ABSTRACT

Programs are integrated hierarchical structures of formal texts - instructions - and informal texts - documentation. Only if there is a mechanical connection between program design and program development there can be higher chances that the program documentation, originated at design time, will not, sooner or later, become obsolete with respect to the current state of the system. DUAL is an interactive, incremental, intelligent editor of program and documentation texts to be used for software implementations whose anticipated life cycle is significantly long to deserve much attention. The screen interface plays a unique role in DUAL since it allows a visitor (designer or maintenance engineer) "to replay" as many times as desired the evolution of the system from design to implementation. The access to the hierarchically organized Design Information is made easy and natural by the DUAL video oriented user interface.

1. Introduction

According to many authors [BAUE 76, BUDA 77], program design is an evolutionary process whose evolving steps must be properly documented for ease of understanding and maintenance. As Chestham [CHEA 81], points out program maintenance is simplified when the program to be modified can be viewed as a documented implementation of a readable abstract model capturing the essence of the behaviour of the system. DUAL enforces the idea that program design is a "continuous" process transforming a design draft into a final implementation. The various steps of the design process are "photographed" and the "movie" which shows the derivation of each piece of program code starting from its initial design idea can be dynamically replayed on the screen at the visitor (the user) will.

The DUAL view of the program data base is an abstract tree to be visited interactively from a video-keyboard workstation. DUAL enforces a strict isomorphic association of the program text with the program documentation at all levels by forcing the implementor to give an identical tree structure to both.

Comments of a program are considered part of the system documentation and are shown only on explicit request to a visitor "walking" on the abstract design/program tree. Comments are given a scope, i.e. a region of the program text to which they refer to and are, in a certain sense, typed: a comment is either a comment à la Hourie, expressing a pre or post-condition of the program fragment or is a behavioural comment, e.g. an alternative English description of the program behaviour.

DUAL can be used as an intelligent editor of a Program Design Language, as an incremental intelligent editor of the programming language or as an interactive tool to organize and retrieve program texts and documentation.

DUAL is currently being implemented "around" the language Pascal, but it can be easily adapted to other programming languages like ADA, or even around a rather conventional language like COBOL. Actually, the current version of DUAL is built around Pascal, [OLIV 80], an extension of Pascal for system programming, which includes separate compilation of module/monitor units, abstract data types and facilities for process synchronization.

DUAL takes also special advantage of the dynamic features of the screen-keyboard interface. A programmer can visit the design data base by using only a restricted set of commands. Essentially, only access paths to the design information are shown on the screen; they may be entered on request by a visitor and their expansion will show other accesses of the same nature, in a recursive way.

Since a program is no longer treated as a linear string of characters but as a tree of linked informations, DUAL provides a set of offline utilities to generate several print-outs integrating in various ways the informations stored in the design data base.

An important objective is thus achieved: the programmer routine task of repeating redundant information is minimized. Once the relevant information is generated and the semantic association made, the computer takes the burden of formatting, line-spacing and listing it.

2 Program development and documentation

by step-wise refinements

Well known works by Dijkstra [DIJK72], Wirth [WIRT 71], Gries [GRIE 73] and others have assessed the usefulness of deriving a program in a hierarchical way. In any case, whenever has been followed to actually build the program, it is illuminating to be able to study and examine it by increasing levels of details at the reader's convenience.

As a consequence, it is natural to introduce in a language like Pascal or ADA, or in a programming development support, a new concept: the concept of abstraction. An abstraction has a name, say A, a lexical denotation, say "A", and is a language construct that can be used whenever statement, list-of-statements or expression is legal in the original grammar.
The meaning of an abstraction is manifold. In the first place, an abstraction can be used in the early stages of program development to stand for a fragment of program text yet to be developed. A program typically containing one or more occurrences of abstractions which are not yet expanded is an abstract program and cannot per se be executed (unless support systems for symbolic execution are made available); it can only be analyzed for formal correctness and be used for expressing in a precise procedural way the design structure of the program.

An abstract program stands also for a piece of program text that is considered too detailed to be shown at the current level of description of the program. The ability to conceal unwanted details is useful for understanding a program structure, one of the primary objectives of program maintenance. But if the reader wants to investigate also the details, he may do so by means of the (interactive) expansion mechanism.

An abstraction is finally a tool for imposing a hierarchical structure on a program text which in a first approximation is considered as a tree. Such a tree structure can be used to give an identical tree by allowing the program documentation by a physical association of each program fragment with the corresponding documentation text (see later the duality of functional versus program expansion).

The idea of supporting stepwise top-down programming is certainly not new. From the initial attention to programming languages, the interest shifted to design languages [VANL 76] and further to integrated development tools [TEIT 81, DONZ 80, CHEA 81, ASIR 79, HABE 79]. DUAL fills in the class of integrated program development tools and its specific objectives are to provide a smooth mechanical transition from design to implementation, to tightly link documentation to program structure. We believe the most valuable documentation of a program is the structure of its stepwise top-down development. In an industrial environment where long-lived large programs are developed and maintained, the process of understanding a program is repeated over and over again (on the same piece of text). The process of understanding (to modify or fix) is by no means easier than the one of developing and we believe it should be to be supported by the same methodology and tools.

A lot of the work being done on top-down program development tools is centered around the notion of supporting syntax-directed development [TEIT 81, HABE 79, LEWI 81]. This means that an abstraction can be introduced only if it corresponds to a nonterminal symbol of the programming language. This is a somewhat restrictive approach, and, in some cases, the DUAL abstraction mechanism provides more freedom to the programmer.

For example, consider the abstract statement:

```
if "table is not full" then "insert the item"
```

If the abstractation "insert the item" is later expanded to a list of statements, then a begin-end pair is to be inserted (while it shouldn't if the abstraction would be embedded already in a list of statements). This is a trivial example, which would cause just minor annoyances to the programmer. But the if-part is more interesting. In practice it is not always the case that the abstraction "table is not full" is expanded to just a boolean expression. More generally, a programmer will want to expand it into a program fragment setting up the values of variables that are subsequently used in a boolean expression. For instance "table is not full" might expand as

```
search loop setting the variable z
```

The result of substituting the defining text of "table is not full" into the original abstract program should yield the text

```
search loop setting the variable z
if z then "insert the item"
```

Let us call such expansions extended boolean expressions.

DUAL allows abstractions occurring as control clauses of statements, while-statements and repeat-statements to be expanded as extended boolean expressions. In terms of the previous example the construct

```
repeat "insert the item" until "table is full"
```

once that "table is full" is expanded will yield the text

```
repeat "insert the item" "search loop setting z" until z
while the following construct
while "table is not full" do "insert the item"
```

will yield the text:

```
loop
"search loop setting z"
if z then exit
"insert the item"
end loop
```

which is implemented using a while true and the Pascal exit statement.

However, there are other cases where a high level program description requires a more complex program manipulation. A typical example is the abstract control structure

```
"for any element z of S" do
"action on z"
```

which can be implemented in different ways depending on the concrete structures used to implement the concept of the abstract set S. Cheatham (CHEA 81) faces this problem by providing the user with the possibility of defining rewriting rules, one for each concrete implementation of the abstract set. An occurrence of a rewriting rule within the program text is detected by pattern matching algorithms; when the matching is successful the appropriate substitutions are made for parameters in the replacement part of the rewriting rule. However, the concrete representations of an abstract set are too numerous with too many irrelevant variants that the mere work of listing them and consulting the list is perhaps greater than the advantage of a prechecked program transformation. In any case, if such a repertoire exists the programmer can consult it and perform manually the intended transformation.

For this reason DUAL provides only a documentation mechanism that we call program transformation. A transformation of a program text is a refinement of the original text that cannot be accomplished by an expansion of an abstraction. Its correctness is the programmer's responsibility. Its purpose is to document an evolution of a program fragment. In this sense it is not a program historic version and is subjected to different treatment and limitations. The emphasis is more on capturing the intentions of the programmer and in encouraging him to express his algorithms in abstract.
terms rather than in providing him with a set of prechecked high level constructs. Obviously, a transformation can also be intended in the general sense of [BUDA 77]. DUAL can thus be used to simulate manually a promising methodology waiting for the automatic transformation systems of the next generation.

3. Documentation of a program

Axiomatic - i.e. input/output - and procedural - i.e. operational - descriptions of programs have been presented as alternative candidates for program documentation. Comments of programs reflect this business situation; in fact one can easily show examples of comments à la Holle that are intended to denote invariants, i.e. conditions that are satisfied before (or after) the execution of the corresponding program fragment and procedural comments, i.e. comments conditioning in one short sentence the intended behavior of a certain fragment of text. Both points of view are legitimate and there are practical situations where one can be preferred to the other. DUAL provides two distinct mechanisms for those alternative forms of documentation: the functional expansion of an abstraction and the invariant abstraction.

The text making up the expansion of an abstraction is considered as the program expansion of that abstraction. But an abstraction can be expanded also functionally: the resulting text being the procedural alternative description of the abstraction in a natural language or in a formalism different from the programming language. In this way procedural comments are given a scope which is a program text making up the abstraction. The abstractions of a given program which have a functional expansion are highlighted on the screen so that the user can ask to be shown them. With reference to the previous example the program expansion of the abstraction "insert the item" is the program fragment actually inserting the item in the table, while its functional expansion might be, for instance, the following English text (see [WIRT 76], pag. 216):

"The table is a balanced tree, and thus the item insertion consists of the following:

1. Follow the search path until it is verified that the item to be inserted is not already in the tree.
2. Insert the new item and determine the resulting balance factor.
3. Retreat along the search path and check the balance factor at each node."

An invariant abstraction may be seen as a comment of the environment language (which is Pascal in our case). Normally an invariant abstraction is not shown on the screen; its presence is marked by a special invariant symbol. An invariant symbol can be program or functionally expanded. The program expansion is treated as instrumentation auxiliary code to be inserted optionally in the source code of the program. The functional expansion of the invariant symbol is a natural language text describing the conditions satisfied by the program variables or a first-order predicate calculus formula for those classic examples of programs, currently used to teach formal program correctness. An invariant symbol can be introduced whenever a statement is allowed since its program expansion is an executable statement. MENTOR [DONZ 80] policy is in this respect quite different: conditions are there treated as nodes attributes.

4. Interactive visit of the Design data base

The Design data base of a Pascal+ module is made up of two isomorphic abstract trees; a visiting programmer can start his/her visit of the currently implemented (design data base) from its root or resume it from exactly the same point it was left at the end of the preceding session (see section 8). An abstraction of a given name can be in one out of four states, obtained from the possible combinations of the associated expansion (program, functional, both, no expansion); accordingly, four different screen font attributes are used for its representation.

A visitor has four commands available for tree traversal: program expand, functional expand, reduce, done, abbreviated respectively as E, F, R, D. The distinction between visiting the tree and editing it is subtle and we will return on it in section 8. The commands E and F correspond to the general request of "wanting more detail" while R and D mean the opposite. The E and F commands expand the abstraction pointed to by the current cursor position on the screen. To make an example, let us suppose that the current screen content is the abstraction "queen problem" and that a formal expansion of "queen problem" has been previously entered in the system. The E command replaces the current screen content with the following view:

```
queen problem:
  procedure try (i: integer);
  if "k-th candidate is acceptable" then
    "record it"
    if i < n then try (i+1) else "print"
    "cancel recording"
  end
```

```
Figure 1
```

where D is the invariant symbol and "D" is a standard abstraction name standing for the declaration parts of the procedure. The expansion of the abstraction "queen problem" contains four abstractions the first two of which have been so far program expanded.

The visitor can edit the current version of "queen problem". When he is done, he can issue for example an E command while the cursor is pointing to the abstraction "k-th candidate is acceptable". The previous content of the screen work area will be replaced by:

```
when done:
  begin
    if "k-th candidate is acceptable":
      z[k] = A[i+k] \ A[i-k]
  end
```

...
In other words, the expansion of "k-th candidate is acceptable" is shown framed in the context of the upper node. There is however one point to note: the context is accessible to the programmer in read-only mode. There is only one region, the editing region, which can be modified and this is the most recently expanded abstraction. As a consequence of this fact, a programmer cannot alter the source code of a program unless he does it at the proper level of abstraction thus assuring always a strict coherence between the high level design and the final source code.

If the abstraction pointed by the cursor had never been expanded before, the framed editing region would appear "blank" and the programmer could fill it in. An abstraction can be unnamed. In this case, it is shown on the screen as a sequence of three dots and is used to represent an action (a book - keeping action, for instance) that is not worthwhile to get a specific name. However, if the text of an abstraction has undergone a program transformation, then the expansion of the abstractions there occurring will first indirectly cause the display of the transformed text.

The R command is the inverse command of both E and F in the sense that while E and F go top-down, R allows the programmer to go bottom-up towards the root of the Design database. The effect of R depends on the position of the screen cursor: the abstraction-node from which the pointed information directly "derives" replaces on the screen all the information which had been derived from it. The R operation is "similar" to the familiar reduce operation of bottom-up parse algorithm. The R command is used when the study of a detail is over and the programmer desires to come back to a higher level of abstraction. Let us point out that the R command can help a programmer to freely inspect and/or amend various parts of his program without carefully planning an ordered tree traversal. For example, with reference to Figure 2: if the cursor lies within the framed region, then an R command will replace Figure 2 with Figure 1; if the cursor is outside the framed region, then an R command with replace Figure 2 with "queen problem".

The D command allows the cursor to go beyond the boundary of the current abstraction, a sort of return command for abstraction. As soon as the editing of an abstraction expansion is over, the system performs a syntactic and semantic analysis of the text returning a list of errors. If the text contains no syntactic errors it is stored in the data base as a syntactic tree, otherwise it is kept in source form but no expand operation are allowed from it. If the text contains a semantic error or if its semantic analysis cannot be performed for lack of information (an undeclared identifier), a message is shown on the screen and, in the latter case, an entry of an undeclared item is also made in the symbol table.

The first version of DUAL makes use of a character oriented editor within an editing region. We are evaluating whether to incorporate a syntax directed incremental program synthesizer [Test 81, Lewi 81] to support abstraction expansion. On one side, a synthesizer is very much in style with a DUAL approach, on the other side, there is less gain to be obtained since a DUAL user can make an extreme use of the abstraction expansion mechanism strictly following the grammar of the source language (up to the point of one statement expansion of unnamed abstractions) and, in some sense, he can simulate the behavior of a synthesizer.

5. Interactive access to the extended symbol table

The expansion game can also be played with respect to the program identifiers and is called lateral expansion. In order to show lateral expansions a side window, provided of an auxiliary cursor, is created on the screen. The program (lateral) expansion of an identifier is the source program fragment declaring it, the functional expansion is a comment describing, in full detail, the intended use of such a identifier. The program expansion of a procedure identifier is the procedure declaration with its body shown at the highest level of abstraction: any expansion of identifiers or abstractions therein contained replaces the previous side window content, since lateral expansions cannot be nested. With reference to the example of Fig. 2, if the programmer E expands the identifier a, the upper right corner content of the current screen view is replaced with

```plaintext
var a: array [1..max] of a-type
```

If the user moves the side window cursor to point to "a-type" and issues an E command, the previous side window view would be replaced by

```plaintext
type a-type = boolean
```

In other words, lateral expansion are not nested and the R command returns the system in normal mode. As another example we may add that the lateral expansion of the identifier "try" within the text "try (n)" would cause the display of Figure 1 within the screen side window. Let us point out that the nesting limitation of lateral expansions is necessary in order to avoid cases of recursive generation of text code on the screen work area. Lateral expansion provides a sort of interactive access to the symbol table, very useful during program maintenance since it frees the programmer from ceaseless browsing into the source file or into the listing pages. The comment associated to an identifier declaration, which hopefully conveys its intended use, is actually needed within the context of its points of use. The functional lateral expansion provides exactly this facility.

6. Handling of declarations

Program declarations can also be entered or altered by expanding the standard abstraction "D" of the procedure text. In this case there are no nesting limitations for the screen access and, moreover, the programmer has full echo in determining the most appropriate scope by selecting the "D" abstraction of the desired procedure. Declarations are handled incrementally: an identifier can be used "before" the programmer decides to declare it and any declaration can later be modified at any time. It is necessary to separate semantic analysis from syntax analysis and to include in the semantic analysis any access to the information
that is usually kept in a symbol table, like data structuring.
In principle, semantic analysis should be performed every
time a subtree is made visible in the user work area and the
user work area should be entirely reanalyzed each time a
modification to a declaration, accessed in lateral mode, is
made. In practice, some analyses can be avoided by using
modify-flags. Incremental semantic attribute evaluators are
described in [ DERE 81 and REPS 81], but have not influ-
enced the current DUAL implementation.

7. Offline printing utilities.

While the interactive access to the design data base presents
certain advantages in flexibility, the printed information has
still some other unmet needs. Not a secondary factor
is the mediocre quality of average present day CRT screens,
where confronted to the information written on paper.

Presently a DUAL user can select a number of options which
can be combined to obtain various forms of listings. Some
interesting combinations are the following:
a) Source Pascal+ listing, supplemented with the comments
associated to all identifier declarations.
b) Source Pascal+ listing with a header describing each
procedure at the highest level of abstraction.
c) Selective listing of the expansions that have undergone
modifications.

The description of each module is given first at the
highest level of abstraction, then each abstraction is
defined in turn. The final complete Pascal+ listing, can
be supplemented as in a).

A hierarchy of the functional expansions can be supplied
in the same order as found in the program expansions.

A better approach would take profit of the usually large
number of characters per line of a computer or plotter listing
to provide a side by side printing of program and functional
descriptions. In general, a lot of freedom is given to the
designer of the print-outs since the program listing which
includes all sorts of comments is no longer an input to the
compiler.

8. Implementation and extensions

Initial work on a DUAL type of development started in
1975, when one of the authors [FETR 76] was involved in
specifying a PDL language based on PLI and on step-wise
refinements in terms of named abstractions. The analyzer,
called Design Analyzer, was implemented by Draper Laborato-
ries [DRAN 75] for Olivetti Research and development and
was in use for some years. The project DUAL was initially
In order to minimize the implementation effort, the initial
experimental version of DUAL has replaced only the language
processor of the programming development support environ-
ment running on a VAX system. In other words, once that
the target code is generated (the target is a 28000 micro-
processor), one needs to use the conventional linkers and
loaders to execute the object modules. A second implementa-
tion of DUAL is being developed under UNIX on the Olivi-
tti network of PDP 11/70's and is geared to support large
programs written by a cooperating group of programmers.

Since a program is stored in the form of syntactic tree, the
expansion of each abstraction requires unparsing (and pretty
printing) of the associated portion of program text. Also the
program declarations are kept in the form of symbol tables,
and therefore as syntactic trees.
The first implementation of DUAL has been derived in fact,
from a modification of an existing compiler of Pascal+ and is
of course incremental extended compiler.
The major modifications to the original compiler have been:
i) introduction of the concept of abstraction with associa-
ted links to the program subroutines;
ii) separation of the semantic analysis phase from the
syntax analysis phase and delay of memory allocation to
the code generation phase.
iii) redesign of the garbage collection mechanism to exploit
the greater movement and fragmentation of source code
due to the abstraction mechanism coupled to the incre-
mental syntax analysis. In fact, editing of a single
abstraction requires to dispose the old syntax tree of
that unique abstraction, while building a new tree and
keeping all the remaining syntax trees of the source
code unmodified.
iv) implementing the video-keyboard interface and the disk
management of functional expansions.
v) implementing the off-line utilities for generating the
various options of print-outs.

A state-dependent syntax tree

To better understand some implementation issues which are
specific of the DUAL approach, let us restrict the use of the
term expand to the action of defining the associated text of
an abstraction and let us introduce the term visit for the
action of inspecting the text associated to an expansion of an
abstraction.

In correspondence with the fact that DUAL provides at the
same time a design language and an implementation language
and that the transition from the former to the latter is
dynamic, each DUAL abstraction plays a double role of
terminal symbol of the design language and of nonterminal
symbol pointing to the associated expansion text. A DUAL-
Pascal source text is stored as a syntactic tree where the
abstraction nodes are represented in such a way that they
can switch from nonterminal-state to terminal-state depending
on whether they have been visited or not in the
course of the current session.

At the start of a session all abstraction nodes of the syntax
tree are in exactly the state they were left at the end of the
preceding DUAL session so that a "visitor", who is the
owner of the module and has write access rights upon it, is
put back in the same level of detail he had closed before
closing his preceding DUAL session. As soon as a visitor visits
any abstraction node (issuing an E command) then that
node changes state becoming "transparent" and its name is
replaced on the screen by the pretty printing of its subtree.

For example, if the abstraction nodes of the abstract syntax
tree of Figure 3 are all in nonterminal state except the node
"something", then the pretty printing of the syntax tree is
the text:

if T or V then
for l = 1 to max do "something"

If both the abstractions of Figure 3 are in terminal state the
resulting text is

354
An expand operation logically amounts to the switch of an abstraction node from terminal-state to nonterminal-state, while a Reduce command is equivalent to ascending the syntax tree up to the nearest abstraction node, changing its state to terminal state and pretty printing the resulting syntax tree. To assure that all descendants of an abstraction node that happens to be currently in terminal-state are also in terminal-state, the Reduce operation performs a complete traversal of the subtree whose root is the abstraction node in question.

Abstraction nodes play the role of a context dependent begin-blocks in the sense that both the pretty printer and the code generator will consider them as begin blocks if the abstractions are inserted in certain contexts.

The situation is slightly more complicated for the extended Boolean expressions occurring a control clauses of control statements. In fact, one has to face a program transformation (and its inverse). To obtain the desired result other types of state-dependent syntax node have been added, namely a context-dependent beginblock (CDBB) and a control abstraction. A CDBB is a state dependent node: in unvisited state it is "transparent", in visited state it is treated as a begin block node if not inserted in a statement list otherwise it is transparent. Moreover a CDBB functions as a conditional statement list, in the sense that it points to both its descendants or only to its right offspring depending on its state.

In this way the same syntax tree can be used to represent a program and its "transformed" version depending on the visited or unvisited state of one node.

For instance in correspondence with the text

```
if "table is full" then 3
```

equivalent to

```
while undone do
  if "table is full" then 3
```

The code generator will generate a control abstraction node in visited state representing the clause corresponding to "table is full" while the pretty printer will print a "something" whenever a control abstraction node is encountered.

The next figure shows the control abstraction node and the CDBB being evaluated for a "table is full" condition as the first part of the statement list.

The screen after expanding "table is full" would look as in Figure 4. An abstractions occurring as a control clause is represented by a control abstraction node (see Figure 4). In unvisited state a control abstraction is a terminal node, in visited state it is a nonterminal node pointing to its rightmost offspring.

```
   while undone do
     if "table is full" then 3
   end
```

If the control-abstraction of Figure 4 and the CDBB node are in visited node, then the syntax tree is pretty printed as follows:

```
if "table is full" then 3
```

However, immediately after the user has just expanded "table is full" the screen appears as in Figure 5.
In other words, the control abstraction “table is full” appears as an extended Boolean expression and, in this context, the third and fourth pointers of the associated control node are used.

The remaining cases of extended Boolean expressions occurring as control clauses of while and repeat statements are treated in a similar way.

Extensions

One possible extension of the current implementation can be in the direction of emphasizing the documentation aspects of DUAL, for example by allowing the user to associate a “type” to each abstraction. The type of an abstraction can be used to select different preformatted screen forms in order to standardize and make easier the writing of the associated documentation (see TATUM 81). In general, we may point out that the use of DUAL is greatly influenced by the methodology or the programming standards prevailing within each software development environment.

No abstraction can presently occur within the functional expansion of an abstraction or within the expansion of an invariant abstraction. The former constraint assures the isomorphism of the program and functional trees and simplifies the user understanding of the content of the work area (it will never contain too much intertwined program and function texts). One can ask whether it makes sense to relax any of these constraints.

Some automatic program transformations in the limited sense of [TEIT 81] or, more generally, of [BAUE 79] can also be introduced in DUAL, but they cannot in some inefficiency in its storage management since the presence of the reduce command requires the invertibility of each transformation, implicitly causing storage of a double version of a transformed node in those cases where such inverted transformation cannot easily be specified.

9. Conclusion

DUAL has not been conceived as an advanced experimental tool to be used in restricted “toy” environments. DUAL objectives are more ambitious. DUAL is designed to be used in real life software factories or within large organizations as a tool to achieve higher documentation standards, to encourage hierarchical program development, to make program maintenance easier. We believe that the most formidable task to be faced in order to succeed in applying systematic methodologies for program development is the enterprise of retaining a consistent fraction of designers and programmers. DUAL can be used to achieve this goal too.

The programmer is greatly encouraged and to some extent even forced to integrate his programs with the associated documentation and to keep both updated in a consistent mode. In fact, he is only required to make the logical association of the comments with the program text: it is the computer which is in charge of the routine task of maintaining the association and of presenting the material at user request.

As a final, perhaps minor, remark let us point out that, since all source programs are stored in the form of intermediate syntax trees and many editing sessions require only the editing of the text of a single abstraction, the delay between program editing and program execution is greatly reduced. It accounts to a small fraction of lexical and syntax analysis (of the order of 10%, say, if the programmer has introduced a reasonable structure) and the execution of code generation phase, which, in our compiler, takes less than 40% of the total compiling time. Obviously, DUAL requires more computer resources than a traditional text editor but the total trade-off seems to be still positive for the average user especially now that the personal computers are booming.

Acknowledgements: Various persons have contributed to the design and implementation of this system; in particular we thank O. Ghio, L. Mascherpa, and we acknowledge the fundamental contribution of G. Capella.

References

ADA 80 Reference Manual for the ADA Programming Language, Department of Defense (July 80).


BAUE 76 F.L. Bauer “Programming as an evolutionary process” Proc. 2nd Int. Conf. on Software Engineering, San Francisco 76, pp. 223-234.


CHEA 81 T.E. Chretian Jr., G.H. Holloway, J.A. Townley “Program refinement by transformation” Proc. 5th Int. Conf. on Software Engineering, San Diego 81, pp. 430-437.

DERE 81 A. Demers, T. Reps, T. Teitelbaum “Incremental evaluation of attribute grammars with application to syntax-directed editors” Proc. 8th Symp. on Principles of Programming Languages, Williamsburg 81.


DONZ 80 V. Donzeau - Gouge, G. Haet, G. Kahn, B. Land “Programming Environments based on Structured Editors: The MENTOR experience” presented at the Workshop on Programming Environments, Ridgefield, CT, June 1980.


GRIE 73 D. Gries “Describing an algorithm by Hopcroft”


LEWI 81 J. Lewis "Beyond ALBE/P: language neutral form" Proc. 5th Int. Conf. on Software Engineering, San Diego 81, pp. 422-429.


PETR 76 L. Petrone, F. Salini "Metodologie e strumenti per lo sviluppo del software di base". Congresso AIACA, 1976, Milano.


WIRT 71 N. Wirth "Program development by stepwise refinement" BACM 14, Apr. 71, pp. 221-227.