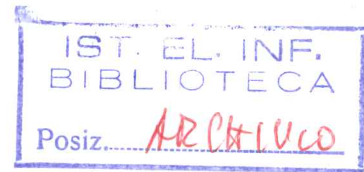

IST. EL. INF.
BIBLIOTECA
Posiz. ARCHIVIO

F82-22

TOWARDS THE DERIVATION OF AN EXPERIMENTAL PROGRAMMING ENVIRONMENT FROM LANGUAGE FORMAL SPECIFICATIONS

Roberto Barbuti, Marco Bellia and Alberto Martelli
Istituto di Elaborazione dell'Informazione C.N.R.
Pisa, Italy

Enrico Dameri and Carlandrea Simonelli
Systems & Management SpA
Pisa, Italy

Pierpaolo Degano and Giorgio Levi
Istituto di Scienze dell'Informazione
Università di Pisa, Italy

Abstract

The paper is concerned with a methodology for deriving an integrated programming environment for a specific programming language, from a set of language independent tools and a formal definition of the programming language. The approach is to define some general tools (abstract syntax manipulation, editing and debugging commands, a parser generator, an executable specification meta-language) which can be used to obtain a set of granular language-dependent tools (syntax driven editor, parser, static analyzer, interpreter, debugger, code generator) from syntactic and semantic language definitions.

1. INTRODUCTION

A major problem, in the development of computer applications, is the efficiency of the software production process. In fact, today's technology makes hardware available at a very low cost, while the overall cost of the software production (software design, development, testing and maintenance, including project management) becomes more and more critical.

The efficiency of the software production process is related to the cost of the production and to the product quality. Reliability is one of the most relevant features affecting software quality.

The main goal of the research activity in the software engineering area is increasing software productivity and reliability.

This goal has been pursued following three major lines:

- the definition of suitable high level programming languages;
- the definition of programming and project management methodologies;
- the development of tools which support programming and management activities.

In this paper, we are concerned only with programming systems, i.e. collections of tools which help managers, designers and programmers in program editing, testing, analysis, documentation, configuration, maintenance.

Conventional programming systems do not support some programming activities (for instance, semantic oriented program analysis and program configuration), nor they fully exploit the facilities offered by interactive systems, nowadays in widespread use. Moreover, they do not supply tools for program/project documentation, for managing the project status, and for communication

---

This work has been supported by C.N.R. Progetto Finalizzato Informatica, Obiettivo CNET.

within the project team. Finally, conventional programming systems consist of a collection of loosely coupled tools, which are hard to combine, which have non-uniform interfaces and do not allow a soft switch from tool to tool.

The current trend is moving from programming systems towards integrated programming environments, whose main features are:

- a more comprehensive machine support to the programming and management activities;
- tool integration;
- tools to enable an easy and natural communication between man and machine and among individuals.

Any integrated programming environment has to be designed taking care of the peculiarities of the organization, which it is addressed to. Since the integrated programming environment is, in turn, an expensive software system, it is necessary to single out a minimal environment, i.e. a set of tools, which must be contained in any integrated programming environment. Moreover, the minimal environment must provide mechanisms to extend the basic set of tools so as to fit with the requirements coming from a specific organization.

A great deal of activities supported by an integrated programming environment, e.g. specification and design, testing, validation, performance evaluation, strongly depend on specific methodologies. Therefore, the minimal environment should contain no tools which enforce specific methodologies. Rather, it should provide only those basic functionalities and mechanisms, which allow to build any tool.

The major contributions to the definition of the general features of today's integrated programming environments come from interactive LISP systems (1, 2) and from the operating system UNIX (3).

The very first programming environments have been developed within the LISP community. One reason for this arises because LISP users are quite demanding, since they face difficult problems and deal with large and complex programs. Moreover, LISP has some features which naturally lead to the development of tools. Let us mention the equivalence between data and programs, the internal tree representation (close to the abstract syntax representation of the program), the embryonal data base and data base management systems provided by the property list abstraction. Finally, the interactive interpreter-based implementation promotes the development of debugging and testing tools, and provides a flexible and powerful command language (LISP iteself!).

The main contribution from UNIX-based systems is the idea of defining a machine-independent programming environment. The Programmer's Workbench (PWB) (4) is a collection of tools, running on a system (host machine) fully devoted to program development and maintenance. The host machine can be linked to application specific target machines, to act as a remote job entry. PWB contains tools for documentation, for managing a project data base, for selective communication within a team. The most relevant features of the underlying operating system are the hierarchycal file system and the extendible job control language, which is the basis for tool composition. The choice of a single host machine allows the definition of standards and results in lower costs for training and for development of tools for different target machines.

The first integrated programming environment providing most of those features, which are nowadays considered necessary was, the Program Development System (PDS) (5, 6). The main components of PDS are:

- a program data base, which allows the user to manage item versions, development histories and dependencies among items;
- a command interpreter, wich provides an interactive user interface;
- a set of tools for program editing, analysis and execution. The integration among the tools is achieved by a common internal representation (program abstract syntax), by sharing the data base, by allowing mutual tool invocation. The tools are granular, i.e. every tool corresponds to a specific simple functionality. For example, a compiler is decomposed into a set of granular tools, which includes a parser, a type-checker, a scoper, an optimizer, a code generator.

Most of the ideas of PDS underlie the STONEMAN proposal for the programming environment for the ADA language (7). The innovative aspect of such a proposal is concerned with the portability of the environment. In order to achieve portability:

- the tool implementation language should be a suitable ADA extension with primitives for tool writing, e.g. an operation to transform a data structure (compiled code) into a running process;
- the ADA abstract machine needs to be further extended to contain suitable abstractions for those operating system functionalities which would make the environment partially dependent on the underlyng operating system, e.g. resource management and file system organization.

The resulting STONEMAN proposal defines an abstract machine, the Kernel of ADA Programming Support Environment (KAPSE), which contains all the above outlined extensions.

Let us summarize those features which are common to all the environments currently under development (8).

1. Integration, which requires:
   - common internal program representation,
   - uniform inter-tool interfaces,
   - achievability of tools from other tools.
2. Open-endedness, which require the kernel to provide the user with mechanisms for new tool definition.
3. Granularity of tools.
4. Interactivity which relies on:
   - a simple yet powerful command language,
   - a uniform user interface,
   - user oriented interaction devices.
5. Multiuser support, including:
   - a project data base,
   - personal work-stations,
   - an inter-use communication facility.
6. Host environment, i.e. the environment is supported by a dedicated machine different from those on which the produced programs will run.
7. Environment portability, obtained by re-implementing the kernel only.

2. THE PROGRAMMING CYCLE IN AN INTEGRATED PROGRAMMING ENVIRONMENT.

The tools of the minimal environment can be grouped into two families.

- Tools which do not depend upon the programming language and which can be viewed as standard operating system primitivies. Most of the tools in the kernel (7), as well as the project data base, the communication and the management tools belong to this family, and can be shared by environments for different languages, provided that inter-tool interface are kept homogeneous so as to allow easy integration with language specific tools.
- Tools which depend upon the syntax and/or the semantics of the programming language supported by the environment. Tools in this family cover all the activities which range from program editing to compiling, linking and debugging.

In this paper we will be mainly concerned with tools within this family. Our aim is suggesting a methodology which allows to define some meta-tools which can be used to derive language specific tools, given a formal definition of the programming language. Let us first look at the functionalities of the different tools in the framework of an integrated programming environment.

The standard programming cycle requires the strictly sequential use of the text editor, compiler, linker-loader and debugger. The efficiency of such a cycle can be substantially improved by supplying the user with interactive, granular and integrated tools. As an example, a programmer in such an environment is allowed to perform the following actions.

1. Edit the program.
2. Check the syntactic correctness.
3. Repeat Steps 1 and 2 until syntactic correctness is established.
4. Perform some other static analysis, e.g. type cheking or systematic static testing.
5. Repeat Steps 1-4 until necessary.
6. Interactively run the (possibly incomplete and incorrect) source language program.
7. Repeat Steps 1-7 until the program behaves properly.
8. Perform source code optimization.
9. Generate object code.

The above example shows that the nature of the tools in an integrated programming environment leads to deep modifications in the pattern of tool composition and sequencing, which results in:

- efficient error detection, since all the static analysis tools are applied separately without repeating time wasting compilations;
- early error detection, due the possibility of analysing, testing and executing programs, which

may be only partially defined and, in any case, are not yet compiled nor linked;

- efficient error correction, since as soon as an error is detected by any analysis or testing tool, a correction may be performed by resorting to the editor and subsequently resuming the suspended activity.

The general idea underlying such a pattern is to interactively perform the analysis of programs along with their development, contrasted to a "batch" philosophy which requires the program to be fully defined before attempting any analysis. Program execution tools should operate at the source language level, thus allowing the programmer to better understand the behaviour of his program and to interact with the execution tools.

3. INTERNAL REPRESENTATION OF PROGRAMS AND ITS MANIPULATION

In order to allow integration of the tools in the environment, they must operate on the same internal representation of programs (abstract representation). Such a representation has a tree-like form and is defined according to the abstract syntax of the language. Abstract syntax has been introduced in order to facilitate the definition of programming language semantics (9, 10). It stresses the tree structure of programs, clarifies the way a construct is obtained by composition of simpler constructs, points out the various kinds of syntactic structures the language provides. All the details concerning lexical aspects or requirements for efficient parsing (e.g. non-ambiguity, determinism) are completely neglected.

Our goal is to define an internal representation common to every language the environment will support. The specific internal representation will be derived according to the abstract syntax of the specific language. Hence, we have defined a meta-language to express abstract syntax which provides a set of basic constructs which are used to built and manipulate trees.

We will now informally describe the way we use to define the abstract syntax of the language. A definition is a set of syntactic domain definitions of the following form:

(F1) $op(sel_1:DTYPE_1,..,sel_n:DTYPE_n) \longrightarrow DTYPE$

which constructs elements of the syntactic domain DTYPE (e.g. IDEntifier, TYPE, STATement,..) by applying op to elements of the syntactic domains $DTYPE_1,...,DTYPE_n$, identifiable by selectors $sel_1,...,sel_n$ (see the example below).

In the scheme (F1), domain $DTYPE_i$ may have also the following forms:

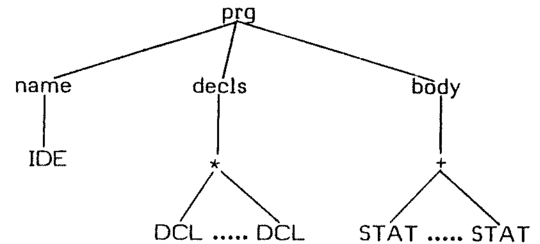
i) DTYPE *, which stands for a (possibly empty) list of elements of the domain DTYPE;

ii) DTYPE+, which stands for a non-empty list of elements of the domain DTYPE.

Example.

$prg(name:IDE, dcls:DCL , body:STAT^+) \longrightarrow PROGRAM$

defines a (simple) program construct consisting of a name, a list of declarations and a non-empty list of statements. A tree of domain PROGRAM is the following:



where name, dcls, body are the selectors of the sons of type IDE, DCL and $STAT^+$, respectively.

The basic nodes of a tree are:

- the operator node, with a fixed number of sons;
- the (possibly empty) list node * and the non-empty list node +, with a varying number of sons;
- leaf node, without sons;
- the domain node, which has no sons but can be expanded according to the definition of the domain (e.g. the leaves of the figure above).

The basic operation to visit and modify trees are:

- root, which gives the root of the internal representation;
- father, which gives the node having the argument as a son;
- son_of_name, which, taken a node and a selector name, gives the subtree labelled by the selector name;
- i-th_son, which, taken a list node, gives its i-th subtree;
- brother, which allows to visit the elements of a list node;

- change, which, given two trees, substitutes the first (possibly empty) for the second.

The tree structure of internal representation must be augmented by allowing property lists to be appended to the nodes. Such properties can be defined and used by the programmer and by the tools of the environment. Each tool will define its own properties (for instance, the editor will operate on properties like "syntactic domain" and the type checker on properties like "type"). The abstract syntax manipulation module provides primitives for properties insertion, deletion and modification.

4. PROGRAM EDITING AND MANIPULATION

The abstract representation of a program is built starting from its text (concrete representation) by means of the syntax directed editor. Unlike ordinary editors, the syntax directed editor does not operate on strings of characters, rather it handles trees. The syntax directed editor is composed by a routine for handling abstract representations, lexical and syntactical analyzers, and a routine for building abstract representations. Moreover, a program must be displayed according to its concrete syntax, since the user should operate on it (e.g. inspect, modify, analyze it) always through the same (and more familiar) representation.

The syntax directed editor must support incrementality, since should be possible to incrementally write (or modify) a program. The incremental definition of a program could be achieved both by a "generative" and by an "analytic" definition mode. The generative mode consists in driving program definition by displaying the menu of the language constructs usable in that specific context. For example, if the user wants to write a while-statement, the syntax directed editor will display the following

```
while COND do
      STAT
      od .
```

The user is then allowed to select and expand one of the elements of the two syntactic domains COND and STAT, according to their definitions.

The major advantages of the generative mode are that programs are edited in terms of the programming language constructs, and that users cannot write syntactically incorrect programs, thus avoiding the need of a parsing phase. On the other hand, this approach may be rather tedious, e.g. when writing an arithmetic expression.

The analytic definition mode is the standard way of writing programs, enriched with possibly incremental parsing and by allowing the user to let unexpanded some of its parts (with the constraint that the unexpanded part must be a legal syntatic element). The program will be completed by successive modifications of its abstract representation resulting from the complete expansion of all the syntactic constructs.

Our syntax directed editor supports a mixed use of the two modes, in order to achieve the advantages of both the generative and the analytic modes.

To generate a syntax directed editor from the definition of a language, a parser generator and a constructor of abstract representations is needed. The technique adopted in the design of the parser generator is SLR(1), extended to cope with incrementality. The parser must be able to analyze not only a whole program, but an element of any syntactic domain, typically when in generative mode one substitutes for an unexpanded element its full definition. In other words such a situation arises when a part of a program (belonging to a specific syntactic domain) is to be replaced by another part which must belong to the same domain. The adopted solution consists in splitting and slightly extending the parsing table of the language in several parsing (sub)tables, each corresponding to a syntactic domain. The parsing of a part of a program is done by using the (sub)table(s) corresponding to its syntactic domain. As a particular case, a program will be parsed by using all the (sub)tables.

To generate the constructor of abstract representations, a meta-tool has been designed which takes as input a mapping from concrete to abstract syntax. Such a mapping is needed since in general the two syntaxes are quite different. In fact, the concrete syntax contains more productions than the abstract one, both to express syntactic details (such as presence of begin-end) and to define semantic aspects (such as the operator precedence relationships). The way this mapping is obtained is similar to that of (11), and will not be given here, due to space limitations. The constructor of abstract representations is a meta-tool which, given a mapping from the concrete to the abstract syntax, builds the internal representation.

The integrated programming environment user must interact with his program always through its concrete representation. Thus, there must be a displayer which, given an abstract representation of a program, displays it in a concrete form. Let us note that the displayer is used also in the generative program definition mode.

The text of a program is displayed according to a form normalized by the abstract syntax. Moreover it is prettyprinted to better show its structure. All the standard functionalities of a prettyprinter (see for example (12)), e.g. displaying only few levels of a programs, are supported.

The program displayer is generated by another meta-tool which takes as input a mapping from the abstract syntax to the displaying format. Let us finally remark that the displayer results to be a non-standard interpreter for the language.

5. PROGRAM EXECUTION TOOLS

The essential feature of an execution tool in a program development environment is that of allowing interaction during execution in terms of the source language. For this reason, the main execution tool of our programming environment is an interpreter. In this section we describe a methodology for deriving interpreters, and other execution tools from the formal definition of the source language semantics. In particular we will refer to the denotational style of definition, which is now widespread for sequential languages. For instance, it has recently used to give the semantics of the sequential part of ADA (14, 15).

The denotational definition of the semantics gives to a program a meaning in some abstract domain which is defined using mathematical concepts like functions or sets. For instance, the meaning of a program might be a function from an input to an output domain.

The main characteristic of denotational semantics is that the meaning of a program is given in terms of the meaning of its syntactic components. More precisely, the denotational semantics gives a meaning to every (abstract) syntax construct of the language, and this meaning is defined in terms of the meanings of the components of that construct (compositional property). To give the meaning of every syntactic construct, some auxiliary domains must be introduced. For instance, the semantics of imperative languages makes use of domains like environment, state or continuations (to descibe jumps).

Thus the main features of denotational semantics are that of being necessarily structured with respect to the abstract syntax of the language, and of expressing the meaning of the constructs in terms of abstract domains, which model important semantic concepts, without any constraint from real machine.

The denotational semantics of the source language may be expressed in an algorithmic formalism, i.e. another programming language, called meta-language (for instance the SIS system by Mosses (11) provides a meta-language, called DSL, to define the denotational semantics).

Once the semantics of a language L has been written in a meta-language M, it is possible to execute programs of L by means of the executor of M. This execution may be performed in steps. Let us assume, for instance, that the semantics LSem(P) of a program P is a function from an input to an output domain. Then we can first apply LSem to a given program P obtaining a function from input to output. Then this function LSem(P) may be applied to input data to get the corresponding output. Note that this is essentially a "compilative" approach, where "compiling" means translating from L into the meta-language. In fact, the first step "compiles" P into an M program, which in the second step is evaluated on input data.

An "interpretive" approach requires a different style of giving the denotational semantics, where only "first order" functions are allowed. That is, for instance, the semantics of a program will be defined as a function which takes two arguments, a program and an input, and returns an output. The main difference between the two approaches lies in the semantics of procedures: in the "higher order" approach the semantics of a procedure is a function, whereas in the "first order" approach it is usually a so called "closure", i.e. a structure consisting of the procedure itself and of the environment in which its body has to be evaluated.

However the "first order" semantics is not quite denotational, since the meaning of a procedure is not given in a semantic domain, and the semantics of a procedure call is not compositional. Thus a correct methodology would be to consider the "higer order" semantics as the true semantics of the language, and to

derive the "first order" semantics from it through an equivalence preserving transformation (10, 13).
Besides the interpreter, the programming environment will provide a compiler to run already tested and debugged program units with a gain in efficency. By compiler we mean, of course, the granule of traditional compiler performing code generation and optimization. This tool must be integrated with the interpreter in order to allow execution of partially compiled programs. The required integration may be achieved by constructing the compiler with the same technique described above, i.e. through a sequence of transformations starting from denotational semantics (13, 16).
Finally, if the source language possesses the concept of module as a separate program unit, then the programming environment will provide another tool, the configuration interpreter, for extracting modules from a program library and connecting them together to form program ready to be run. Thus, this tool will provide the functionalities of traditional linkers and loaders, but tuned with the specific source language (by performing, for instance, type checking).

5.1 STATIC ANALYSIS TOOLS

Static semantics is the part of semantics which describes all the features of a language which can be checked statically such as, for instance, scope rules or type consistency. It may be also defined using a denotational technique, if the meaning of syntactic constructs is given in different (non standard) domains. For instance, the meaning of a program might simply be either "correct" or "wrong", the domain of values might become the domain of types, and the store might be eliminated. As in the case of standard semantics, if we define the static semantics in the meta-language, we have an interpreter (static interpreter), which can be used to check the static aspects of a program. As pointed out before, this interpreter operates on non standard domains, and usually has a simple control structure than the standard one (cycles are executed only once).
Often the semantics of a language is given, by definition, in two parts, the static part and the dynamic part (see for instance the semantics of ADA (14, 15)). The static semantics establishes which syntactically correct programs are well formed, and the dynamic semantics is defined only for well formed programs.
We propose instead to start with a unique standard denotational semantics of a language, dealing with both static and dynamic aspects, and then derive from it the static semantics. If the meta-language possesses an abstract data type constructor, the standard semantics will be given by referring to domains defined as abstract data types; then the static semantics, and the associated interpreter, will be obtained by modifying the specification of those data types. Note that with this approach the structure of the standard and static interpreters is always the same, and that they differ only in the specification of semantic domains. Furthermore we point out that we might derive more than one static interpreter, for dealing separately, for instance, with scope rules and with type checking, if the structure of the language allows it.

5.2 DEBUGGING TOOLS

The process of interactive error detection and correction may be divided into three phases, which usually require different tools:

- detection and correction of syntax errors;
- detection and correction of static semantics errors (types, scope rules,...);
- detection and correction of dynamic semantics (run time) errors.

In traditional programming environments, either compiler or interpreter based, the first two phases are performed during translation from the text of the program into the internal form. Thus the interaction is necessarily limited, consisting simply in correcting the program text according to the error messages and in retranslating it.
Traditional debuggers deal with the third phase, allowing to interact with the program during execution. The main feature of a debugger is that the programmer must always be able to reason at the level of the source language in terms of semantic concepts, without worrying about the underlying implementation. Most currently available debuggers are compiler based, and the interactions with the source program are rather limited, depending on the informations about the source program which are passed to the compiled code. Usually they allow to inspect, and in some cases to alter, the value of a variable and to refer to text lines in the

source program in order to insert or remove break and trace points. More sophisticated interactions are possible if an interpreter is used instead of a compiler, since the interpreter may refer directly to (an internal representation of) the source program. A further improvement is achieved by using a symbolic interpreter during the debugging phase (17).

In the programming environment described in this paper, the first phase, dealing with syntactic errors, will be performed interactively through the syntax directed editor.

The other two phases, dealing with static and dynamic errors, may be described, in general terms, in the same way. In fact, as we pointed out in the previous subsection, static checking of a program is performed by the execution of a non standard (static) interpreter, which may stop either at some point of the program having detected a static error, or at the end of the program if it is correct. Similary, the run time interpreter will stop whenever a run time error or a break point will be encountered. In any case, when an interpreter stops, another tool is entered, the (static or dynamic) debugger, which manages the programmer's interaction. Within the debugger the programmer is able, first of all, to inspect the state of the interpreter at the point where it stopped. This means examining the program through the editor, and accessing the semantic domains through suitably defined abstract operations. Furthermore, the programmer can, possibly in a limited way, modify the program or the state of a semantic domain, and restart the interpreter.

More specifically, the dynamic debugger allows to examine and alter the state of the program by on line editing and executing source language statements provided by the user. Thus, the value of variables can be known through a "write" statement, and it can be altered through an assignement statement.

Furthermore, the dynamic debugger allows to add, or remove, properties (breaks, trace points, frequency counts,...) to the internal representation of a program in order to monitor its execution.

6. CONCLUDING REMARKS

The above described programming tools, derived from a formal definition of the programming language, must be integrated with the language-independent tools, so as to define a complete minimal environment. Integrability strongly depends upon uniformity of tool interfaces and upon the data base which contains, structures and manages the informations about both tools and program entities, providing the input to all the tools and storing their outputs, e.g. program manipulations, analysis results, etc.

The minimal toolset can be extended with new tools defined in the programming language supported by the environment, possibly through invocation of the basic tools. This can be achieved only if the environment defines a model for transmitting parameters and sharing data between modules in the programming language and modules of the implementation (meta-) language.

A final relevant integration mechanism is the command language, which provides the basic user interactive interface, and can be the same as the language supported by the environment. Each interactively supplied command is entered through the editor, checked by the static analysis tool, and executed by the interpreter.

The programming environment briefly described in this paper is currently under development and will be hosted on a DEC VAX-11/780. The first prototype will be tested on a subset of ADA.

REFERENCES

1. Teitelman W. - INTERLISP Reference Manual - XEROX Palo Alto Research Center, Technical Report (1978).
2. Sandewall E. - Programming in the Interactive Environment: the LISP Experience - ACM Comp. Surveys 10 (1978) 35-71.
3. Ritchie D.M. and Thompson K. - The UNIX Time-sharing System - Comm. ACM 17 (1974) 365-375.
4. Ivie E.L. - The Programmer's Workbench: A Machine for Software Development - Comm. ACM 20 (1977) 746-753.
5. Cheatham T.E., Townley J.A. and Holloway G.H. -A System for Program Refinement - Proc. 4th Int'l Conf. on Soft. Eng. (1979) 63-72.
6. Cheatham T.E. - Program Development Systems - Center for Research in Computing Technology, Harvard University, Technical Report (1979).
7. U.S. Department of Defense - Requirements for ADA Language Integrated Computer Environments -

STONEMAN (1980).

8. Hunke H. (ed.) - Software Engineering Environments, North-Holland Pub., Amsterdam (1981).
9. McCarthy J. - Towards a Mathematical Science of Computation - Proc. IFIP Congress 1962, C.M. Popplewell (ed.), North-Holland Pub., Amsterdam (1963) 21-28.
10. Reynolds J.C. - Definitional Interpreters for Higher-Order Programming Languages - Proc. ACM Nat. Conf. (1972) 717-740.
11. Mosses P. - SIS: Semantics Implementation System, Reference Manual and User Guide -Aarhus Univ., Computer Department, Internal Report DAIMI MD-30.
12. Oppen D.C. - Prettyprinting - ACM Trans. on Prog. Lang. and Systems 2,465-483.
13. Bjorner D. - Programming Languages: Formal Development of Interpreters and Compilers - Int. Comp. Symposium 1977, Morlet E. and Ribbens D. (eds.), North-Holland Pub., 1-21.
14. Bjorner D. and Oest O.N. (eds.) - Towards a Formal Description of ADA - Lect. Notes in Comp. Science, 98, Springer Verlag (1980).
15. Formal Definition of the ADA Programming Language - November 1980 edition, Honeywell Inc., Cii Honeywell Bull, INRIA (1980).
16. Gaudel M.C. - Compiler Generation from Formal Definition of Programming Languages: A Survey - Int. Coll. on Formalization of Programming Concepts, Lect. Notes in Comp. Science, 107, Springer Verlag (1981) 97-114.
17. Asirelli P., Degano P., Levi G., Martelli A., Montanari U., Pacini G., Sirovich F. and Turini F. -A Flexible Environment for Program Development Based on a Symbolic Interpreter -Proc. 4th Int'l Conf. on Soft. Eng., (1979) 251-263.

Roberto Barbuti, degree in Computer Science from the University of Pisa in 1977, is presently interested in programming languages and software engineering. Until 1981 he has been a research fellow at Istituto di Elaborazione dell'Informazione C.N.R. in Pisa. Now he is assistent professor at University of Pisa, Computer Science Department.

Marco Bellia, degree in Computer Science from the University of Pisa in 1975, has been research fellow at Istituto di Elaborazione dell'Informazione C.N.R. in Pisa until 1981. His present position is at Computer Science Department, University of Pisa, as assistent professor. His fields of interest include artificial intelligence and software engineering.

Enrico Dameri, degree in Computer Science from the University of Pisa in 1979, is a member of the research staff of Systems & Management in Pisa. His main professional interest is on software development tools and programming languages.

Pierpaolo Degano, degree in Computer Science from the University of Pisa in 1973, has been since then at the Computer Science Department of University of Pisa first as a research fellow and then as an assistent professor. His work centers on programming languages and software engineering.

Giorgio Levi, degree in Electrical Engineering from the University of Padova in 1966, has been a researcher at Istituto di Elaborazione dell'Informazione C.N.R. in Pisa and now is a full professor at the Computer Science Department of University of Pisa. He is presently interested in programming languages and software engineering.

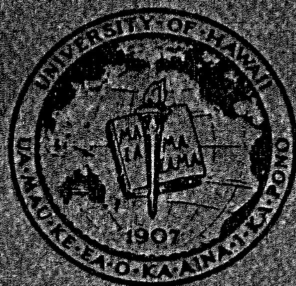
Alberto Martelli, degree in Electrical Engineering from the Politecnico of Milano in 1967, has been a researcher of Istituto di Elaborazione dell'Informazione C.N.R. in Pisa and now is full professor of Computer Science at University of Torino. His work is concentrated on programming languages and software engineering.

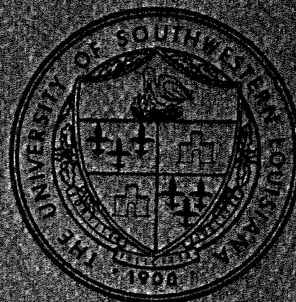
Carlandrea Simonelli, degree in Computer Science from the University of Pisa in 1978, is a member of the research staff of Systems & Management in Pisa. His professional activity is mainly concerned with software engineering and programming languages.

PROCEEDINGS of the

FIFTEENTH HAWAII INTERNATIONAL CONFERENCE ON SYSTEM SCIENCES

1982





VOLUME I

SOFTWARE, HARDWARE, DECISION SUPPORT SYSTEMS, SPECIAL TOPICS

EDITED BY

WILLIAM RIDDLE
Cray Laboratories

KEN THURBER
Sperry Univac

Peter Keen
Massachusetts Institute of Technology

RALPH H. SPRAGUE, JR.
University of Hawaii