Data & Knowledge Engineering 5 (1990) 119–128
North-Holland

Toward higher-level abstractions for software systems*

Mary SHAW
Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213, U.S.A.

Address to the Third International Symposium on Knowledge Engineering, Polytechnical University of Madrid, October 17–21, 1988, Madrid, Spain.

Abstract. Software now accounts for most of the cost of computer-based systems. Over the past thirty years, abstraction techniques such as high level programming languages and abstract data types have improved our ability to develop software. However, the increasing size and complexity of software systems have introduced new problems that are not solved by the current techniques. These new problems involve the system-level design of software, in which the important decisions are concerned with the kinds of modules and subsystems to use and the way these modules and subsystems are organized. This level of organization, the *software architecture* level, requires new kinds of abstractions. These new abstractions will capture essential properties of major subsystems and the ways they interact.

Keywords: Software architecture, Abstraction techniques.

1. Introduction

1.1 The nature of the software problem

Twenty-five years ago, software accounted for only a small fraction of the costs of a computer-based system. Now software often accounts for the majority of the costs over the lifetime of such a system. As a result of this shift, the development and maintenance of software have become the major critical factor in system development. This is particularly true when the systems are large and complex, when they involve real-time control of machinery or processes, and when they are significantly more ambitious than earlier systems.

The problems that are cited most frequently are the high cost of developing software and low productivity of software developers. However, the problem of software production has several other important aspects. These include:

- *Complexity*. The demand for both size and capability of computer-based systems increases steadily.
- *Change*. Individual systems evolve over time, and software from one system may be reused in another; consistency among related products affects the ease and cost of change.
- *Reliability*. Software is increasingly used for critical processes to which reliability of the software is critical, for example when fast execution speeds make direct human surveillance impractical.
- *Management control*. Costs and schedules are unpredictable; this is often more significant than the actual cost, even if the actual cost is high.

* An extract from this paper titled "Larger Scale Systems Require Higher-Level Abstractions" was published in the *Proceedings of the Fifth International Workshop on Software Specification and Design*, Pittsburgh, Pennsylvania, May 1989 (Copyright 1989, Association for Computing Machinery, Inc.).

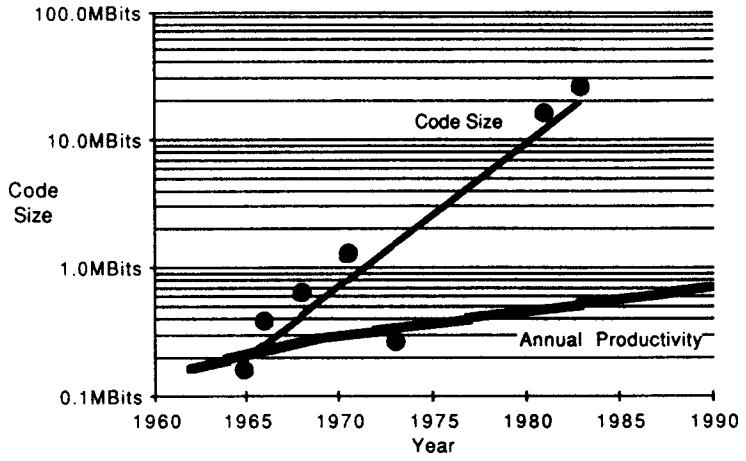

Fig. 1. Growth of demand.

Our expectations for software size, complexity, and performance have grown exponentially over this quarter-century, but our production ability has not kept pace. *Fig. 1* illustrates the growth of demand by comparing the growth of programmer productivity with the growth in code size for one class of systems, on-board software for manned space flights [3, 16]. The exponential increase of code size (about 20–25% annual growth in this example) is very substantially larger than the improvement in programmer productivity (about 4–6% annual growth). The most obvious conclusion from this data is that we must find ways to achieve an order of magnitude improvement in software productivity. A less obvious, but equally significant, conclusion is that we must find ways of developing software systems whose size far exceeds any that we have experienced.

1.2 Current research areas

We often speak of software development as a routine production or assembly activity. To the contrary, it is primarily a design activity. Software development now involves establishing a requirement, separating its functionality into modules, producing program text for these modules, integrating the modules, and validating the result. Also, maintenance for software is unlike maintenance for machinery; for software it includes both repairing design flaws and changing the design, but not the alleviation of normal wear. In both development and maintenance, the major focus is on the program, expressed as lines of code written in some programming language and often augmented by summary reports such as dictionaries.

Two classes of software development tasks are distinguished, *programming-in-the-small* and *programming-in-the-large*. Programming-in-the-small is concerned with algorithms and data structures, with programming languages for implementing them, and with functional specifications for describing their properties. The tasks of programming-in-the-large were identified as a distinct class when software grew too large for individual programmers to manage. This class is concerned with notations for connecting modules, with configuration management and version control, and with tools for programmers such as integrated environments and management tools.

These areas continue to have good research problems. Current research areas include programming languages (taking full advantage of Ada and improvements in non-imperative languages), tools and environments (integrated development environments, computer-aided software engineering (CASE), and project management tools) and "methodologies", or

procedures for software development (various design methods, software reusability, and techniques for management control).

This work usually assumes that software – and the process of developing software – will remain much like it is now: the task is decomposed into work units; programmers write sequences of statements in programming languages; these statements are grouped into modules; the programs are modified from time to time, usually by hand, as requirements change and errors are discovered. However, there is substantial reason to doubt that the required levels of performance and reliability can ever be achieved if the primary emphasis of software developers is on the program text as such. Rather, it appears that the gross organization of a software system is a major determinant of the performance and suitability of the final product.

Growth in the scale of software systems led to the need for the techniques of programming-in-the-large. Similarly continued growth in system scale and in demands for performance, reliability, and economy has made a new set of problems become significant. These problems are concerned with the overall organization of software systems: what kinds of system structure are useful, when is each most useful, what sorts of common subsystems are used to build these systems, what properties are important to specify, and how can design knowledge be codified and made available?

1.3 Plan of the paper

This paper identifies a new reseach area and describes the central research questions in that area. The premise of this discussion is that design and development should be carried out at a level of abstraction where the focus of attention is on patterns of system organization, including overall organization strategies and common subsystems. At this level, details of implementation do not interfere with organizing overall system function, system integrity is not vulnerable to clerical errors, and engineering tradeoffs can be thoughtfully made. We will call this the *software architecture* level of software design and reuse.

We begin by reviewing the leverage that abstraction techniques have provided in software development over the years. We then introduce the architectural design level and give examples of software system architectures to show that design already takes place, at least informally, at this level. Finally, we identify some important research questions.

2. Abstraction techniques

2.1 Historical significance

The development of abstraction techniques has been a major source of improvement in programming practice. It has provided programming language constructs, specification techniques, program structures such as algorithms and data types, strategies for modular decomposition, and more [15].

We periodically recognize that we are regularly reconstructing the same pattern – for example, a code fragment, a specification, or a relation among data elements – in different contexts. Each time we automate that pattern in some way, we are performing abstraction. The automation may be macro generation, introduction of a control construct in a progamming language, specification in terms of an existing model, type checking of parameters, naming of a representation as a data type, etc.

The essence of abstraction is recognizing a pattern, naming and defining it, and providing some way to invoke the pattern by its name without error-prone manual intervention. This

process suppresses the details of implementing the pattern, reduces the opportunity for clerical error, and eases understanding of the result.

There are, of course, many kinds of detail associated with software. It is possible to identify abstractions of many kinds, organizing different sorts of detail – about functionality, implementation, performance, physical properties, and so on. At any given time, some of these details are significant and others irrelevant. Consequently, different abstractions are appropriate at different points in the software design and development process. In other words, good abstraction is ignoring the right detail at the right times.

The development of individual abstractions often follows a common pattern. First, problems are solved *ad hoc*. Experience shows that some solutions are better than others, and a sort of folklore is passed informally from person to person. Eventually the useful solutions are understood more systematically, and they are codified and analyzed. This enables the development of models that support automatic implementation and theories that allow the extension or generalization of the solution. This in turn enables a more sophisticated level of practice and allows us to tackle harder problems – which we often approach *ad hoc*, starting the cycle over again. As time passes, the abstractions capture larger and larger amounts of detail with more and more precision. This allows increasingly complex programs to be described with a given amount of effort.

Two examples illustrate the role abstraction techniques have had in software development technology: higher-level programming languages and abstract data types.

2.2 Higher-level programming languages

When digital computers emerged in the 1950s, software was written in machine language; programmers placed instructions and data individually and explicity in the computer's memory. Insertion of a new instruction in a program might require hand-checking of the entire program to update references to data and instructions that moved as a result of the insertion. Eventually it was recognized that the memory layout and update of references could be automated, and also that symbolic names could be used for operation codes and memory addresses. Symbolic assemblers were the result. They were soon followed by macro processors, which allowed a single symbol to stand for a commonly-used sequence of instructions. The substitution of simple symbols for machine operation codes, machine addresses yet to be defined, and sequences of instructions was perhaps the earliest form of abstraction in software.

In the latter part of the 1950s, it became clear that certain patterns of execution were commonly useful – indeed, they were so well understood that it was possible to create them automatically from a notation more like mathematics than machine language. The first of these patterns were for evaluation of arithmetic expressions, for procedure invocation, and for loops and conditional statements. These insights were captured in a series of early high-level languages, of which Fortran was the main survivor.

Higher-level languages allowed more sophisticated programs to be developed, and patterns in the use of data emerged. Whereas in Fortran data types served primarily as cues for selecting the proper machine instructions, data types in Algol and its successors serve to state the programmer's intentions about how data should be used. The compilers for these languages could build on experience with Fortran and tackle more sophisticated compilation problems. Among other things, they checked adherence to these intentions, thereby providing incentives for the programmers to use the type mechanisms.

Progress in language design continued with the introduction of modules to provide protection for related procedures and data structures, with the separation of a module's specification from its implementation, and with the introduction of abstract data types.

2.3 Abstract data types

In the late 1960s, good programmers shared an intuition about software development: *If you get the data structures right, the effort will make development of the rest of the program much easier.* The abstract data type work of the 1970s can be viewed as a development effort that converted this intuition into a real theory. The conversion from an intuition to a theory involved understanding

- the software structure (which included a representation packaged with its primitive operators),
- specifications (mathematically expressed via abstract models or algebraic axioms),
- language issues (modules, scope, user-defined types),
- integrity of the result (invariants of data structures and protection from other manipulation),
- rules for combining types (declarations),
- information hiding (protection of properties not explicitly included in specifications).

By the late 1970s, good theory existed for abstract data types. This material is now taught routinely to undergraduates.

The effect of this work was to raise the design level of certain elements of software systems, namely abstract data types, above the level of programming language statements or individual algorithms. This provided understanding of a good organization for an entire module that serves one particular purpose. This involved combining representations, algorithms, specifications, and functional interfaces in uniform ways. Certain support was required from the programming language, of course, but the abstract data type paradigm allowed some parts of systems to be developed from a vocabulary of data types rather than from a vocabulary of programming-language constructs.

Abstract data types provide one way, but not the only way, of organizing software. However, comparable theories have not been developed for other software organizations, though theories for some aspects of some structures have emerged. Nor do we understand systematic ways to combine several software organizations in a complete system – or even of how to tell whether it's reasonable to try.

3. The software architecture level of design

Just as good programmers recognized useful data structures in the late 1960s, good programmers now recognize useful system organizations. One of these is based on the theory of abstract data types. But this is not the only way to organize a software system. Some of the popular system organizations are

- *Data abstraction.* Abstract data types provide a useful way to organize a software system. With the addition of inheritance of definitions, it is known as *object-oriented programming* [11]. *Fig. 2* suggests a typical system organization, in which a number of independent objects (*obj*) invoke each others' operations (*op*).
- *Pipes and filters.* Each module accepts a stream of inputs and emits a stream of outputs; usually the output stream involves local transformation of the input stream, making the module a *filter*. As indicated in *Fig. 3*, modules are strung together by connecting the output stream of one module to serve as the input stream of its successor (a *pipe*).
- *Layered systems.* The system is organized hierarchically, with each layer providing service to the layer above it. Inner layers are usually hidden from outer layers, except for certain functions carefully selected for export [13]. *Fig. 4* gives an example of such an architecture.

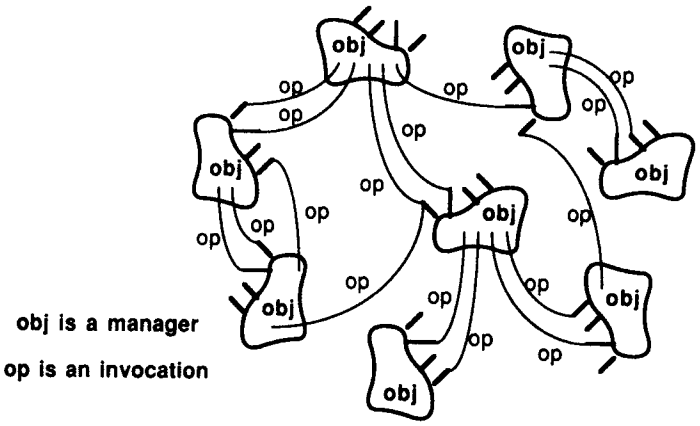

Fig. 2. Data abstraction.

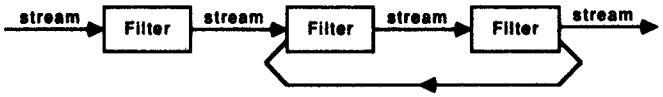

Fig 3. Pipes and filters.

- *Rule-based systems*. The computation is organized as an unordered collection of rules, each of which gives a condition under which the rule applies and an action to take when it does apply [9]. *Fig. 5* shows a typical system organization in which an inference engine interprets the rules that define the knowledge base for a particular application.
- *Blackboard systems*. A central data structure represents the current state of the computation; a collection of independent processes, each responsible for a certain kind of knowledge, check the current state and update it if they can. As indicated in *Fig. 6*, processes, or knowledge sources (*ksi*), are invoked when they are able to improve the current state; the progress made by each knowledge source should enable some other process to operate. This organization is useful for applications in which diverse kinds of knowledge must be brought to bear on interpretation of raw data; the current state represents the current state of the interpretation and the various processes augment that interpretation using whatever expertise each one implements [6].

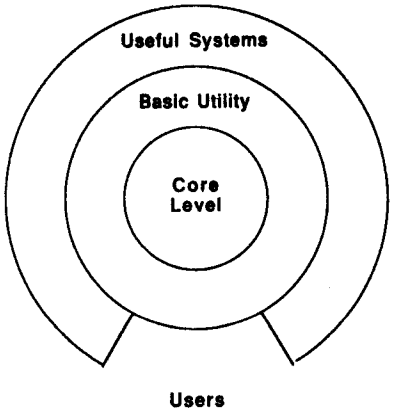

Fig. 4. Hierarchial system.

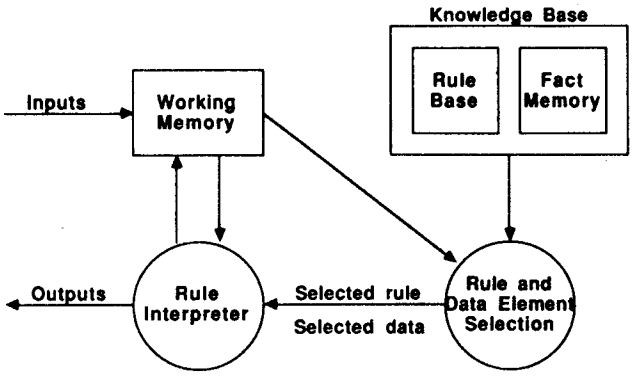

Fig. 5. Rule-based system.

This level of system organization and design is the *software architecture* level. Software organizations like these are used informally, but they are not systematically understood. Nor are they taught systematically, with guidance about reasons for choosing one rather than another for a given application. However, a software designer now often describes the architecture of a particular system. Most commonly, the description includes a block diagram of the major subsystems and labels the elements in terms that refer to their specific function in the system.

In particular problem domains or user communities, a single pattern many predominate. This pattern may be used because it is familiar, or it may be used because other alternatives are not considered, even when some other organization might be better. Consider, for example, the unix community, in which pipes and filters are clearly the preferred system structure. In any case, the resulting uniformity of program structure may permit the development of utilities to support the structure [7] or partial automation of program construction.

In order to design effectively at this level, we need to be able to describe and analyze architectures of software systems in uniform, comparable terms. The remainder of this paper sets forth the tasks required to do so.

3.1 *Composition of systems for subsystems*

System organizations such as the ones described above are constructed by combining subsystems. Sometimes, as for most of the examples cited, the subsystems are independently compilable modules, linked by shared data or procedure calls. At other times the subsystems

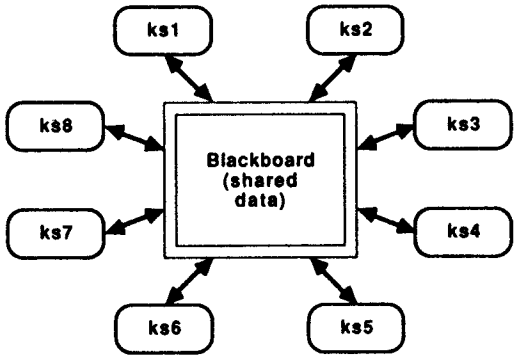

Fig. 6. The blackboard.

are sets of design decisions, for example the choice of a synchronization strategy, and the code that implements them is woven together with code for other subsystems.

Each subsystem performs some clear function within the system. In current architectural descriptions of systems, this function is often specific to the system being implemented; but it may also be a more common function such as communication or storage. For example, compilers include lexical analyzers, which are actually instances of filters. Identifying and classifying the system functions that are common to many applications is a significant first step to the development of software architecture.

Subsystems have internal structure. It is often useful to design that substructure at an architectural level before implementing it. The internal structure of a subsystem need not be the same as the structure in which the subsystem is embedded. For example, many of the filters provided with unix are not organized internally as filters. At some point, of course, it is necessary to drop from the architectural design level to the programming level. At this point, the programmer can choose from a variety of programming paradigms, such as imperative code, rule-based systems, functional systems, and table-driven interpreters.

Subsystems allow a variety of implementations of the functions they provide. The choice of implementation strategy depends on characteristics of the application. For example, the strategies for implementing the storage function include files, data bases, symbol tables, and arrays. These have different access patterns, different data models, slight variations in the detail of the function they provide. They also have different abilities to accommodate heterogeneous or variable-sized data, different response times, latencies, and overheads.

Subsystems are combined into systems in a number of ways. The most common are the usual mechanisms for combining units of code: invocation (e.g. of procedures), instantiation (e.g. of generic definitions), and replication (e.g. of progam text). In these cases, implementations of subsystems may be constrained to a single programming language, hence a single paradigm. Other mechanisms for combining subsystems, such as communication links and pipes, are not sensitive to the implementation paradigm of individual units, though they often require that the individual units observe certain protocols with respect to the communication. In cases where subsystems are not independent compilation units are composed by mingling code for one subsystem with code for another.

3.2 Results to build on

Two existing areas of work support the development of software architectures: module interconnection languages and reference models.

A number of tools and notations for defining module interfaces, inter-module dependencies, and the allocation of processes to processors have been developed, starting with the module interconnection language of DeRemer and Kron and continuing to the present [5, 12, 1]. These have emphaized generality of interconnection, and they include features such as dependency declarations, procedure signatures, and resource allocation information. They have not made a concentrated attempt to take advantage of special properties of modules such as the algebraic character of an abstract data type or the very local context in the input stream required by a true filter. To support software architectures, it will be important to find ways to recognize and exploit special properties of particular classes of subsystems. It will also be important to find ways to preserve the independence of definitions of subsystems that do not correspond to compilation units.

Reference models are developed to provide conceptual frameworks for describing architectures and showing how components are related to each other. They define the external properties of a subsystem without pre-empting design decisions about the implementation. They have been developed for network communication [8], for data base architectures [4],

and for interactive command languages [2]. These reference models show a way to define the significant features of functional subsystems and the range of variability that can reasonably be supported.

3.3 Codification and dissemination

Engineering disciplines rely on shared understanding of design and implementation strategies and standards. This shared understanding is based on extensive experience; in engineering disciplines other than software it is codified and disseminated through handbooks. Some of these handbooks cover entire fields; Perry's Chemical Engineers' Handbook [14] and Kent's Mechanical Engineers' Handbook [10] are examples. Other handbooks concentrate on specialized areas such as waste management or pesticides. These handbooks serve not only as repositories of information, but also as carriers of the conceptual structure of the field.

For software architectures, it is not sufficient to identify system structures, subsystem types, and techniques for composing subsystems into systems. These elements are sufficiently complex that they can't be reconstructed from memory, and their analysis requires detailed information about alternatives, tradeoffs, and interactions. Knowledge about them must be codified in a uniform way. Comparative knowledge must be captured in a form that permits easy reference; uniform presentation is essential to this. We must consider both the content and the form in which this knowledge is captured. The content must address issues such as

- *Informal description:* purpose of element, general domain of utility,
- *Abstract model:* basis for specification, either formal or informal,
- *Syntax:* notation, signatures of operations,
- *Semantics:* meanings of primitive elements and for the rules for composition,
- *Evaluation criteria:* correctness, generality, performance, adaptability, domain-specific power, etc.
- *Implementation:* template for structure, system support,
- *Design dimensions:* implications of design decisions,
- *Engineering considerations:* interactions, good practice,
- *Selection criteria:* indicators and contraindicators,
- *History:* primary users, major upgrades.

To understand appropriate forms for organizing and disseminating this information, we should begin by understanding how engineering hardbooks support the traditional engineering disciplines. We should then ask how innovations in electronic publishing can be exploited for this special task of providing reference information on demand.

3.4 Summary

This paper has examined the use of abstraction techniques to solve software development problems. It described the contributions these techniques have made in the past and identified the architectural level as the next area where new abstraction mechanisms are required. This is the next logical step after the procedure, type, and general module, which are currently the largest-grain abstractions in common use. Further, system descriptions already provide evidence of informal abstraction at the architectural level. These factors suggest that the step to software architecture is now possible. We identified *identification*, *codification*, and *dissemination* as the three major problem areas to attack and indicated interesting tasks within each of these areas.

Methods and tools for software architecture can potentially contribute to software production an several ways. By providing higher-level abstractions, they may reduce the need for system-level detail and, as a result, the apparent complexity of systems. By

providing a shared vocabulary for defining systems, they may simplify making changes. By providing access to a collection of well-understood organizations, they may inprove reliability. And by reducing the amount of reinvention they may help to make development times more predictably, thereby improving management control.

Acknowledgement

This work was supported by Carnegie Mellon University and the Mobay Corporation.

References

[1] M.R. Barbacci, C.B. Weinstock and J.M. Wing, Programming at the processor-memory-switch level, *Proc. 10th Internat. Conf. Software Engineering* (April, 1988).
[2] Members of IFIP Working Group 2.7, D. Beech (ed.), *Concepts in User Interfaces: A Reference Model for Command and Response Languages* (Springer, New York, NY, 1986).
[3] B.W. Boehm, *Software Engineering Economics* (Prentice-Hall, Englewood Cliffs, NJ, 1981).
[4] T. Burns et al. Reference model for DBMS standardization, *SIGMOD Record* 15 (1) (March, 1986) 19–58.
[5] F. DeRemer and H. Kron, Programming-in-the-large versus programming-in-the-small, *IEEE Trans. Software Engrg. SE-2*, 2 (June, 1976) 80–86.
[6] L.D. Erman, F. Hayes-Roth, V.R. Lesser and D. Raj Reddy, The Hearsay-II speech-understanding system: Integrating knowledge to resolve uncertainty, *Computing Surveys* 12 (2) (June, 1980) 213–253.
[7] N. Gammage and L. Casey, XMS: A rendezvous-based distributed system software architecture, *IEEE Software* 2 (3) (May, 1985) 9–19.
[8] H. Zimmermann, A standard layer model, in: P.E. Green, Jr. (ed.), *Computer Network Architectures and Protocols* (Plenum Press, New York, 1982) 33–54.
[9] F. Hayes-Roth, Rule-based systems, *Comm. ACM* 28 (9) (September, 1985) 921–932.
[10] W. Kent, *Kent's Mechanical Engineers' Handbook* (John Wiley, New York, NY, 1950).
[11] B. Liskov, Data abstraction and hierarchy, in: *OOPSLA'87 Addendum to the Proceedings*, New York, NY (October, 1987) 17–34.
[12] J.G. Mitchell et al. *Mesa Language Manual* (Xerox Palo Alto Research Center, Systems Development Department, 1979).
[13] P.A. Oberndorf, The Common Ada Programming Support Environment (APSE) Interface Set (CAIS), *IEEE Trans. Software Engrg.* 14 (4) (June, 1988) 742–748.
[14] R.H. Perry et al. *Perry's Chemical Engineers' Handbook* (McGraw Hill, New York, NY, 1984).
[15] M. Shaw, Abstraction techniques in modern programming languages, *IEEE Software* 1 (4) (October, 1984) 10–26.
[16] J.E. Tomayko, *Computers in Space: The NASA Experience* (Marcel Dekker, New York and Basel, 1987).