedings of the IFIP Congress’71*, Booklet TA-3, 1971, pp. 26-30.

24 D.L. Parnas, “On the Criteria to be Used in Decomposing Systems into Modules”, *Communications of the ACM*, Vol. 15, No. 12, December 1972, pp. 1053-1058.

25 D.L. Parnas, “On a “Buzzword”: Hierarchical Structure”, in: *Proceedings of the IFIP Congress ‘74*, North Holland, 1974, pp. 336-339.

26 D.L. Parnas, “Designing Software for Ease of Extension and Contraction”, *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 2, March 1979, pp. 128-138.

27 D.L. Parnas, “A Generalized Control Structure and Its Formal Definition”, *Communications of the ACM*, Vol. 26, No. 8, August 1983, pp. 572-581.

28 D.L. Parnas, “Documentation of Communications Services and Protocols”, *Technical Report* 90-272, Queen’s University, C&IS, Telecommunications Research Institute of Ontario (TRIO), Kingston, Ontario, Canada, February 1990, 4 pp.

29 D.L. Parnas, “Predicate Logic for Software Engineering” *IEEE Transactions on Software Engineering* Vol. 19, No. 9, September 1993, pp. 856-862.

30 D.L. Parnas, “Tabular Representation of Relations”, *CRL Report* 260, McMaster University, CRL, Telecommunications Research Institute of Ontario (TRIO), Hamilton, Ontario, Canada, October 1992, 17 pp.

31 D.L. Parnas, “Software Aging”, in: *Proceedings of the 16th International Conference on Software Engineering*, Sorento Italy, May 16-21 1994, IEEE Press, pp. 279-287.

32 D.L. Parnas, “Mathematical Descriptions and Specification of Software”, in: *Proceedings of IFIP World Congress 1994, Volume I”* August 1994, pp. 354-359.

33 D.L. Parnas and J. Madey, “Functional Documentation for Computer Systems Engineering”, *Technical Report* 90-287, Queen’s University, C&IS, Telecommunications Research Institute of Ontario (TRIO), Kingston, Ontario, Canada, September 1990, 14 pp.

34 D.L. Parnas and J. Madey, “Functional Documentation for Computer Systems Engineering. (Version 2)”, *CRL Report* 237 McMaster University, CRL, Telecommunications Research Institute of Ontario (TRIO), Hamilton, Ontario, Canada, September 1991, 14 pp.

35 D.L. Parnas and J. Madey, “Documentation of Real-Time Requirements”, in K.M. Kavi (ed.) *Real-Time Systems. Abstraction, Languages and Design Methodologies*, IEEE Computer Society Press,

1992, pp. 48-56.

36 D.L. Parnas and Y. Wang, “The Trace Assertion Method of Module Interface Specification”, *Technical Report* 89-261, Queen’s, C&IS, Telecommunications Research Institute of Ontario (TRIO), Kingston, Ontario, Canada, October 1989, 39 pp.

37 D.L. Parnas and Y. Wang, “Simulating the Behaviour of Software Modules by Trace Rewriting Systems”, *IEEE Transactions of Software Engineering,* Vol. 20, No. 10, October 1994, pp. 750-759.

38 D.L. Parnas, G.J.K. Asmis and J.D. Kendall, “Reviewable Development of Safety Critical Software”, in: *Proceedings of the International Conference on Control & Instrumentation in Nuclear Installations*, The Institution of Nuclear Engineers, Glasgow, United Kingdom, May 1990, No. 4.2, 17 pp.

39 D.L. Parnas, G.J.K. Asmis and J. Madey, “Assessment of Safety-Critical Software”, *Technical Report* 90-295, Queen’s University, C&IS, Telecommunications Research Institute of Ontario (TRIO), Kingston, Ontario, Canada, December 1990, 13 pp.

40 D.L. Parnas, G.J.K. Asmis and J. Madey, “Assessment of Safety-Critical Software in Nuclear Power Plants”, *Nuclear Safety*, Vol. 32, No. 2, 1991, pp. 189-198.

41 D.L. Parnas, P.C. Clements and D.M. Weiss, “The Modular Structure of Complex Systems”, *IEEE Transactions on Software Engineering*, Vol. SE-11, March 1985, pp. 259-266.

42 D.L. Parnas and P.C. Clements, “A Rational Design Process: How any Why to Fake It”, *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 2, February 1986, pp. 251-257.

43 D.L. Parnas, J. Madey and M. Iglewski., “Formal Documentation of Well-Structured Programs”, *CRL Report* 259, McMaster University, CRL, Telecommunication Research Institute of Ontario (TRIO), Hamilton, Ontario, Canada, September 1992, 37 pp.

44 D.L. Parnas, J. Madey and M. Iglewski., “Precise Documentation of Well-Structured Programs”, *IEEE Transactions on Software Engineering*, Vol. 20, No.12, December 1994, pp. 948 - 976.

45 D. Peters and D.L. Parnas, “Generating a Test Oracle from Program Documentation”, in: *Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA),* August 17-19, 1994, pp. 58 - 65.

46 G. Schmidt and T. Strohlein, *Relations and Graphs - Discrete Mathematics for Computer Scientists*, Springer-Verlag, 301 pgs, 1993.

47 A.J. van Schouwen, “The A-7 Requirements Model: Re-examination for Real-Time Systems and an Application to Monitoring Systems”, *Technical Report* 90-276, Queen’s, C&IS, Telecommunications Research Institute of Ontario (TRIO), Kingston, Ontario, Canada, May 1990, 93 pp.

48 A.J. van Schouwen, D.L. Parnas and J. Madey, “Documentation of Requirements for Computer Systems”, in: *Proceedings of the IEEE International Symposium on Requirements Engineering*, San Diego, California, USA, January 4-6, 1993, pp. 198-207.

49 Y. Wang, “Specifying and Simulating the Externally Observable Behavior of Modules”, (Ph.D. Thesis), *CRL Report* 292, McMaster University, CRL, Telecommunication Research Institute of Ontario (TRIO), Hamilton, Ontario, Canada, 1994.