Implementation Languages for CACE Software

Brück, Dag M.

1987

CODEN:LUTFD2/(TFRT–3195)/1-020/(1987)

Implementation Languages for CACE Software

Dag M. Brück

STU project 86-4047

Department of Automatic Control
Lund Institute of Technology
September 1987

Department of Automatic Control
Lund Institute of Technology
P.O. Box 118
S-221 00 Lund Sweden

Document name
FINAL REPORT

Date of issue
September 1987

Document Number
CODEN:LUTFD2/(TFRT-3195)/1-020/(1987)

Author(s)
Dag M. Brück

Supervisor

Sponsoring organisation
The National Swedish Board of Technical Development (STU project 86-4047)

Title and subtitle
Implementation Languages for CACE Software

Abstract

This report discusses the selection of implementation languages when developing tools for computer aided design of control systems. Experiences have been compiled from special case studies within the project and from previous projects.

When developing large software systems, the availability of modularization, data abstraction, object-oriented programming and safe separate compilation are of great importance. The development work is also affected by the programming environment, that is, compilers, subroutine libraries, editors, graphics packages and window management. Standardization and portability must be considered as well.

We think future packages will be written using many programming languages for the following reasons: A CACE package will consist of a large number of modules (naturally falling into the areas of numeric computations, symbolic computations, user interfaces and expert systems), and it is natural to use different languages for different purposes. It is also necessary to reuse existing software, for example, numeric libraries written in Fortran. Most workstations support, although with some limitations, the mixing of modules written in different languages, such as, Fortran, Pascal, C and Lisp.

This report covers in particular Fortran, Pascal, Modula-2, Ada, C and C++, Prolog, Common Lisp, KEE and Smalltalk. These languages are either widely available, or have features of special interest. For implementation of production systems, we recommend C++ in particular, and possibly Ada, C and Modula-2. Common Lisp is recommended for expert system applications. New development in Fortran must be avoided. For exploratory programming and rapid prototyping, Common Lisp and more advanced programming environments like KEE are recommended.

Key words
Computer Aided Control Engineering, Programming languages, Programming tools, Programming environments, Modularity, Data abstraction, Object-oriented programming, Exception handling

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title

ISBN

Language
English

Number of pages
20

Recipient's notes

Security classification

Table of Contents

1. Introduction

This is the final report of the project "Implementation Languages for CACE Software" (STU project 86-4047). It has been a part of the project "Computer Aided Control Engineering (CACE)" at the Department of Automatic Control, Lund Institute of Technology.

When implementing CACE software it is not obvious which language to choose. We have obtained some insights via the other pilot projects that have been carried out. The project "New forms of man-machine interaction" (STU project 84-5069) [Mattsson et al., 1986] has given a considerable insight into the use of Pascal as an implementation language. The projects "Experiments with expert system interfaces" (STU project 85-3042) [Larsson and Persson, 1987] "Expert control" (STU project 85-3084) [Årzén, 1987], "High level problem solving languages" (STU project 85-4808) [Åström and Mattsson, 1987] and "Representation and Visualization of Systems and Their Behaviour" (STU project 86-4049) [Mattsson, 1987] have all given experiences with working in a Lisp environment. Additional Lisp experience has been gained from the study of system structures. Other projects carried out by the team members include large Fortran programs. We have also Ada experience from other projects. Modula-2 is used extensively on our IBM PC AT's for programming of real-time applications.

We felt, however, that it would be useful to have a project which was specifically devoted to implementation languages. Firstly, it would be an opportunity to compile our experiences. Secondly, it would give us possibilities to get experience of using additional languages, for example: Prolog, the purely object-oriented language Smalltalk, and other sophisticated programming environments, such as KEE. These experiences are reported in Brück [1987], Kreutzer and Åström [1987] and Denham [1987], respectively.

The report is organized as follows. In Section 2, different types of users and different uses of a CACE package are identified. The components of a CACE package (calculations, user interface, expert systems) are shortly described. Typical software development phases are also noted. In Section 3, the programming environment is discussed. This includes a number of programming tools: compilers, subroutine libraries, editors, graphics libraries and window management. An important area is the user's background and his subjective opinion of the environment. Related issues are standardization, adaptability and quality of the environment. Section 4 discusses a small number of programming language concepts of particular importance to CACE software: modularity, data abstraction, object-oriented programming, correctness and reliability. In Section 5, overviews are given on Fortran, Pascal, Ada, Modula-2, C and C++, Prolog, Common Lisp, KEE and Smalltalk. Conclusions are given in Section 6.

In this report, a *CACE package* is the goal and purpose of the discussion; a *tool* is a piece of software that a user can grab in order to accomplish a task; the *environment* is about everything in sight of the user, including all tools.

2. Users and uses

This section will cover different types of users and the different ways the CACE packages are used, major components of such a package, and the development phases of software systems.

Types of users

The uses of a CACE package can be related to three different kinds of generic users: the end user, the developer, and the implementor. Of course, a real person can perform several duties, none of which may be clearly specified; the "typical" user serves as a conceptual model.

The *end user* uses the CACE package as it is, i.e., in the way it was installed on his computer. He will use it, more or less frequently, to solve his application problems. He will only get in touch with the user interface of the package (in other areas called the top-level), and not the underlying implementation.

The end user's major concern is to get the job done as efficiently as possible. An important aspect is execution speed; although computer performance increases linearly over time, the complexity of many problems grow exponentially. Another important aspect of user efficiency is integration; the CACE package must be integrated with all the other facilities the computer can provide the end user. For example, the same user interface (e.g., editing commands and window control) should be used in both the word processor and the CACE package. It should also be possible to move equations to the document the user is preparing. Of course, execution speed and user efficiency are of no use if the CACE package is not available on the computer system the user is familiar with and has immediate access to.

The *developer* adapts and extends an existing CACE package to meet local requirements. The developments are used by the end user; developments are typically initiated by requests from end users.

We assume that the package can be changed in some ways (not merely installed on a particular computer), but that there will exist a "core" which is pre-defined and not alterable by the developer. Extendability should include facilities for adding new operations (commands), new data structures, and ways of enhancing the user interface. This indicates that there is no clear distinction between users, developers and implementors.

The developer must have easy access to a large number of utility routines, both complex and powerful (essentially all operations available to the end user), and simple routines (for example, to read and write numbers). The development language may be an ordinary programming language, which will most likely be the same language as was used for implementing the core of the package, augmented by a run-time library. It can also be a special purpose language, specificly tailored at writing CACE applications; an example of this approach is EAGLES [Gavel, 1986], with the embedded language M [Gavel et al., 1986].

The *implementor* is responsible for the core of the CACE package, which is then used by the developer and the end user. The basic facilities of the computer system will be heavily used: compilers, graphics libraries and window management. The possibility to combine parts written in different programming languages may become important. Porting the package to other computer systems, maintaining multiple versions and debugging the package, are likely problem areas for the implementor.

Components of a CACE package

This section will outline a few components or facilities that are required in a CACE package. The emphasis has been put on the implementor's view.

Numerical calculations are dominant, and will remain so as long as we can predict. There are a huge number of numerical routines that must be included in a general CACE package: NAG, LINPACK, EISPACK, ODEPACK and many more. Fortran is the dominating language for writing numerical software; the only other languages which may become more widely spread are Ada and C. Good numerical software is difficult to write, and the quality of compilers (and run-time libraries) must be checked before porting the CACE package to another computer.

Symbolic calculations are becoming increasingly popular, and there are several good reasons for providing symbolic manipulation. The user interface would be improved by allowing the user to work on a higher level. He could present his problem to the computer on a form suitable to himself. Results on analytic or symbolic form could give a better insight into structural properties than numerical tables. Even when it is not possible to carry the symbolic calculations all the way through, symbolic manipulation could be useful; symbolic manipulation could simplify the problem and transform it to a form better suited to numerical solution. Symbolic manipulation could also be used for automatic generation of procedures for calculating gradients, Jacobians, Hessians, and much more, thereby reducing the user's work load and hopefully decreasing the risk of introducing errors.

Existing packages for symbolic calculations, such as MACSYMA [MACSYMA, 1983] and muMATH [muMATH, 1983], are powerful, but require considerable computer resources. In particular, MACSYMA is best run on specially designed Lisp machines. MACSYMA is large and unmodularized; inclusion of existing well-proven routines in a CACE package is therefore almost impossible. It should be noted that existing packages, even without the integrated CACE environment, are useful tools for certain problems.

More and more attention is now drawn to the design and implementation of *user interfaces*. The style currently in fashion was introduced with Apple's Macintosh [Apple, 1985], and most workstation manufacturers follow similar ideas, for example, Sun Microsystems [Sun, 1986]. It is very important to *design* a consistent and predictable user interface, based on the limited scientific results in existence today. The overall design is more important than, for example, whether pop-up menus or buttons are used to select an operation. Luckily, there exists some general advice [Kyster, 1987], or guidelines for a computer family, leading to a consistent user interface for several packages on a particular computer.

The user interface has traditionally consisted of two parts: command decoding and presentation. Both parts are becoming increasingly complex with modern window based workstations. Input operations, including commands, can be initiated using menus, buttons, textual commands, and more. Output from the package consists of text and complex graphics, such as, animation. In addition, there are operations for handling windows: activation, deactivation, moving, resizing, just to mention a few.

With modern window based systems, the user will expect the computer to handle multiple related tasks simultaneously. The use of concurrency, real or time-shared, imposes new requirements on the software. A typical example is the "notifier" which sends asynchronous interrupts to all application programs;

a common window management approach. In particular, exception handling and error recovery must be carefully considered.

The hottest area, in particular among sales people, is so called *expert systems* or *knowledge based systems* (KBS); a good definition is beyond the scope of this report. Some promising results have been obtained, but the goal must not be set too high. In particular, expert systems can be used for the user interface, for example, to provide intelligent help systems [Larsson and Persson, 1987]. It may also be possible to design user interfaces that adapt themselves to the user's needs.

Software development phases

The implementor's work can be broken down into a number of tasks, or phases. Each of these phases are of course not isolated from the others; further more, a software system will be revised many times, leading to iterations over all phases.

The first, probably most important, and surely least understood, phase is the specification of the system. A specification is usually informal, outlining the facilities of the package. Such a specification is useful, in particular when developing an experimental system, but not sufficient for all purposes; reliability and correctness issues, to be discussed below, require a more stringent specification. Much interest has been devoted to the principles of specifications and the development of practical methods [Gehani and McGettrick, 1986]. Regrettably, all methods have some deficiences, preventing them from catching on.

There is an alternative to the tedious task of writing specifications, which may have no relevance to a quickly changing reality anyway. It is called exploratory programming, or rapid prototyping. The basic idea is to sit down in front of a highly interactive computer system, preferably with a genuine end-user, and develop a rudimentary system "on-the-fly." The prototype can then be refined, or even completely re-implemented in another programming language. The main benefits are that the end-user takes part in the development and that quick results (i.e., impressive demonstrations) can be shown. Rapid prototyping is usually done in a bottom-up fashion. The introduction of expert system tools (such as KEE) opens new possibilities in this area.

The act of writing code has been extensively covered elsewhere. Two things deserve to be mentioned though: Firstly, the concept of structured programming has been universally accepted, which also influences the choice of programming language. Lately, so called object-oriented programming has gained much support [Stefik and Bobrow, 1986]. Secondly, a CACE package is a large piece of software. There are a number of problems which arise when "programming in the large" that can be disregarded when "programming in the small." There is also a change of view on how software systems are developed: First, programming was considered a handicraft; later, programs were built, emphasising the idea of a programmer's toolbox; recently, programs are allowed to grow "organicly." Controlling organic growth will probably become a new management area.

Programming is one of the few areas where debugging an existing product is considered good engineering practice. Still, modern debugging tools help the programmer find errors more quickly than before. The most powerful interactive, symbolic, debuggers exist on Lisp machines and personal computers. Complex systems, using multiple processes and graphics, are beyond the

capabilities of current debuggers.

Documentation should not be a separate development phase, but integrated with all other activities. Window based workstations allow easy and instant access to textual documentation, but graphics can rarely be included in the document. Documentation tools aiding the user with hierarchical decomposition, on-demand explanations, and cross referencing, are now emerging [Nelson, 1980]. One example is NoteCards developed at Xerox PARC.

Nor is maintenance a separate activity; in fact, initial development could be regarded as a change from nothing to something.* All problems related to specification, documentation and programming in the large become aggravated when changes must be made in existing software systems.

* Coined by Ivar Jacobson, Ericsson Telecom.

3. The programming environment

There is more to chosing implementation tools than just looking at a few programming languages. It is also necessary to consider the implementor's programming environment: tools for program development, how the implementor fits into the environment, and how the environment can be adapted to fit the implementor.

Programming tools

The compiler (or interpreter) of the implementation language is one of the most important programming tools. Compilers are improving in speed and quality of generated code, but more important are diagnostics and the possibility to detect logical errors — "Make it right before you make it faster" [Kernighan and Plauger, 1978]. The latter issue is closely related to the language definition, but there are also some differences between compilers for a particular language. Facilities for checking array indices, overflow, illegal pointers, unused variables, etc., ought to be available in every compiler. In some cases, the deficiences of a compiler can be covered by a separate program [Johnson, 1979a].

Regardless of language, there is (sometimes) the choice between a compiler and an interpreter. The compiler should be designed to detect as many errors as possible, also when separate compilation is used. With the interpreter, less diagnostics can be accepted if additional interactive debugging facilities are provided; with interpretative systems, small modules can be efficiently tested using a bottom-up approach. A minimum set of debugging facilities are: tracing, examination of data structures, and command execution at breakpoints. Some systems provide both a "checkout" and a "production" compiler. This may be a good idea, but subtle differences between the two versions are unavoidable; an incorrect program may incidentally return correct results using one compiler but not the other, for example, because of differently initialized data areas. So called incremental compilers offer the possibility to change individual modules, without a full time consuming recompilation; in some systems, modules can be substituted in a running system.

We predict that future CACE packages will consist of modules written in many different programming languages. No single language is suitable for all applications, and packages with dissimilar backgrounds will be merged. Therefore, the possibility to freely mix modules written in different languages is essential. Real systems often provide limited capabilities in this area, for example, subroutine calls in only one direction, usage of "wrappers" for changing argument passing conventions, loss of precision, different representations of strings, high calling overhead, and much more; special attention should be drawn to this issue when chosing a programming environment.

Subroutine libraries are associated with both the programming language and with the operating system. Some combinations (e.g., C on Unix) provide a very rich selection of routines. Numerical libraries, such as NAG, and libraries for handling graphics and windows, are also required. These libraries are essential for quickly implementing a CACE package on a computer system. Regrettably, standardization of subroutine libraries is lagging far behind the standardization of programming languages. Subroutine libraries are often sensitive to changes between operating system releases. Relinking or recompilation is usually needed, and sometimes, major routines must be completely

rewritten. In any case, there will be delays between the release of the operating system and the release of a working CACE package; end users are often required to postpone certain upgrades until all application packages have been revised.

The most important tool for the programmer is the text editor; it is the tool used most of the time, and personal preferences are often debated. The editor of choice, with regard to facilities and power, is Emacs [Stallman, 1984 and 1986]. Emacs is customizable and extensible; it even contains a built-in Lisp interpreter. There are a number of "modes" which give access to high-level operations suited to a particular application. Of course, all these facilities do not come for free, and Emacs becomes slow on small time-sharing systems. Other aspects that must also be considered when chosing a text editor. Because of personal taste and previous experiences, people often like one editor more than another. Some editors, for example, *vi* on Unix systems, are readily available and can handle any type of equipment.

Graphics and window management are the most difficult, or rather least developed, areas of the programming environment [Hopgood et al., 1985]. Most workstations provide window based environments that look wonderful to people used to "dumb" terminals; when trying to use windows and graphics from a program, an overwhelming number of inconsistencies and problems arise. It is not yet possible to say whether the users lack experience of these new tools, or if the design of the tools must be changed, or both. Some experiences can be found in Brück [1986] and Mattsson et al. [1986].

Two levels of supporting software is needed: Firstly, a basic framework of drawing primitives, window operations, segment handling and picking. With some effort, a consistent and efficient design can be devised. Secondly, routines for "high-level" facilities, such as menus, buttons, cut and paste, and scroll bars, are welcome. The high-level routines speed up development work considerably, and maintains a compatible user interface among all packages on a particular computer system. No vendor-independent package of this type is yet available.

Lastly, there are a number of other useful tools, in particular in a Unix environment. The parser generator tools, Yacc [Johnson, 1979b] and Lex [Lesk and Schmidt, 1975], can be used for command parsing; defining a grammar for the command language will often produce a consistent and general user interface. It is also possible to "compile" raw input data, with excellent support for error detection and diagnosis. An interesting problem is how graphical input, for example, picking and rubberband lines, should be expressed using a context-free grammar.

So called profilers show how much the different parts of a program are executed. Output from a profiler can be used for detecting errors (for example, code that is never executed), and for detecting "hot-spots" that can be improved (programmers' guesses are often wrong), thus increasing execution speed.

An example from the Lisp world is Flavors [Cannon, 1982], a package for object-oriented programming. Flavors is quite powerful, and exists today on most workstations and many mini-computers. For maximum efficiency, Flavors must be integrated with the underlying Lisp system.

PA Consultants, Cambridge, England, has developed a commercial "software engineering toolkit" called PA-SET. It provides a number of tools for database management, window management and graphics, plus an "infras-

tructure" that ties the application program to the tools. The PA-SET will form the base of a CACSD environment funded by SERC in the U. K.

A few final words must be said about how the user (in our case the implementor of a CACE package) experiences the programming environment. The very general term quality encompasses many aspects that contribute to a good environment. Quality is bug-free system software, the absence of arbitrary limits, sensible behaviour at boundary conditions, and much more. A good design is often rejected because of a bad implementation. Programmers are also sensible to turn-around time. Interactive systems decrease the turn-around time tremendously. Multi-process systems, for example, a window based workstation, gives the user the opportunity to switch tasks and schedule his work, rather than having to sit idle.

The user's background and skills determine to a large degree the effectiveness of a programming environment. It may sometimes be better to employ techniques that are known to the users, rather than introducing the "right" thing, whatever that may be at the time. There is also a risk, in particular among less experienced end users, that people feel over-run by technology.

Standards

Standardization is a key issue in decreasing development time, both by making software more portable, and by decreasing learning time. Programming languages are quite well standardized, but most packages contain inherently unportable routines. The problem is often that programmers lack the knowledge and understanding required to write easily portable code. Ada is one of the few language where machine dependencies can be handled in the language, but it is still possible to disregard these facilities.

Standardization in other areas have not come as far. Two graphics standards exist: GKS (Graphics Kernel Standard) and PHIGS (Programmer's Hierarchical Interactive Graphics Standard). References to GKS are Hopgood et al. [1983] and Enderle et al. [1984]; to PHIGS Brown [1985] and SIS [1985]. Neither GKS, nor PHIGS, have been wholeheartedly accepted by the user community, and there is still a shortage of efficient implementations. It should be noted that PHIGS has just recently become commercially available. Window management is still far from any standards. A small number of window managers, such as, Macintosh, SunView and X-Windows, have gained considerable interest, but are fully incompatible. It is interesting to note that there exists standards for related user interfaces, for example, layout of letterhead paper or roadsigns.

A somewhat contradictory goal for the designer of a programming environment is adaptability. Adaptability implies changes, and therefore potential anarchy. In reality, some of the best environments gain their power from being customizable or even programmable: the Unix shell is a small programming language, and EMACS is a Lisp system [Stallman, 1984 and 1986]. By adapting and extending an existing tool, big improvements can be gained with little effort. The same view should of course be applied to the CACE package itself.

4. Programming language concepts

This section contains a discussion of programming language features and concepts of particular importance to CACE packages. Too much can be said about programming languages in general. In this report, we will confine the topic to a few key issues that are of particular importance when implementing a CACE package: modularization, data abstraction, and reliability. For a more profound understanding of concepts, design and implementation of programming languages, see Ghezzi and Jazayeri [1982] and Horowitz [1983].

Modularity

Development of any CACE package will involve quite a lot of programming. A CACE package will probably contain code from many different sources and for performing many general tasks (for example, command decoding). Modularization is the key to flexibility and reusability of existing code. The code must also be split up into a number of source files in order to be manageable. Common guidelines for modularization is that functions or procedures (subroutines) should be no longer than one page, and source files should not exceed 500 lines.

Separate compilation is supported by most language implementations, whether part of the language definition (as in Ada or Modula-2), or not (as in Pascal). The important issue is how well this facility agrees with the requirements for data abstraction and reliability, which are discussed below.

The ideal language should provide both protection and accessability. A module (or compilation unit) must be able to hide information which is not needed outside the module; it must also be possible to protect data areas from illegal modifications. A difficult problem in most programming languages is to exchange information between two modules via a third module, without giving the third module full access. Accessability means that all kinds of language elements should be exportable, i.e., not only data and subroutines, but also compile-time constants and type definitions.

The ideal language will ensure that all this is done in a safe and consistent way. Dependencies can be automaticly checked by the compiler, and compilation timestamps can be checked by the linker to make sure no outdated modules are used. Ada and Modula-2 come close to the ideal language in this respect. In other languages, for example, when using C with Lint and Make, dependencies can be explicitly stated by the programmer. Conventions established for a programming team must be enforced by the programmers themselves, with only little support from the programming language.

Data abstraction

The aim of programming languages has changed over time. First, the language was designed to give maximum control over the hardware (assembly language). Then it was designed to express calculations (Fortran). Today, most languages are designed to express algorithms, i.e., with an emphasis on on control structures (Algol, Pascal). This is the result of the general acceptance of "structured programming" in the late 1970's.

The trend is now going away from specifying algorithms that operate on data structures, to specifying so called abstract data types which also include operations on data objects. The programmer decides what data types are required; then the data types are provided with necessary operations, and

the internal representation of the data is completely hidden. Object-oriented programming is often explained as messages being sent to objects, expressing the idea of data objects as active entities. The object is even allowed to ignore a message, which can be regarded as argument passing through "call-by-desire." Object-oriented programming is also characterized by inheritance: common properties are made explicit by introducing classes; the objects inherit properties (data and operations) from these classes.

Well, why should we go for data abstraction? In other reports, for example, [Mattsson, 1987], the need for abstraction mechanisms in a CACE package has been identified; similar reasons apply to programming as well. Data abstraction is a change of view on how data and operations interact; it also implies new programming languages with improved syntax and semantics. In the same way as structured programming resulted in better programs in 1980, so will data abstraction and object-oriented programming yield better programs in 1990.

The data abstraction mechanism can be designed in basicly three different ways. As usual, some languages form a "grey zone" between these categories, and some experimental languages are almost impossible to label.

Basic overloading. Ada provides overloading of operators, functions and procedures. A package contains subprograms that define operations on data types which are protected from the outside. There is no dynamic binding of attributes or operators.

Classes with inheritance. Firstly, a class represents a more object-oriented view on data, with tighter coupling between data and operations. Secondly, a class can inherit properties from other classes, so called superclasses. Overloading is often used to meet special requirements for the class; general purpose routines are defined in the superclasses. The programmer also has (optional) control over object creation and destruction, and type conversions. Binding is still static. Important languages in this category are C++ and Simula-67.

Classes and dynamic binding. Smalltalk and Lisp based packages for object-oriented programming offer multiple inheritance and dynamic binding, in addition to the facilities of the previous category. Multiple inheritance means that a class can inherit properties from more than one immediate superclass, which is common with real-world objects. With dynamic binding, the type (or class) of an object is determined at run-time. Dynamic binding is sometimes convenient, but invites to an obscure usage of variables.

Every time a new concept is introduced, concern about efficiency is raised. Classes and overloading can be efficiently implemented. For example, the run-time overhead for virtual functions in C++ is about four memory references; the storage overhead is one word per object plus one word per virtual function per class (C++ was explicitly designed with efficiency in mind). In systems with dynamic binding, more work must be made at run-time. Programmers instinctively avoid advanced features when maximum performance is needed.

Correctness and reliability

Correctness is one of the basic requirements for any software package. Two approaches are used to ensure correct programs: error correction is to find and correct existing "bugs" in a software system; error prevention involves methods to produce a correct program in the first place. Only trivial programs can

easily be shown to be correct, and all major software packages contain known and unknown errors.

Testing is the most common way of achieving program correctness, but testing can only prove the presence of errors, not their absence.* This deficiency has stimulated a large amount of research in program verification, i.e., to verify that a program is correct independently from its execution. This means proving that a program is consistent with its formal specification. A few languages support formal verification, but there is no practical experience with real systems of interesting complexity.

When correctness cannot be guaranteed (which is normally the case), reliability is the second best thing. Reliability means that the system is likely to perform to the user's satisfaction, but minor or infrequent errors can be tolerated. On the other hand, correct programs can be unreliable, for example, if the specification does not reflect actual requirements, or if the specification does not state what should be done under critical circumstances.

Some programming languages naturally support safe and reliable systems. Modularity and data abstraction are essential for reducing complexity of the software system, and for enforcing well-defined properties of data objects. Exception handling (as in Ada) is a very good way to handle errors and other uncommon events. Given an easy-to-use and powerful exception handling mechanism in the programming language, the programmer is more willing to program defensively and take all sorts of errors into account. The language implementation should include optional tests for detecting overflow, out-of-range array indices and illegal pointers; these runtime errors are typical symptoms of earlier programming errors.

* After Edsger W. Dijkstra.

5. Overview of programming languages

This section contains an overview of a few selected programming languages. The languages have been selected either because they represent an "obvious" choice for implementing (parts of) a CACE package, or because they offer significant advantages over other, more common, languages. A more comprehensive overview is found in [Ghezzi and Jazayeri, 1982].

Not only the features of the language itself must be considered when selecting a programming language for a project. Availability and implementation quality are important factors. The intended application should of course have the greatest influence on the selection. Of the languages listed below, Lisp, Prolog, KEE and Smalltalk are tailored to research and exploratory programming; compiled languages, such as, Ada, Modula-2 and C++ may be more suited in a production system. Traditional languages (Fortran, Pascal and C) can usually be mixed; other combinations are less common, but it is often possible to call C and Fortran from Lisp.

Fortran

Fortran is one of the first programming languages still used, and is therefore traditionally much used for scientific computing. Good implementations are widely available; numerical and graphical subroutine libraries all have Fortran bindings. There exists today a great deal of software written in Fortran, that would be extremely costly (several million U.S. dollars) to rewrite in a superior language.

On the other hand, Fortran is in all respects an old language. The lack of control structures, data structures and recursion, plus all potentially dangerous features (no variable declarations, common block, equivalence, reference parameters), lead to a strong recommendation against using Fortran for future work.

Pascal

Pascal is the language most widely used for teaching programming. It has many good aspects, but modularization, separate compilation and data abstraction are not part of the standard definition; vendor specific extensions cover some of the deficiences, but in an unportable way. Pascal is widely available and known to most programmers.

Modula-2

Modula-2 is a new programming language heavily based on Pascal. All good features of Pascal are present, and minor problems have been rectified. The biggest improvement is the support for modularization: each module is split into two parts, a definition module and an implementation module. Modula-2 enables safe separate compilation and encourages clean modularization.

Regrettably, there is no exception handling, and no support for object-oriented programming. Still, Modula-2 displays most good features of Ada, but in much smaller and readily mastered language. The transition from Pascal to Modula-2 is easy. Modula-2 is recommended for future work, when available.

Ada

Ada has all the good features of Pascal and Modula-2, plus exception handling; tasking may be of value for CACE software. There is no support for object-oriented programming.

Ada is a large and comprehensive language, and therefore quite difficult to learn and implement. We think much of the criticism from academia is unjustified, but Ada cannot be recommended yet because of the lack of good compilers and run-time support.

C and C++

C is a very expressive language which becomes a powerful tool in the hands of an experienced programmer, but has many dangerous features [Koenig, 1986]. C is flexible, efficient, available, portable, and has become quite popular, also outside the Unix world. C is on many computers the only language fully integrated with the Unix operating system.

C++ was designed to be "a better C," by removing most of the dangers of C and by supporting data abstraction and object-oriented programming. The result is a very good programming language. The C++ translator produces C code, so portability is good. Exception handling can be added to C [Lee, 1983], but not yet to C++. C++ is highly recommended for future work; good references are [Stroustrup, 1986a and 1986b].

Prolog

Prolog [Clocksin and Mellish, 1981] is a programming language based on predicate logic and the resolution method. For more general applications, Prolog offers very powerful pattern matching, and database facilities in main memory. Some common programming tasks are awkward to express in Prolog, and the backtracking facility is sometimes more of a problem than an aid.

Prolog is well suited to symbolic computations, in particular tasks involving pattern matching [Brück, 1987]. It is probably difficult to mix an interpretative Prolog system with other languages.

Lisp

Lisp is the oldest programming language for symbolic computations. The basic data structure is the list (even numerical expressions are represented by lists), but modern dialects, for example, Common Lisp [Wilensky, 1986], support other data structures (e.g., arrays and hash tables). Common Lisp is becoming a de-facto standard [Steele, 1984]. Add-on packages, such as Flavors and Loops, provide support for object-oriented programming. Expert systems are usually based on Lisp, for example KEE. Lisp is highly interactive, often with a good programming environment, in particular on special-purpose Lisp machines. Lisp is efficient for symbolic computations, but Lisp systems are large and resource consuming (typically require 12 MB of memory on a workstation).

Lisp is highly recommended for research and rapid prototyping in the area of CACE software. A full-fledged lisp system is one of the most efficient programming environments available.

KEE

KEE is not an ordinary programming language, but a complete development system for AI applications and expert systems [Intellicorp, 1984]. An important feature of KEE is a rule system, which is normally used to represent "expert" knowledge; rules can also be used for expressing algorithms in a declarative way (as in Prolog). KEE also supports frames [Minsky, 1975], with inheritance mechanisms added. KEE has many built-in features which are easy to use; using KEE in a way that wasn't anticipated by the developers of KEE is less pleasing.

KEE is costly and requires a powerful computer; it is therefore only available on Lisp machines and upper-range workstations. KEE is highly recommended for research and rapid prototyping.

Smalltalk

Smalltalk is the first truly object-oriented programming language. It was originally developed at Xerox PARC in the early 1970's, and has influenced all other research in object-oriented programming.

The current version, Smalltalk-80 [Goldberg, 1984], is not only a programming language, but a complete development environment with file handling, text editing and other support functions. Regrettably, this implies that a Smalltalk application cannot be mixed with other languages. Smalltalk is an interesting research language, also in the CACE area [Kreutzer and Åström, 1987].

6. Conclusions

Future CACE packages will consist of a number of related modules. These modules will originate from many sources, and will be written in different programming languages. One of the key issues in developing a CACE package is therefore flexibility and extendability; the possibility to reuse previous work and take advantage of existing competence. It must be possible to mix different programming languages.

Data abstraction and object-oriented programming are probably the best methodologies for program development. The best languages are Common Lisp (with Flavors or Loops) and C++. Interactive environments, like those on lisp machines or like KEE, are particularly tailored at research and rapid prototyping. Ada and C are also possible, and offer the advantages of good exception handling. Safe separate compilation is found in Ada and Modula-2, for example. Fortran cannot be recommended for any future development work, but the large existing base of subroutine libraries should be used when possible.

Finally, other aspects of the programming environment should be considered. Required tools are: a text editor, a debugger, a graphics package, window management, and possibly compiler generators and profilers. A difficult problem is to provide flexibility and adaptability to specific needs and preferences, while still maintaining some level of standardization.

Acknowledgements

This work was supported by The National Swedish Board of Technical Development (STU), project number 86-4047.

This report draws on experiences from special case studies performed in the CACE project, as well as other projects at the Department of Automatic Control in Lund. The case studies have been carried out by Mats Andersson and Mike Denham (Lisp and KEE), Dean K. Frederick (window management) and Wolfgang Kreutzer (Smalltalk).

The author wishes to thank Dr. Sven Erik Mattsson for his encouragement, criticism and helpful comments.

References

APPLE COMPUTER, INC. (1985): "The Macintosh User Interface Guidelines," in: *Inside Macintosh, Volume I*, Addison-Wesley, Reading, Mass., USA.

ÅRZÉN, K-E. (1987): "Realization of Expert System Based Feedback Control," CODEN: LUTFD2/TFRT-1029, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

ÅSTRÖM, K. J., and S. E. MATTSSON (1987): "High-Level Problem Solving Languages for Computer Aided Control Engineering," CODEN: LUTFD2/TFRT-3187, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

BROWN, M. D. (1985): *Understanding PHIGS – The Hierarchical Computer Graphics Standard*, Template, San Diego, CA, USA.

BRÜCK, D. M. (1986): "Implementation of Graphics for HIBLIZ," CODEN: LUTFD2/TFRT-7328, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

BRÜCK, D. M. (1987): "Simplification of Expressions using Prolog," CODEN: LUTFD2/TFRT-7364, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

CANNON, H. I. (1982): "A Non-Hierarchical Approach to Object-Oriented Programming," unpublished report, Artificial Intelligence Laboratory, MIT, Cambridge, Mass., USA.

CLOCKSIN, W. F. and C. S. MELLISH (1981): *Programming in Prolog*, Springer-Verlag, Berlin-Heidelberg, FRG.

DENHAM, M. J. (1987): "Knowledge Representation in Systems Modelling," CODEN: LUTFD2/TFRT-7365, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

ENDERLE, G., K. KANSY and G. PFAFF (1984): *Computer Graphics Programming (GKS—The Graphics Standard)*, Springer-Verlag.

HOPGOOD, F. R. A., D. A. DUCE, J. R. GALLOP and D C. SUTCLIFFE (1983): *Introduction to the Graphical Kernel Standard (GKS)*, Academic Press.

HOPGOOD, F. R. A., D. A. DUCE, E. V. C. FIELDING, K. ROBINSON and A. S. WILLIAMS (Eds.) (1985): *Methodology of Window Management*, Proceedings of an Alvey Workshop at Cosener's House, Abingdon, UK, April 1985, Eurographic Seminars, Tutorials and Perspectives in Computer Graphics, Springer-Verlag.

GAVEL, D. T. (1986): "Introduction to EAGLES," Lawrence Livermore National Laboratory, California, USA.

GAVEL, D. T., C. J. HERGET and B. S. LAWVER (1986): "The M Language, An Interactive Tool for Manipulating Matrices, Systems and Signals," Lawrence Livermore National Laboratory, California, USA.

GEHANI, N. and A. D. MCGETTRICK (Eds.) (1986): *Software Specification Techniques*, Addison-Wesley, Reading, Mass., USA.

GHEZZI, C. and M. JAZAYERI (1982): *Programming Language Concepts*, John Wiley & Sons, New York, USA.

GOLDBERG, A. (1984): *Smalltalk-80 — The Interactive Programming Environment*, Addison-Wesley, Reading, Mass., USA.

HOROWITZ, E. (1983): *Fundamentals of Programming Languages*, Springer-Verlag, Berlin-Heidelberg, FRG.

INTELLICORP (1984): *Knowledge Engineering Environment (KEE) — User Manual*, Intellicorp, Menlo Park, CA, USA.

JOHNSON, S. C. (1979a): "Lint, a C Program Checker," in: *Unix Programmer's Guide, Volume II B.*

JOHNSON, S. C. (1979b): "Yacc: Yet Another Compiler-Compiler," in: *Unix Programmer's Guide, Volume II B.*

KERNIGHAN, B. W. and P. J. PLAUGER (1978): *The Elements of Programming Style*, Second edition, McGraw-Hill, New York, USA.

KOENIG, A. R. (1986): "C Traps and Pitfalls," Computing Science Technical Report No. 123, AT&T Bell Laboratories, Murray Hill, New Jersey, USA.

KREUTZER, W. and K. J. ÅSTRÖM (1987): "An Exercise in System Representations," CODEN: LUTFD2/TFRT-7369, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

KYSTER, H. (1987): "Menneske-Maskin-Samspil," EC-Rapport ECR–199, Elektronikcentralen, Hørsholm, Denmark.

LARSSON, J. E. and P. PERSSON (1987): "An Expert System Interface for Idpac," CODEN: LUTFD2/TFRT-3184, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

LEE, P. A. (1983): "Exception Handling in C programs," *Software—Practice and Experience* **13**, 389–405, May 1983.

LESK, M. E. and E. SCHMIDT (1975): "Lex — A Lexical Analyzer Generator," in: *Unix Programmer's Guide, Volume II B.*

MATTSSON, S. E., H. ELMQVIST and D. M. BRÜCK (1986): "New Forms of Man-Machine Interaction," CODEN: LUTFD2/TFRT-3181, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

MATTSSON, S. E. (1987): "Representation and Visualization of Systems and Their Behaviour," CODEN: LUTFD2/TFRT-3194, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

MACSYMA (1983): *MACSYMA Reference Manual*, The Mathlab Group Laboratory for Computer Science, MIT, Cambridge, Mass., USA.

MINSKY, M. (1975): "A Framework for Representing Knowledge," in Winston, P. H.: *The Psychology of Computer Vision.*

MUMATH (1983): *muMATH-83 Reference Manual*, The Soft Warehouse, Honolulu, Hawaii, USA.

NELSON, T. H. (1980): "Replacing the printed word: A complete literary system," in Lavington (Ed.): *Information Processing 80*, IFIP 1980, North Holland.

SIS (1985): *Datorgrafi—PHIGS, Programmers Hierarchical Interactive Graphics Standard*, Technical report no. 306, SIS—Standardiseringskommisionen i Sverige, Sweden.

STALLMAN, R. M. (1984): "EMACS: The Extensible, Customizable, Self-Documenting Display Editor," in Barstow, D. R., H. E. Shrobe and E. Sandewall: *Interactive Programming Environments*, McGraw-Hill, New York, USA.

STALLMAN, R. M. (1986): *GNU Emacs Manual*, Fourth Edition, February 1986, Free Software Foundation, Inc., Cambridge, Mass., USA.

STEELE, G. L. (1984): *Common LISP: The Language*, Digital Press, Bedford, Mass., USA.

STEFIK, M. and D. G. BOBROW (1986): "Object-Oriented Programming: Themes and variations," *The AI Magazine* **6**, No. 4, Winter 1986.

STROUSTRUP, B. (1986a): "An Overview of C++," *SIGPLAN Notices* **21**, 10, 7–18, October 1986.

STROUSTRUP, B. (1986b): *The C++ Programming Language*, Addison-Wesley, Reading, Mass., USA.

SUN (1986): *SunView Programmer's Guide*, Sun Microsystems, Inc., Mountain View, California, USA.

WILENSKY, R. (1986): *Common LISPcraft*, W. W. Norton, New York, USA.