TOWARD A SYSTEM'S ENVIRONMENT FOR COMPUTER ASSISTED PROGRAMMING

Carlos J. Lucena
Potifical Catholic University
of Rio de Janeiro

Donald D. Cowan
University of Waterloo

CS-76-06

January 1976

TOWARD A SYSTEM'S ENVIRONMENT FOR COMPUTER ASSISTED PROGRAMMING

by

Carlos J. Lucena
Informatics Department
Pontifical Catholic University
of Rio de Janeiro
Brasil

Donald D. Cowan
Computer Science Department
University of Waterloo
Waterloo, Ontario
Canada

Key Phrases Modularity, Programming languages,
Programming systems.

CR Categories 4.2

This research was supported in part by grants and contracts from:
The National Research Council of Canada,
The Canadian International Development Agency,
IBM (Canada).

TOWARD A SYSTEM'S ENVIRONMENT FOR COMPUTER ASSISTED PROGRAMMING

Carlos J. Lucena and Donald D. Cowan

Abstract

This report describes the design approach being used for the specification of a system's environment for computer assisted programming. The proposed system is aimed at supporting some major results in programming methodology such as specification and correctness techniques, techniques for program modularity, the representation of abstract data types and program portability and efficiency. Instead of providing a list of the desirable features for a system with the above general goals the authors attempt to present a sound logical argument on why and how the environment's components are interrelated. The proposed system attempts to amalgamate a number of apparently diverse notions related to specification and programming language mechanisms.

1. Introduction

Research in programming methodology has suggested that the development of programming systems should be supported by other programming systems generally known as environments for program development. Practically all of the research efforts under way to produce such environments are still at the design phase. There are several reasons for this situation. First, the field is very new. Second, the required features for a system's environment for program development dictate the production of complex software which requires careful thinking before implementations are produced. Finally, implementation techniques for the most promising programming methods and language features currently known are not precise enough to enable automatic procedures to be built. This report attempts to do more than present a list of the features that we intend to incorporate in our programming system's environment. We opted for focusing attention on the basic principles that need to be followed in the design of one such system, as well as the justification of the engineering decisions already made toward the construction of our system.

Two major differences distinguish the design of our system from most proposed systems. First the system is not centered in one single linguistic level. Second, the system is aimed at supporting both the specification and the implementation of each level in its multi-layered structure.

We believe our system differs from most of systems proposed in the literature (e.g. [1,2,3]). These systems can generally be described as a central language processor (for a high level language, an extended high level

language or at most a very high-level language) surrounded by a number of software tools that support the development of programs in that language and hence use only one linguistic level. Figure 1 outlines the system's architecture for such systems.

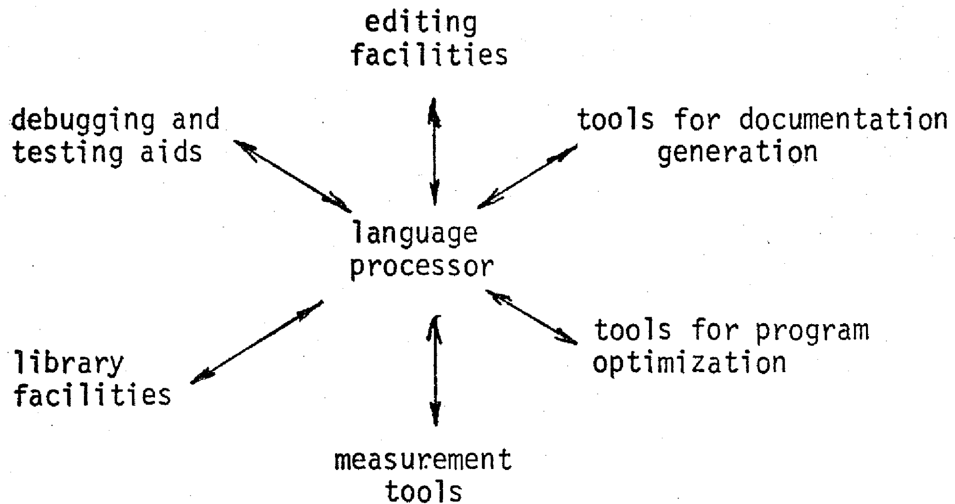


Figure 1 System's environment based on a single programming language

Most proposed systems stress program reliability and so the basic software tool supported is either a semi-automatic verification facility [4] or a powerful testing facility [1]. These systems also were mostly engineered to support one programmer in developing a program in a high-level language (FORTRAN, PASCAL, ECL, etc.).

The development of large software packages for both applications and systems is quite common and one questions whether the tools discussed in the previous paragraph will support such activities. These doubts arise because

most of these systems are convenient for only one programmer and hence, they do not support a strong form of communication among the programmers and designers of software systems. Communication in this sense means that the system does not provide the facility for a formalized form of program documentation which clearly states the specifications of a module. Consequently, the programmer would not have a complete view of the program he has to develop and would have difficulty constructing it completely and accurately.

Our proposal describes a system's environment whose goal is to have several programmers at different levels of the software development process developing a relatively large software system. Some examples of these different levels might help to clarify our meaning. In constructing a system one usually specifies the modules that are required, their input-output requirements and files and the interrelationships among various modules. Eventually this specification is converted to a linking language which is executed by the operating system. As a second step, the modules have to be specified in more detail so that they can be properly implemented in a suitable programming language. These two steps of specification and implementation describe two different levels of software development. A complete discussion of the choice of levels and the concept of modularity is contained in [5].

The goal that was mentioned in the previous paragraph imposes one strong requirement on the design of the system's environment. Communication intra-level and inter-level within the environment has to be very precise. For that reason, our system's environment requires that all the programming done in the system be driven by documentation. It means that the documentation

at every level has to be encoded in a formal notation and verified for consistency before any code at that level is produced to implement a program segment. Other requirements are also suggested by the above general goal. Each level of programming in the software development process should be able to express itself through a linguistic level that models conveniently the abstractions that occur at that level. This fact induces the idea of interrelated linguistic levels (the number and characteristics of each level remaining to be determined). Finally, it is required of the systems design that it supports in all its levels a very clear and operational concept of modularity, since we aim at supporting groups of programmers working simultaneously for the production of various parts of the same software system. Figure 2 displays the over-all organization of our proposed system.

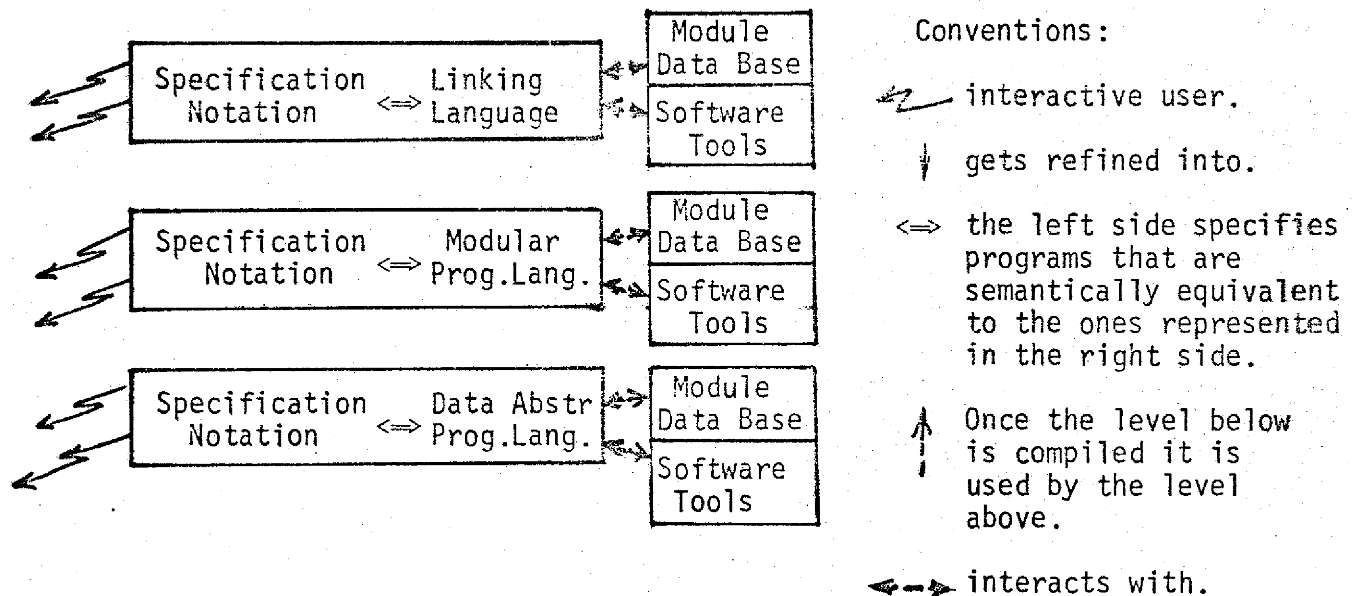


Figure 2 Multi-layered system's environment

After examining the requirements imposed by the general goal, we turn to more specific services that the system's environment is intended to support. The system should support:

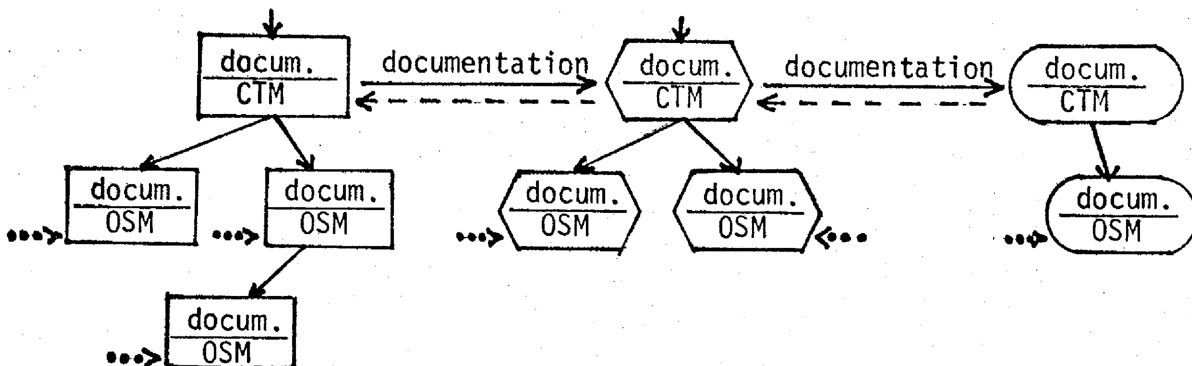
a) Off-the-shelf modular programming;

b) The development of custom-tailored program modules;

c) A design strategy through which the management of a software project is able to reject as soon as possible in the process of development, a given software product being developed with the aid of the environment. In other words, the need for iteration in the design and implementation process should be recognized as early as possible. The indicators to be used for that purpose are: correctness, robustness (tolerance to errors) and performance;

d) The development of portable programs.

Figure 3 presents an overview of the system's environment in operation.



Conventions:

CTM - Custom-tailored module
OSM - off-the-shelf module
⟶ - documentation is transferred for the encoding of the next level down.

←--- compiled module is returned.
•••> - from the modules' library.

Note: Incomplete levels (that is, specified CTMs together with OSMs) can be executed symbolically.

Figure 3 System's environment in operation

Two important strategies will be used in the design of our system's environment. The sub-systems that support the specific linguistic levels can be built and used independently. It is not our intention to reinvent notations and language features; in fact, we incorporated into our system (sometimes in an adapted version) several programming mechanisms that have been proposed by different authors.

2. On Modules and Modular Programming

In a previous report [5] we attempted to define precisely the concept of module as it is used in the context of programming systems. There we characterized modularity as a property possessed by a given linguistic level of programming. A linguistic level is a formal language with a distinguishing syntax and semantics. We require modules to be programming units characterized at given linguistic levels which possess the properties of syntactic non-interference, semantic context-independence, composition by nesting, definitional completeness and data generality. The names of the properties suggest their intended meanings (the reader is referred to [5] for details). The last two properties in the list are not satisfied by either contemporary or recently proposed programming constructs and yet are fundamental for the development of our system's environment.

By definitional completeness we mean that modules have always to carry an operational definition associated with its implementation code. In the case of a custom-tailored module the definition drives the synthesis of the implementation (not in the sense of automatic compilation) and in the

case of an off-the-shelf module the definition serves to completely identify modules in the module library. Furthermore, the documentation (definition) establishes the link between the linguistic levels: it must be possible to make consistency checks between successive levels of documentation that occur in the step-wise refinement process. In our system's environment, the following decisions were made about documentation (definition, specification):

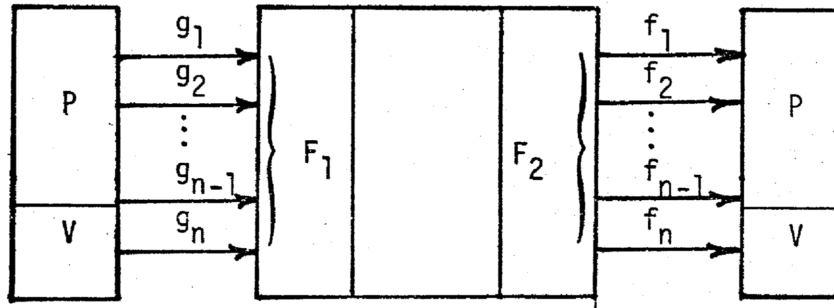
a) Documentation will have a non-procedural form at all linguistic level and will be machine processable.

b) It must be possible to make consistency checks intra-level and inter-level based on the documentation notation.

c) The documentation notation will be sufficiently formal to allow interpretive symbolic runs of the programs they describe at a given linguistic level (not necessarily the generation of code to a base machine).

By data generality we mean the possibility of inter-communication between modules via arbitrary data structures. Without this property there is no way a programmer can encode his module without some knowledge about the internal workings of the modules he will be using. Data generality will be enforced in our environment through two program mechanisms. The first method assumes the existence of two types of universal data structures in the system's environment: physical media representations and a virtual generalized representation. The physical media representations stand for the representations of data dictated by the standardized access mechanisms adopted within the environment for the various mass storage devices available

to the system. The virtual generalized representation is not related to any specific storage medium and is universally known by users of the environment. This method, which we shall call the conversion function method, requires that each module possesses two sets of mapping functions F_1 and F_2 that allow for the conversion to and from the universal data structures (communication's media) to the module's internal representation [5]. Figure 4 illustrates the outlined method.



P - Physical media representation
V - Virtual generalized representation

Figure 4 Conversion function method for data generality

The above method handles the problem of data generality between compiled modules. The second approach is called the type descriptor's transmission method. In [6] a type descriptor is an extension of the concept of the SIMULA [7] class. It allows the transmission of type descriptors between modules. Values of type type (data abstractions) can be transmitted

since there are variables, parameters, and functions in the programming language of type type. The transmission of type descriptors allows the postponement of strong type checking until the moment when modules are assembled together. Since type descriptors typically do not change during execution, it is possible to perform static association of type descriptor parameters and hence perform static type checking. Conditions for static type checking and parameter association can be found in [6]. The type descriptor's transmission method allows for module composition with data generality at the two lower linguistic levels discussed below.

We adopted a three-level model as the fundamental way of referring to the linguistic levels defined within the environment. In fact, there are more than three levels since two of them have access to common base languages that provide a uniform way of referencing data structures. The three basic levels that compose the framework on which the environment is built are called: system-level,program-level and data-abstraction level. No automatic translation takes place between levels for the synthesis of custom-tailored modules: each level is individually hand-coded by programmers at that level. Each level induces a programming system within the environment that can be implemented and used separately. Nevertheless, we believe that the major strength of the system is the capability of interconnection between levels and the software management techniques that it suggests.

3. Major System Features

The system-level is the linguistic level defined by the system's specification language PSL: a Problem Statement Language [8]. PSL is

essentially a convenient notation for encoding the system's graph (representing control and data flow) that usually appears in the early phases of programming system's design. In particular, PSL describes a system's network with two kinds of nodes: process and data element nodes. Directed links are found in the network between process and data element nodes, but not between process nodes or between data element nodes. The links connect a process with its input and output data. The nodes of the graph are described in a COBOL-like manner through program segments called sections. The basic sections of PSL are the following:

Component of the system	SECTIONS
Documents	INPUT
	OUTPUT
History Data Information	SET
	ENTITY
	RELATION
Data Definition	GROUP
	ELEMENT
Process Definition	PROCESS
Conditional Control	EVENT
	CONDITION
	INTERVAL
Operational Interfaces	INTERFACE
System Parameters	SYSTEM PARAMETER

PSL being a formal language, it allows a PSL problem statement to be analysed for consistency and other properties through a query language called PSA (Problem Statement Analyser). Recently reported efforts [9] to generate code from PSL have taken the approach of describing further the semantics of a PSL section by means of a COBOL-like programming language. Even though we are not interested in generating code automatically from this level, we will be able to use some ideas about modifications to be introduced in the PSL language identified during this effort. The unstructured data type ELEMENT, for instance, could have its domain described in a manner similar to PASCAL [10]. In our system, the procedure part of a section will be called the input-output state description part. This part of the PROCESS section is the last one to be written in a problem statement. Since PSL describes transformations and data, the input-output state description part of a section will comprise a set of assertions that describe in a non-procedural manner the input and the output states of a process (for instance, these assertions will state explicitly the subsetting criteria relating ELEMENTs to GROUPs etc.). Instead of generating PSA reports our system will be able to perform symbolic executions with the PSL source code. As soon as some statistics are gathered on the average performance of PSL operations (UPDATE, DERIVE, etc.) when actually implemented, the designer will be able to compare time requirements for a system, with times obtained from simulated runs done with the problem statement. This simulation feature will probably enhance the programmer's analytical capability at the system's level. PSL will then be compiled into a linking language which will combine off-the-shelf modules with the code for newly developed modules.

All the information about a given process (text of the process section in the PSL source and associated data sections) are transferred by the system to a programmer in charge of a module at the program-level (in case a custom-tailored module needs to be developed). At the program-level the programmers will rewrite this inherited documentation in the form of an extended version of Parnas' software module specification language [11]. The constituents of a program module specification will be in our case:

a) the set of possible values;

b) data types specification, including axioms for operations on the types;

c) limited values;

d) parameters;

e) effects.

Items a) and b) are the input-output assertions of the module. Item b) is a suitable representation of an algebraic specification of the abstract data types that will be used by the module (as in [12]). Coming from the specification-level of documentation to the program-level will require the creation of a set of input, output and program variables. These variables will have to be declared as being of certain abstract types and the input-output assertions will be expressed in terms of them. The content of the input-output state descriptions and the module input-output assertions are the same although they are expressed in a different way. Consistency checks can be carried out mechanically between the two levels of specification (e.g. by generating tables of values from the respective sets of assertions).

The programming language to be used at this level is a very high-level language (using abstract data types and the type descriptor transmission technique) called MOSAICO [13]. Experiments with this level of language are being conducted with an extended version of PL/I. Since implementations (a very high-level program) and program specifications are interchangeable, that is, in the absence of an implementation the operations defined on the abstract data type may be interpreted symbolically, we can as in the previous level "test the specification". Actual code for the implementation can only be generated when data types are implemented.

The data-abstraction level receives the data type specifications from the program module plus some requirements about precision and efficiency. The cluster mechanism [14] that we use for the data abstraction level comprises a common language [15] which refers to representation level clusters. It allows us to have a library of "base machines" which can be selected according to their performance in a given application [16]. Portability should also be achieved using the library of base machines, since compilers can be written to implement the base machines on different computers. The same test data used for symbolic runs with the program module specification can be used for actual runs with the code generated for the very high-level language program (eventually supported by instrumentation features to check the performance requirements).

Before the code generated for the program-level is linked with the other modules at the system-level, the program module is extended with conversion functions to satisfy the criterion of data generality.

4. Conclusions

We have discussed the central concepts of a system's environment for computer assisted programming. We have avoided practical issues such as the editing facilities required by a system's environment [17] to concentrate on the problem of levels of language. As the implementation work progresses we are sure that some practical restrictions will interfere with the conceptual design level and some design changes may have to be introduced.

References

[1] Ramamoorthy, C.V., Ho, S.F., "Testing Large Software with Automatic Software Evaluation Systems", Proceedings of the International Conference on Reliable Software, 1975.

[2] Culpepper, L.M., "A System for Reliable Engineering Software", Proceedings of the International Conference on Reliable Software, 1975.

[3] Wegbreit, B., "ECL Programming System", Proceedings of the Fall Joint Computer Conference, 1971.

[4] Good, D.I., London, R.L., Bledsoe, W.W., "An Interactive Program Verification System", Proceedings of the International Conference on Reliable Software, 1975.

[5] Cowan, D.D., Lucena, C.J., Staa, A.v., "On the Concept of Modules in Programming Systems", Technical Report CS-76-05, University of Waterloo, 1976.

[6] Staa, A.v., "Data Transmission and Modularity Aspects of Programming Languages", Research Report CS-74-17, Department of Computer Science, University of Waterloo, 1974.

[7] Dahl, O.J., et al., "The Simula 67 Common Base Language", Norwegian Computing Centre, Oslo, 1968.

[8] Teichroew, D., Bastarache, M.J., "PSL User's Manual", ISDOS Working Paper No.98, Department of Industrial Engineering, The Univ. of Michigan, 1975.

[9] Nunamaker, J.F., Konsynski, B., "Progress Report on Automatic Code Generation from PSL", Management Information Systems, Univ. of Arizona, 1975.

[10] Wirth, N., "The Programming Language Pascal", Acta Informatica 1, 1971.

[11] Parnas, D.L., "A Technique for the Specification of Software Modules with Examples", CACM, vol.15, No.12, 1972.

[12] Guttag, J.V., "The Specification and Application to Programming of Abstract Data Types", Technical Report CSRG-59, Univ. of Toronto, 1975.

[13] Staa, A.v., Lucena, C.J., "MOSAICO: A Language for Modular Programming", to appear.

[14] Liskov, B.H., Zilles, S.N., "Programming with Abstract Data Types", Proceedings of ACM SIGPLAN Symposium on Very High Level Languages, 1974.

[15] Schwabe, D., Lucena, C., "Specification and Uniform Reference to Data Structures in PL/I", Research Report, Computer Science Department, Pontificia Universidade Catolica do Rio de Janeiro, 1976.

[16] Low, J.R., "Automatic Coding: Choice of Data Structures", Stanford University, Computer Science Department, STAN-CS-74-452, 1974.

[17] Donzeau, V. et al., "A Structure Oriented Program Editor: A First Step Towards Computer Assisted Programming", Rapport de Recherche No.114, Institut de Recherche d'Informatique et d'Automatique, 1975.