CENTAUR: the system

P. Borras, D. Clément

SEMA, Paris and SEMA, Sophia–Antipolis

Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, V. Pascual

INRIA, Sophia–Antipolis and INRIA, Rocquencourt†

FRANCE

This paper describes the organization of the CENTAUR system and its main components. The system is a generic interactive environment. When given the formal specification of a particular programming language – including syntax and semantics – it produces a language specific environment. This resulting environment includes a structure editor, an interpreter/debugger and other tools, all of which have graphic interfaces. CENTAUR is made of three parts: a database component, that provides standardized representation and access to formal objects and their persistent storage; a logical engine that is used to execute formal specifications; an object-oriented man-machine interface that gives easy access to the system's functions. CENTAUR is essentially written in Lisp (Le_Lisp). The logical engine is Prolog (Mu-Prolog). The man-machine interface is built on top of the virtual graphics facility of Le_Lisp, itself primarily implemented on top of X-Windows.

1. Introduction

The CENTAUR system is a generic interactive environment. When provided with the description of a particular programming language – including its syntax and semantics – it produces a language specific environment. The resulting environment comprises a structure editor, an interpreter/debugger and other tools, all of which have graphic interfaces. We believe that designing new languages – whether large or small, specialized or general – will be a very common activity in the future. Hence it makes sense to build such a system.

In the design of CENTAUR, we are faced with at least three challenges:

i) Is it possible to use as the description of a programming language a *formal specification* in the sense used by researchers in programming language semantics?

ii) Is it possible to obtain on that basis an efficient system, both in terms of speed and in terms of memory space?

iii) Is it possible to create a man-machine interface that is comparable in convenience to what can currently be found in dedicated commercial systems, say on a MacIntosh?

This paper describes the CENTAUR system. The conclusion presents preliminary answers to the questions above.

1.1. *Outline of the CENTAUR architecture*

The architecture of the CENTAUR environment generator is based on a decomposition into three classes of functions:

- a *kernel* for the representation and manipulation of structured objects,
- a *specification level* to support both the syntactic and the semantic aspects of languages
- a *user interface* that takes care of all interactive communication between the system and its users.

The kernel plays a central part in this architecture because it is intended to be used, directly or indirectly, by all other components of the system. Every

† This research is partially supported under ESPRIT, Project 348.

structured object is represented within the system as an abstract syntax tree that is manipulated with primitives of the kernel.

1.2. *The kernel*

The purpose of CENTAUR's kernel is to support symbolic manipulation of structured documents. By symbolic manipulation we mean destructive manipulation, typically used while editing documents, as well as evaluation, more common in semantic processing. The kernel of the CENTAUR system is made of two specialized components:

- an abstract machine that handles syntactic aspects, called the Virtual Tree Processor or VTP [19],
- a logical machine that handles semantic aspects, i.e. evaluations of all kinds. At this stage the logical machine is a Prolog interpreter.

These two components are intended to be used as co-routines. They are connected through an interface that contains two classes of primitives:

- *control* primitives to call Prolog from within the Virtual Tree Processor and to call the Virtual Tree Processor from within Prolog,
- *transformation* primitives to perform coercions between Virtual Tree Processor trees and Prolog terms.

1.3. *The specification level*

To generate an interactive programming environment for a given language one must provide specifications for its concrete syntax, abstract syntax, and semantics. These specifications are themselves written in appropriate formalisms.

1.3.1. *Specifying syntax*

The specification of syntax contains the components that are necessary for deriving a structure-oriented editor. After compilation of this specification, one obtains a scanner, a multi-entry parser, a pretty-printer, and abstract syntax tables. The scanner and parser are used to transform a textual representation into a structured representation. The unparser, or pretty-printer, is used to transform a structured representation to a textual form. The abstract syntax tables are used to check the validity of *editing* operations.

The specifications of concrete and abstract syntax, together with their relationship, are presently written in METAL [18], a formalism developed for the MENTOR system [8] [9]. A METAL specification is a collection of grammar rules, with annotations that specify what abstract syntax trees should be synthesized. Pretty-printing of abstract trees is defined in the PPML formalism [22]. A PPML specification is a collection of unparsing rules associated with abstract syntax patterns.

In the future we hope to merge all syntactic aspects of language definition within a single formalism [13].

1.3.2. *Specifying semantics*

To derive systematically a language-specific interactive programming environment it is necessary to go beyond syntax and introduce semantic specifications. For example:

- from a specification of static semantics, one wants to derive a type checker and context-dependent interactions,
- from a specification of dynamic semantics, one wants to derive an interpreter and a symbolic debugger.

Several formalims could be used to express the semantics of programming languages. We have experimented with a formalism based on Natural Deduction that we call Natural Semantics [17]. A semantic specification is written in TYPOL [6], the computer version of Natural Semantics. Other experiments have been carried out using ASF [1], an algebraic specification formalism. Both formalisms are type-checked and compiled into executable form. At the present time, specifications are compiled into Prolog.

1.4. *Principles of the man-machine interface*

The functions of CENTAUR are accessed in two non-exclusive ways: through direct function calls and through the man-machine interface. The design of the interface presupposes that the user is interacting with CENTAUR *via* a modern workstation, using a keyboard, a mouse, and a high resolution screen. Most of the time, the user makes his intentions known using the mouse. When a mouse button is depressed, the location of the mouse determines what graphic object the user is talking to and a finite automaton takes control of the mouse until the button is released.

A variety of graphic objects are used in the man-machine interface: menus offering various options, buttons to control execution, pseudo-terminals, etc. More specific to CENTAUR are the *views* where abstract syntax trees are displayed[1]. Structure editing as well as text editing are available in a CENTAUR view. Within a view, a subexpression is emphasized, appearing in boldface. This subexpression is the subject of most commands. The simplest gesture with the mouse is to point at an expression to emphasize it.

The user is normally manipulating several formalisms at the same time. Each formalism has its own structure and its own specific environment. In fact, in a single CENTAUR view more than one formalism can be present: typically annotations are interspersed within the text of a program. The operations that are permitted and the menus that pop up all depend on the formalism of the currently emphasized expression.

Furthermore, the various objects that appear on the screen are usually not independent. For example, one view may display a source program, and another view the result of translating this program into an abstract machine code; or two views may look at the same abstract syntax tree with different prettyprinters. A network of such dependencies is maintained automatically to assist the user in keeping a consistent screen. This *logical control* of the interface relies for the time being on the Prolog component of CENTAUR.

1.5. *Implementation Issues*

CENTAUR is implemented in Lisp and Prolog. It runs on standard modern workstations. It is also possible to run CENTAUR on a centralized timeshared computer and retain a high quality graphics interface because the man-machine interface is ultimately built on top of X-Windows [10].

1.5.1. *Using Lisp*

The design and implementation of CENTAUR are based on our experience with the design, implementation, and actual use of a first generation system called MENTOR [8] that was implemented in Pascal. The software technology used for environment generators is clearly based on symbolic computation. Traditional languages such as Fortran, Pascal, or Ada are not well suited for implementing such applications. A functional programming language like ML [20] is far more adapted but an efficient implementation of ML was unfortunately not available at the start of the project.

[1] Trees are displayed as text; graphic representations have not yet been considered.

After some experimentation with C it was decided to use Lisp as a flexible *low-level language* for the implementation for the VTP. The Lisp dialect actually chosen is Le_Lisp [3]. This dialect of Lisp was selected for its availability on a large variety of machines and the ease with which it is ported on new hardware.

1.5.2. *Using Prolog*

Early experiments had shown that Prolog was a convenient target language for the compilation of semantic specifications [6]. These experiments were confirmed repeatedly during the project development. In the context of an interactive programming environment mostly implemented in Lisp, using Prolog as an inference system implies that Prolog may be used and accessed from within Lisp, and in turn, that Lisp may be called from within Prolog. Then Prolog is available as a coroutine to Lisp.

An immediate advantage of this strategy is the possibility of experimenting with different Prolog engines. This has already proven useful in switching from a first Prolog interpreter, C-Prolog of Edinburgh University [23], to a more powerful Prolog system, the Mu-Prolog of Melbourne University [21]. In the near future we hope to change over smoothly to *compiled* Prolog.

This approach, however, requires coercions between different internal representations. For example a Virtual Tree Processor abstract tree must be transformed into a Prolog term. Using directly the primitives for constructing Prolog terms (rather than the Prolog reader) the cost of this coercion is roughly that of a copy. One may wonder whether a more integrated approach, where every object has a unique representation that is used by all processors, would not be superior. But in fact, the choice of a proper representation for structured objects depends on what manipulations have to be performed on them and we believe that the need for multiple representations is inherent to symbolic processing.

From Lisp one uses Prolog in a straightforward manner. Goals are sent to Prolog via a buffer, the Prolog mailbox, with a call to the *mailtoprolog* procedure. Then to solve this goal, one calls a *resolve* that:

- fetches a goal in the Prolog mailbox
- silently resolves the goal
- then returns to its caller. Later calls will find Prolog in the exact state where it was left.

Hence Prolog and Lisp behave as two co-routines. Evaluable predicates may be written in Lisp. This is in particular how Prolog is given access to CENTAUR's man-machine interface.

1.5.3. *Using a modern workstation*

The development and effective use of CENTAUR demand, in its current form, a computer system with a large memory (8 megabytes), reasonable computing power, and a graphic interface. Modern workstations provide the appropriate hardware. Code written in Le_Lisp is portable but graphic interfaces are far from standardized. To isolate the programmer from system variations, Le_Lisp includes a virtual multiwindow graphic terminal. Several implementations of this virtual terminal are available, but we use primarily the implementation on top of X-Windows, Version 11 [10]. Using X-Windows should make CENTAUR available on most modern workstations.

2. The database

The first component of the kernel is the Virtual Tree Processor or VTP. The VTP defines and implements efficiently a protocol supporting creation, manipulation, and persistent storage of abstract syntax trees.

2.1. *Structure of the VTP*

As a protocol, the VTP is a collection of abstract data types. Certain functions that are not logically necessary (for example iterators) are included to obtain a far more efficient implementation. The protocol is *open* in that it provides a method for creating new types of objects to enrich the VTP.

The specification of the various data types yields a natural modularity to the VTP. Another level of modularity results from following an object-oriented approach. Many objects provided by the VTP carry with them their own dynamic types. These types are organized hierarchically. The main advantage of this architecture is to allow *extending the system* without any alteration of existing code, as long as the type hierarchy is extended.

2.2. *Main concepts of the VTP*

The abstract concepts defined and implemented in the VTP are motivated by a number of earlier experiments on syntax-directed document manipulation, in particular with our MENTOR system.

The VTP is organized around very few concepts. A *formalism* embodies the notion of an abstract syntax. A *tree* belongs to one formalism, but it may have subtrees in other formalisms, for example using the *gate* mechanism. *Contexts* materialize the idea of one or several locations in a tree. *Annotations* may be hooked to abstract syntax trees to retain additional information without modifying their structure.

2.2.1. *Constructing a formalism*

An abstract syntax is represented in the VTP by an object in the class *formalism*. A formalism is made of objects in the classes *operator* and *phylum*, where a phylum is a set of operators. Primitives are available to create a new formalism and populate it with new operators and phyla. When creating an operator, its arity and the phyla of its operands must be provided. Operators come in three kinds: operators with a fixed arity, list operators where the list may not be empty, and list operators where the list may be empty. It is possible to iterate over all operators and over all phyla in a formalism, and over all operators in a phylum.

When all phyla and and all operators of a formalism have been given, the formalism is declared complete. Then a number of global calculations are performed that result in faster tree traversals and more compact tree representations in persistent storage. The formalism so constructed has a persistent representation: it may be saved in a file and read in at a later time.

The natural way to create a new formalism is to use a specification language. A compiler for the specification language will then use the primitives provided by the VTP to create the formalism. For example, the language METAL is currently used to specify simultaneously the concrete and abstract syntax of a language. It is possible to experiment with other syntax specification languages (e.g. SDF [13]) without changes to the rest of the system because all other components of CENTAUR are only interfaced with VTP objects.

2.2.2. *Trees*

From an operator and an adequate number of subtrees one can create a new tree. The tree is then said to belong to the formalism of its top operator. The VTP provides primitives to navigate in a tree (such as *down*, *left*, *right*, *up*, etc.) and to modify the tree: replacing a subtree by another one, inserting and deleting elements in lists. All modifications *fail* if they are incorrect with respect to the abstract syntax. Syntax checking is strictly local and very fast. It is possible to iterate efficiently over all subtrees of a tree. Finally, trees have a representation in persistent storage.

The VTP defines a *gate* mechanism to associate values to the leaves of an abstract syntax tree. For example, identifiers or integer literals are zero-ary operators: their value is obtained by traversing a gate. In this way, the value associated with a leaf may be *any* VTP object, in particular an abstract syntax tree in a different formalism.

Typically, trees will be constructed by a parser and modified by interactive editing. The VTP defines the protocol that any parser must use to create trees, but it is not committed to any particular kind of parser. Likewise it provides the primitives to build an interactive structure editor but it does not enforce a specific mode of interaction.

2.2.3. *Contexts*

An essential aspect of the VTP is that it supports *imperative* manipulation of formal documents. Thus the abstract representation of a document is not a recursive structure that is built and processed solely in a purely applicative way. Rather it is a kind of database to explore and modify. As a result it is frequently necessary to keep references to places of interest in the manipulated structure.

We call *contexts* values referring to locations rather than sub-objects. A context value is thus a place within a mutable object, where a component of that mutable object may be stored. Several classes of contexts are defined in the VTP. The classes *subtree*, *annotation*, and *gate* correspond to the mechanisms used in constructing trees. The class *sublist* designates several consecutive locations in a list.

In the current implementation of the VTP, contexts do not have a persistent representation. A more abstract notion of context that will denote an arbitrary set of disjoint locations will be added to the VTP in the future. It will have a persistent representation so that, for example, the state of an editing session may be saved.

2.2.4. *Annotations*

Annotations are a more formal and structured view of the old concept of *property lists*, originally introduced in Lisp. The idea is to attach to objects a list of *independently managed* tagged values, where each tag or *property* characterizes the role of the value.

In the VTP, the role of an annotation must be declared by first defining a *decor*. For example, one can define a decor called "pragma" that can be used to attach string annotations to nodes of abstract syntax trees representing Pascal programs. So the definition of a decor includes its name (e.g. pragma), the type of the annotation itself (e.g. string) and the type of objects that may be decorated by annotations in this decor (e.g. Pascal programs).

Annotations are used to handle *comments* and formal *assertions*. But most applications need a general mechanism to decorate abstract representations. Using annotations for document organization and manipulation is discussed at length in [8]. One important application is to use annotations to store attributes, in the precise sense of attribute grammars [24].

In the VTP, the use of decors and annotations has been extended. It is possible to define decors and annotations for other classes of objects than trees. For a given class of objects, new decors may be dynamically created, and corresponding annotations may be attached to the objects in the class. So annotations are essentially a way to define dynamically and independently new fields or properties. Hence annotations are used as a modularity device to *extend* the functionality of VTP classes without interacting with their already existing semantics.

2.3. *Discussion*

The VTP is a stable and trustworthy component of CENTAUR. Further work on the VTP will take place in several directions: improving error handling, adding new functionalities, improving the overall efficiency.

New functionalities are added sparingly. As indicated above, a more abstract notion of context will be added to the VTP. To keep track of the various trees constructed during a session, a notion of VTP *variable* distinct from Lisp variables is being developed as well.

Efficiency is improved on a demand basis. The performance analysis of the VTP is generally difficult, because various processors use the VTP in fairly different ways. Inefficiencies often point to an inadequate choice of the primitive functions for some class. This has motivated adding several iterators to the VTP. Translating between computational and persistent storage must made faster and should produce even more compact representations.

3. **Using the database**

This section presents very briefly three components of CENTAUR and how they use the VTP. The first is the METAL compiler, which creates a new formalism and a parser for it. The second is the PPML compiler, which generates a pretty-printer from a symbolic description. These two combine to make a third component, a simple structure-oriented editor.

3.1. *The METAL compiler*

METAL[18] is a language used in specifying new formalisms. The METAL specification consists of three parts: 1) the abstract syntax, 2) the concrete syntax, 3) a set of *tree building* functions (one associated to each production rule of the concrete syntax).

The compilation of a METAL specification creates: 1) a VTP formalism, 2) a concrete syntax parser, and 3) a tree generator which uses VTP primitives to construct abstract syntax trees. The parser is the result of compiling the concrete syntax into a format on which YACC[16] is called. The link between the parser and the tree generator is a list of action numbers generated while parsing; this tells how to construct the abstract syntax tree.

The METAL compiler has been used on large languages (Pascal, Ada, C). However, it is somewhat difficult to use. This is largely due to the fact that Metal is verbose and the parser generator is YACC. Another formalism, called Syntax Definition Formalism, has been designed and implemented by our partners (see [13]). In SDF concrete and abstract syntax are defined simultaneously and there is no restriction on the context-free grammar used. Furthermore, the scanners and parsers are built incrementally [14][15]. As a consequence, specifying a new language will be far simpler.

3.2. *The Ppml compiler*

A special purpose formalism, PPML [22], is used to associate concrete layouts to abstract structures. In a PPML specification one describes the desired layout for an abstract syntax tree including how much is visible depending on the depth of the tree.

3.2.1. *Pretty-printing rules*

Roughly speaking, a PPML specification is an ordered list of pretty-printing *rules* of the form

$$pattern \longrightarrow format$$

where the pattern is an abstract syntax tree containing variables and the format describes the concrete layout of the pattern using a box language and referring to the variables. A PPML specification necessarily imports a language's abstract syntax.

For pretty-printing a tree, a standard first order pattern-matching mechanism is used to match the top of the tree. Variables in the pattern are associated to subtrees. Within the format, each occurrence of a variable results in a recursive call of the pretty printer for its subtree. For tree leaves, the appropriate pretty-printing function is called depending on the type of the leave's value.

In the box language, a box can be either "atomic", which represents the lexical elements of a language, or "compound", which combines several boxes. The *type* of the compound box and its *separators* determine the layout. The types of compound boxes include: *h* for horizontal, *v* for vertical, *hov* which gives a horizontal or, if there is not enough room, a vertical box, *hv* for horizontal with vertical folding (e.g., list of declarations). Separators are useful to express indentation and spacing. In the example below, *decls* and *stms* are variables:

`program(`*decls, stms*`)` ⟶
    `[<v> begin <v tab,0> [<v>` *decls stms*`] end]`

3.2.2. *The generated code*

PPML programs are type-checked and then translated to Lisp code. The type-checker verifies that no rule is useless (i.e. hidden by a more general rule) and that the specification is consistent with the language definition. When no rule is given for a construct, the system uses by default a generic pretty-printer. As specific rules are added to the PPML specification they supersede the default rules used by the generic pretty-printer.

The generated code contains calls to a formatting machine so that it is easy to print "mixed" structures. This is very useful because annotations are almost certainly in a different formalism. The formatting machine generates a data structure that is useful in quickly selecting and enhancing subexpressions.

3.2.3. *Discussion*

The PPML compiler is incremental: when a rule is modified, only this rule and the rules concerning the same operator are recompiled. The Lisp code generated by the PPML compiler is itself compiled, resulting in an efficient display of tree structures. It is also quite useful to be able to have several pretty-printers defined for the same language. Good results can be obtained in a short time and PPML specifications have been written for large languages (Pascal and C).

However, several improvements would lead to a more robust environment; these include tabular layout and a mechanism for defining new combinators (types of compound boxes). Also the layout algorithm could be more sophisticated by using additional look-ahead.

3.3. *The prototype editor*

To observe trees, one uses pretty-printers to create images that are displayed. A VTP context memorizes the *current expression* of interest; this is the primary argument of editing commands and it is usually printed in boldface. All simple editing commands are combinations of VTP primitives; editing operations are valid only if they respect abstract syntax constraints. A structured clipboard is used for cut-and-paste style edition. For text editing, the input is parsed and the corresponding tree is constructed using VTP primitives. If parsing fails then the input remains available for further editing. (Currently, text and structured editing occur in separate windows.) Language-dependent commands can either be added by Lisp programs using VTP primitives or derived from semantic specifications.

The VTP contains adequate primitives to construct a structure-oriented editor. For example, adding a menu-driven mode of input, in the style of the Cornell Program Synthesizer [25], is the matter of one or two pages of code. When additional objects are needed, the VTP may be extended easily. The real difficulty is in designing a smooth, natural structure-oriented editor, where structural and textual manipulation are blended effortlessly.

4. The logical machinery

A method for specifying the semantic aspects of languages is included in CENTAUR, so that the system is not restricted to manipulations that are based solely on syntax. This section describes the TYPOL language and its compiler, and it shows how it is possible to take advantage of semantic descriptions in an interactive environment. The use of algebraic specifications for this purpose [1] is not discussed in this paper.

4.1. *Specifying Semantics with TYPOL*

TYPOL is an implementation of Natural Semantics [17]. A TYPOL program is roughly an unordered collection of axioms and rules of inference. A rule has basically two parts, a numerator and a denominator. These can contain variables that allow the rule to be instantiated. The numerator is an unordered collection of formulae, the *premises* of the rule. Intuitively, if all premises hold, then the denominator, a single formula, holds.

TYPOL specifications can be organized in modules to separate different semantic concerns. For example, in dynamic semantics, one can distinguish structural rules that state the evaluation of language phrases from the management of storage.

A typical rule is shown below:

$$\frac{s \vdash \mathrm{ID} \Rightarrow x \qquad s \vdash \mathrm{EXP} \Rightarrow v \qquad s \overset{\text{update}}{\vdash} x, v \Rightarrow s_1}{s \vdash \mathrm{ID} := \mathrm{EXP} \Rightarrow s_1}$$

This is the dynamic semantics rule for an assignment $\mathrm{ID} := \mathrm{EXP}$. It uses rules for evaluating identifiers and expressions and then using another set of rules named "update" adjusts the storage.

4.2. *The TYPOL compiler*

A TYPOL specification must be compiled before it can be used in CENTAUR. The compiler includes a type-checker and a code generator. Both have been written in TYPOL itself.

Abstract syntax terms occur in most rules. They have to be valid terms w.r.t. their abstract syntax. Every such term is typed with a syntatic category. The type-checking phase of TYPOL compilation uses this information in verifying a TYPOL specification and generates an intermediate form.

The output from the type-checking phase can be plugged into several code generators. Currently we use a Prolog code generator written in TYPOL. Here we take advantage of the similarity between Prolog variables and variables in inference rules. Roughly speaking, the denominator of a rule maps to a clause head and the premises to the clause body.

Given a semantic specification we want to use the computer to solve various *equations*. Typical unknowns are values, types, states, etc. An equation is turned into a Prolog goal which is then solved by the Prolog interpreter.

4.3. *Operating the logical engine*

The TYPOL compiler generates Prolog code. To use this code from CENTAUR, a goal must be constructed and mailed to the Prolog engine. Goals usually have an abstract syntax tree as argument. Conversely, the result of a computation may be an abstract syntax tree represented as a Prolog term. This requires efficient generic coercion primitives between VTP trees (in Lisp) and Prolog terms. During these coercions, Lisp and the Prolog implementation communicate via a common storage area to speed up parameter passing and to avoid unnecessary *cons*ing.

When the arguments for the goal are ready, the goal is "mailed" to Prolog for resolution. To control precisely the Prolog execution from within CENTAUR, we communicate via messages with the Prolog engine. For each rule Prolog sends one of 4 possible messages: try (a rule), proved (a rule), back (to backtrack over a rule), or fail (to prove a rule). The basic replies one can give are: continue, fail (to force backtracking), retry, and abort. Using this low-level protocol one can step in various ways through the execution of TYPOL specifications.

4.4. *Transforming TYPOL programs*

Usually TYPOL specifications are directly compiled and executed. However, it is possible to analyze and transform specifications into useful variants.

A first example concerns type-checking. Executing directly the rules results in a "pure" type-checker, that succeeds or fails. More useful is an incremental type-checker with diagnostics. Experiments show that such a type-checker can be obtained by mechanical transformation [11]. Another example concerns translations. Here one can deduce a correlation between tree addresses in the source code and tree addresses in the target code. Using this correlation, one may execute the target program and animate the source. Also a partial evaluator has been written which given a TYPOL specification and a program generates the restricted set of rules needed to interpret this program [12].

4.5. *Discussion*

A good TYPOL environment exists within CENTAUR and many specifications have been written. These allow one to prototype quickly those aspects of a new language that are difficult. TYPOL has been used to enrich an environment for a very small language with a type-checker, an interpreter, and a translator into assembly code. In this environment, one can follow the step by step execution of programs, and a debugger exists (breakpoints with backward and forward execution).

The compiled TYPOL code is sufficiently fast. Improvements could come from either using a Prolog compiler or using a different code generator. Methods to control the use of storage and the TYPOL execution are under investigation.

5. The man-machine interface

An essential component of CENTAUR is the user interface. At a very general level the user interface of a system contains all the components for the user to modify the state of the system and for the system to inform the user about its state.

5.1. *Design Goals*

The CENTAUR system used interactively receives user input, through the keyboard and mouse, and returns output to the user, through various devices. However, components of the system can be useful when integrated in specialized tools, such as compilers. Thus the first design goal is to separate carefully the man-machine interface from the system. Our second goal is to keep the interface open-ended in two directions.

- A rich set of graphic objects has to be available but must be extensible. The graphic objects are accessible by programming primitives but interactive tools would facilitate their design.
- An environment developer or user may want to maintain relationships between the various objects that appear on the screen. Thus a mechanism that provides for interaction control must be present.

5.2. *Graphic Objects*

The CENTAUR interface provides a library of standard graphic objects such as buttons, menus, etc. and provides a *framework* to define specialized graphic objects. One such special graphical object, called a *view*, is used to display the abstract syntax trees manipulated by CENTAUR. A view usually has a banner, a scrollbar, a resize corner. In its main area – the view proper – the buttons of the mouse have a distinct meaning. Two of them control selection of the current expression while the third pops up a menu dependent on this expression.

To define graphical objects we specify *separately* their appearance and their behavior. A set of Le_Lisp primitives, the AIDA package [7] is used to describe the geometry of the objects. These include basic constructors such as string, box, etc., and geometric combinators such as row, column, overlay, etc. Compound objects are built using these primitives. For some objects one wants to associate a behavior. This is the case for a button or a menu, for example. These objects are seen as reactive systems that change state in response to input events (from the mouse and keyboard) and generate output events. The behaviors of all the objects are written in ESTEREL [2], a special purpose language for real-time applications. The ESTEREL compiler generates a Lisp description of a finite automaton. The automaton is hooked in a systematic fashion to inputs and output events (see also [5]). Graphic objects are presented in more detail in [4].

The graphical interface of the CENTAUR system is built on top of the Le_Lisp virtual window system. This underlying system provides primitives for creating and modifying windows and subwindows together with primitives to handle both keyboard and mouse input. This isolates us from the level of the window system used in the operating system. The Le_Lisp virtual window system runs on various window managers.

5.3. *Interaction Control*

The man-machine interface permits one to manipulate the structured objects in *views* on the screen. As with any multi-window system, there are problems with the consistency of what appears on the screen.

In CENTAUR it is possible to have several views on the same structure. Clearly a modification in one view is to be reflected in the others. A difficulty can arise when a destructive modification is made in one view while another view was looking at only a part of the destroyed expression. Here some adjustment must be done for the latter view to show some valid part of the object. In this case, to keep the screen consistent we maintain a relationship between views and structured objects.

The relationships we want to maintain between objects and how they effect screen consistency can take many forms. For example, the user may always want that two views show the same current expression. Implicit relations exist between some objects, such as a source program, its code, and the translator that was use to derive the code from it. The user may want a change in the source program to result in running the translator. And what if the translator itself is changed? Thus we want a general mechanism that allows for this type of screen control. Such a mechanism requires knowing the relationship between objects and views as well as checking what relations are to be maintained.

5.4. *User Viewpoint*

From the user's standpoint, the interface is a multi-window system that is menu-driven. The standard manipulations (open/close, move, resize, etc.) exist for the various windows. Through the system the user can manipulate various types of structured objects. To look at an object one opens a "view" on the object. It is possible and useful to have several views on the same object. Each view has its own emphasized subexpression, and possibly its own pretty-printer for the object. Views are aware of the type of the object they contain, as different menus pop-up.

For manipulating the objects in views there are menus, both pop-up and pulldown, and menubars. These graphic objects give access to the standard generic manipulations (e.g., cut and paste, reading and storing objects, elementary navigation, etc.). But the commands available depend on the type of the current expression, in the current view. Menus are dynamically extended for different environments. The pop-up menu associated with a view depends on the current expression in that view. Thus the view can contain an object that is "mixed", i.e., contains expressions in more than one formalism. The popup menu can therefore contain actions that are formalism specific. It may, for example, permit one to compile or interpret a program or to use special searching mechanisms.

It is also possible to bind the commands for keyboard input, here again the user may customize the binding from view to view. Dialog boxes allow the system to query the user for input and allow the user to parameterize and control the system. Various kinds of dialog boxes appear on the screen, using several types of graphic objects: buttons, browsers, text-editable areas.

5.5. *Discussion*

The performance of the graphical objects is good; that is, response to mouse and keyboard input is reasonably fast and one can work with the system. Once all the basic objects are in place it may be necessary to optimize the interface with the underlying window manager. Further development of graphic objects and primitives is needed, in particular to improve the appearance of the objects themselves and to provide a more symbolic way to specify their construction. In particular we would like tools to aid an environment developer in designing his dialog box and attaching it properly to his application or translator.

Operations that involve relations between structured objects and windows seem a little slow. However, further experiments are needed before addressing performance issues. We want to experiment with a more general mechanism for different kinds of relations. At this time, the logical engine of CENTAUR seems well adapted to build an experimental system for this *logical* control of the interface.

6. Conclusion

The construction of CENTAUR is not complete. Very few selected users have had actual experience with the system. Still, it is sensible to examine the system and see how it meets the challenges presented in the introduction.

i) Is it possible to use as the description of a programming language a formal specification in the sense used by researchers in programming language semantics?

On the basis of our experience, extracting useful behaviors from a formal definition is in fact *surprisingly easy*. As an example, handling reverse execution involves a minute programming effort. More work is necessary in the *analysis* of formal definitions, a very promising research area. Also, experiments with parallel languages are just beginning.

ii) Is it possible to obtain on that basis a efficient system, both in terms of speed and in terms of memory space?

The answer to this question is subjective: what is efficient for one user may be considered unacceptable in another context. The main source of skepticism about the efficiency of CENTAUR stems from the use of Prolog as a logical engine. First, experience has shown that Prolog used as a coroutine in CENTAUR is a remarkably flexible tool and, in fact, often suggests novel approaches. Second, provided it is used only when appropriate, even an interpreted Prolog is *fast* – i.e. comparable to compiled Lisp. However, it is not very easy to control the use of space in CENTAUR and much effort in this direction is necessary in the future.

iii) Is it possible to create a man-machine interface that is comparable in convenience to what can currently be found in dedicated commercial systems?

It would be preposterous to answer affirmatively today. But two remarks can be made. First, building a good man-machine interface is an extremely interesting and rewarding task: the software architecture is non-trivial, the human factors questions are challenging, the logical control of the interface is novel. Second, the literature in this area seems to be lagging behind the commercial achievements. As a result, it is easy to make errors and to underestimate the manpower needed to obtain satisfactory results.

Acknowledgments

Beside the authors of this paper, several people were instrumental in the design and implementation of CENTAUR. L. Rideau-Gallot worked on the virtual graphics interface of Le_Lisp. J. Despeyroux and L. Hascoët suggested many improvements. F. Montagnac organized a good development environment and assisted in all Unix matters. The work of M. Devin of ILOG helped establish the man-machine interface.

Throughout the development of CENTAUR, we have had numerous discussions with the CWI group led by Paul Klint.

NOTE

As a matter of reference, we give an account of the size of the system as it is today (32231 lines). The numbers of lines of code indicated in the table below reflect only the generic system modules and exclude: 1) the Prolog interpreter itself (7400 lines in C, 2200 lines in Prolog), 2) the sample languages already described in CENTAUR (several thousand lines of METAL, PPML, and TYPOL).

Function	Size	Language
Database	8287	Lisp
Using database	7329	Lisp
Logical Machine	998	Lisp
	800	Prolog
	500	C
	1796	Typol
Man-Machine Interface	11662	Lisp
	859	Esterel

REFERENCES

[1] Bergstra J.A., J. Heering, P. Klint "ASF – an algebraic specification formalism", Report CS-R8705, Centre for Mathematics and Computer Science, Amsterdam, 1987.

[2] Berry G., P. Couronné, G. Gonthier, "Synchronous Programming of reactive systems: an introduction to ESTEREL", in *Proceedings of the First France-Japan Symposium on Artificial Intelligence and Computer Science*, Tokyo October 1986, North-Holland.

[3] Chailloux J., M. Devin, J-M. Hullot, "Le_Lisp, a portable and efficient Lisp system", *Proceedings ACM Symposium on Lisp and Functional Programming*, Austin, Texas, 1984.

[4] Clement D., Incerpi J., "Graphical Objects: Geometry, Graphics, and Behavior", 3^{rd} Annual Report, Esprit Project 348, 1987.

[5] Clement D., Incerpi J., "Specifying the Behavior of Graphical Objects Using Esterel", INRIA Rapport de Recherche 836, April 1988.

[6] Despeyroux T., "Executable Specification of Static Semantics", in *Semantics of Data Types*, Lecture Notes in Computer Science, Vol. 173, Springer-Verlag, June 1984.

[7] DEVIN M. ET AL., "Aida: environnement de développement d'applications", ILOG, Paris, 1987.

[8] DONZEAU-GOUGE V., G. KAHN, G. HUET, B. LANG, J-J. LÉVY, "Programming environments based on Structured Editors: the MENTOR experience", in *Interactive Programming Environments*, D.R. Barstow, H.E. Shrobe and E. Sandewall (Eds.), McGraw-Hill, 1984.

[9] DONZEAU-GOUGE V., G. KAHN, B. LANG, B. MÉLÈSE, E. MORCOS, "Outline of a tool for document manipulation", *Proceedings of IFIP Congress 1983*, Paris, North-Holland.

[10] GETTYS J., R. NEWMAN, R.S. SCHEIFLER, "Xlib – C Language X Interface, Protocol Version 11", MIT project Athena, February 1987.

[11] HASCOET L., "Transformations automatiques de spécifications sémantiques. Application: un vérificateur de types incrémental" Thèse, Université de Nice, March 1987.

[12] HASCOET L., "Partial evaluation with Inference Rules", INRIA Rapport de Recherche 817, April 1988. To appear in New Generation Computing.

[13] HEERING J., P.R.H. HENDRIKS, P. KLINT, J. REKERS, "SDF Reference Manual", Centre for Mathematics and Computer Science, Amsterdam, 1987.

[14] HEERING J., P. KLINT, J. REKERS, "Incremental generation of lexical scanners", Centre for Mathematics and Computer Science, Amsterdam, 1987.

[15] HEERING J., P. KLINT, J. REKERS, "Incremental generation of parsers", Centre for Mathematics and Computer Science, Amsterdam, 1987.

[16] JOHNSON S.C. "Yacc: Yet Another Compiler Compiler", Unix Programmer's Manual, 7th Edition, Bell Telephone Laboratories, January 1979.

[17] KAHN G., "Natural Semantics", *Proceedings of STACS 1987*, Lecture Notes in Computer Science, Vol. 247, Springer-Verlag, March 1987.

[18] KAHN G., B. LANG, B. MÉLÈSE, "METAL: a formalism to specify formalisms", Science of Computer Programming, Vol. 3, pp. 151-188, North-Holland, 1983.

[19] LANG B. "The Virtual Tree Processor" in *Generation of Interactive Programming Environments*, Intermediate report, J. Heering, J. Sidi, A. Verhoog (Eds.), CWI Report CS-R8620, Amsterdam, May 1986.

[20] MILNER R. "A proposal for Standard ML", *Proceedings of the ACM Symposium on Lisp and Functional Programming*, Austin, Texas, August 1984.

[21] NAISH L., "The MU-Prolog 3.2 Reference Manual", Technical Report 85/11, Department of Computer Science, University of Melbourne, October 1985.

[22] MORCOS-CHOUNET E., A. CONCHON, "PPML, A general formalism to specify pretty-printing", *Proceedings of IFIP Congress 1986*, Dublin, North-Holland.

[23] PEREIRA F., D. WARREN, D. BOWEN, L. BYRD, L. PEREIRA, "C-Prolog User's Manual, version 1.5", SRI International, Menlo Park, June 1985

[24] REPS T., "Generating Language Based Environments", Ph.D. Thesis, Tech-Report 82-514, Cornell University, Ithaca NY, August 1982.

[25] TEITELBAUM T., T. REPS, "The Cornell Program Synthesizer: a syntax-directed Programming Environment", Communications of the ACM, vol. 24 (9), September 1981