

Application and User Interface Migration from BASIC to Visual C++

G. Antoniol (*), R. Fiutem (*), E. Merlo (**) and P. Tonella (*)

(*) IRST-Istituto per la Ricerca Scientifica e Tecnologica
I-38050 Povo (Trento), Italy

(**) Dep. Electrical and Computer Engineering Ecole Polytechnique,
C.P. 6079, Succ. Centre Ville, Montreal, Quebec, Canada

Abstract

In this paper, an approach to reengineer BASIC PC legacy code into modern graphical systems is proposed. BASIC has historically been one of the first languages available on PCs. Based on it, small or medium size companies have developed throughout the time systems that represent valuable company assets to be preserved.

Our goal is the automatic migration from the BASIC character oriented user interface to a graphical environment which includes a GUI builder, and compiles event driven C/C++ code. For this purpose a conceptual representation in terms of abstract graphical objects and callbacks has been inferred from the original code, and a translator from BASIC to C has been developed. Moreover the GUI builder internal representation has been generated, so that the user interface can be interactively fine-tuned by the programmer.

We will present and discuss BASIC peculiarities, with preliminary results on code translation. For the explanation of our approach to user interface migration an example will be used throughout the text.

1 Introduction

The PC revolution has greatly increased software and hardware market: as a consequence, in order to maintain their market and to extend systems lifespan, developers have to update their software to the state of the art. This fact could result in the following dilemma: to throw away legacy code and redesign new software, or to migrate to a new system.

In this context, the problem of legacy-systems reengineering and evolution has been recently recognized and studied by several researchers. A significative collection of reengineering papers can be found in [2]. Studies have quite often involved mainframe applications written in commercial languages such as

COBOL and FORTRAN [12], [8], [4], or proprietary languages [5]. Additional reengineering research approaches are presented in [3]. A topic which has been somehow neglected is the reengineering of PC based legacy systems. These kinds of systems can be found mainly in small or medium size companies. Several of these systems are special-purpose software that was originally designed to run on PCs. Although these systems are definitely smaller in size than the mainframe based applications, for the companies which develop them, they represent valuable company assets to be preserved. The availability of tools and graphic systems on today PCs motivates the migration of these legacy PC systems to modern PC languages and graphical environments. BASIC has historically been one of the first languages available on PCs. This paper proposes an approach to reengineer BASIC PC legacy code into modern graphical systems. The problem has been decomposed into the problem of language translation and that of User Interface (UI) migration.

UIs in BASIC are character oriented. Migrating these UIs to graphical, event-driven UIs implies a major shift in the programming paradigm used: there is a fundamental difference in the way control is handled. In the character oriented paradigm, the application holds control over interaction: it presents information to the user and decides when the user has to provide input. On the other hand, graphical user interfaces are event-driven: that is, the application no longer holds control over interaction but, at any time, it must be prepared to handle user input.

Thus two main problems arise: given the character UI, a representation in terms of abstract graphical objects (e.g menus, dialog boxes, scrollbars) and handlers for the events generated by those objects, have to be inferred from the original code. Then, graphical objects and handlers have to be instantiated to automat-

ically populate the target GUI builder (Visual C++ in our case) internal representation. This information can be compiled into a running UI or inspected and modified interactively with the GUI builder to modify the interface.

Research on UI reengineering has been done at Georgia Institute of Technology where UI migration from PCs to workstations has been investigated [11], at EDS Research [13] where large minicomputer COBOL applications together with their UI have been ported to COBOL/CICS on an IBM mainframe, and at Ecole Polytechnique and Centre de Recherche Informatique de Montreal [10] where an approach to convert character oriented COBOL/CICS UI into PC based graphical ones has been developed.

This paper is organized as follows: Section 2 presents the problem and application description; Section 3 describes our UI reengineering model; Section 4 presents a short description of the BASIC language and translation issues; Section 5 analyzes in detail each phase of the UI migration approach; Section 6 describes the experimental context and reports preliminary results of the proposed approach, while Section 7 reports conclusions and further research issues.

2 Application description

The BASIC system studied in this paper was supplied by a company in the medical sector, which, in the last ten years, developed a system to acquire, process, store, and retrieve Electro EncephaloGraphic (EEG) data on PC like machines under Microsoft¹ DOS operating system. The code is fairly unstructured and is written in Borland Turbo Basic and assembler. No version control was used, no coding standards were set, and no organic and exhaustive documentation is now available. Assembly coding was used mainly to increase the speed of critical modules or to support system functionalities connected with dedicated hardware. UI is character oriented and some graphic facilities are obtained by assembly code and direct access to graphical devices.

Four modules from the system (Table 1), which are representative of the overall system and include a significant number of UI Relevant Statements (UIRS), i.e., selections by function keys and dialogues, were selected to test the reengineering approach and the automatic porting under Windows. Assembler code

¹Microsoft, Microsoft DOS, MS-DOS are registered trademark and Visual C++, Windows are trademark of Microsoft Corporation; Turbo Basic is a registered trademark of Borland International, Inc.

was treated as a black-box: in fact assembler routines are called from the BASIC environment as commands started by system calls.

	Turbo Basic (Kloc)		Assembler (Kloc)	
Modules Chosen	22 (46)		15 (80)	
	4 (4)			
Module	Mod 1	Mod 2	Mod 3	Mod 4
loc	651	1399	467	1238
loc [†]	1318	2798	797	2819
goto	41	200	9	145
gosub	16	28	12	88
label	16	120	8	23
proc	10	10	6	32
sub	2	3	3	5
funct	0	1	0	0
comment	143	261	103	268
UIRS	10	12	1	12

Table 1: Code information and system size in Kilo lines of codes (Kloc).

Table 1 shows some macroscopic features of the programming style used by the developers in the selected modules. The high GOTOs and GOSUBs number (about 10 % of code lines) underlines not only a migration problem but also the necessity to improve code quality.

3 The Reengineering Process

Fig. 1 represents the entire reengineering process: UI migration regards most of the phases while language translation is embedded in the *Functional Code Extraction and Translation* phase.

The process starts from the Abstract Syntax Tree (AST) obtained by parsing the program. The AST is visited to identify certain interaction patterns or *interaction clichés*: clichés are obtained by a preliminary analysis of the application source code searching the recurrent interaction structures. For each of the identified clichés a corresponding Abstract User Interface (AUI) object is instantiated. AUI object classes give a conceptual description of the interaction techniques commonly used in GUIs (for example, menu selection, buttons, dialog boxes, fill-in forms).

The AST information is also used to build Control Flow Graphs (CFGs), used in subsequent phases. Using such CFGs it is possible to identify a hierarchy among the AUI objects (for instance, in the case of menus, which is the application *main menu* and what are its submenus). The output of this phase is a Visual

[†]Single line colon separated instructions are split into multiple lines.

ence of recursive GOSUBs is automatically solved by a CALL to the procedure being built. This approach, shown in Fig. 2, can be thought as a simple application of partial flow analysis in the framework of the CFG reachability problem. The importance of partial flow analysis in software maintenance and more sophisticated applications have recently been presented in [7].

PROCEDURE BUILD-PROC(name)

```

tree ← mk-node(procedure)
tree.ASTpointer ← *first-label-in-flow-graph*
tree.body ← {}
while (empty-body(tree) ≠ {})
  for (node in empty-body(tree))
    i ← 1
    statement ← statement-sequence(node.ASTpointer)
    for (j ← first(statement) to last(statement))
      if (check-insert(statement(j)))
        node.stmt(i) ← mk-node(statement(j))
        if (is-defined(body(statement(j))))
          node.stmt(i).body ← {}
          node.stmt(i).ASTpointer ←
            body(statement(j))
        i ← i + 1
insert-statement(tree.stmt, 1, mk-node(goto name))
tree

```

Figure 2: Algorithm for procedure AST construction

Algorithm of Fig. 2 collects the nodes of the AST fragment that are inserted in the BASIC procedure replacing the original subroutine. The initial GOSUB name statement becomes a procedure call of the kind: CALL sub_name. Later this procedure will be converted into a C function.

Before entering the main loop, the body of the procedure is set to an empty sequence, and a pointer to the *first-label-in-flow-graph* is added. The *first-label-in-flow-graph* is not necessarily equal to name, because some GOTOs in the subroutine can reference previous labels. For this reason the last but one operation of the algorithm is the insertion of the instruction GOTO name as the first instruction encountered in the procedure, allowing to jump to label name even when other labels have been put before it. The algorithm iterates until nodes with an empty body no longer exist in the tree. For each node with empty body the corresponding statement sequence in the original AST is determined, and all the instructions in this sequence are checked for insertion in the new AST of the procedure being built. The invoked function check-insert detects the presence of the current statement in the CFG previously computed for this subroutine, and returns True or False correspondingly. Fig. 9 and 10 show an example of GOSUB translation.

5 UI migration

In this Section, the various steps implementing UI migration, illustrated in Fig. 1, are described in detail, with reference to the Turbo Basic sample program illustrated in Fig. 3. Before doing this, the UI constructs available in the source (Turbo Basic/DOS) and target (Visual C++/Windows) environments are briefly summarized.

Example program.

It consists of a main menu with three choices. The first leads to a second menu with a dummy computation entry (GO) and the return to the main menu. The second simply waits for a user input to go on. The third exits the program.

```

1 main:
2 locate 22,1:
3 print" MENU CONFIRM EXIT"
4 main1:
5 a$ = inkey$:if len(a$) <> 2 then main1
6 a% = asc(right$(a$,1))
7 select case a%
8 case 59: goto menu
9 case 61: goto confirm
10 case 68: goto exit1
11 case else: goto main1
12 end select
13 menu:
14 locate 22,1:
15 print" GO MENU "
16 menu1:
17 a$ = inkey$:if len(a$) <> 2 then menu1
18 a% = asc(right$(a$,1))
19 select case a%
20 case 59: y%=2:goto C
21 case 68: goto main
22 case else: goto menu1
23 end select
24 C:
25 goto main
26 confirm:
27 print "Strike a key to go on ..."
28 confirm1:
29 a$=inkey$:if a$="" then goto confirm1
30 goto C
31 exit1:
32 print "Exiting ..."
33 end

```

Figure 3: Turbo Basic example program.

5.1 Source vs. Target User Interface Environments

Turbo Basic UI allows formatted output instructions (PRINT) to print text at the current cursor position. The cursor position can be directly read and set (CRSLIN, POS, LOCATE). Input instructions include functions to read characters from the keyboard (INKEY\$) as well as formatted input (INPUT).

Graphical output instructions, operating at a pixel level, are also available, thus it is possible to exploit the full screen capabilities.

Microsoft Visual C++ is an environment for building Windows-based applications. It can be classified in the category of GUI builders, in that it allows a programmer to build much part of an application interactively. It incorporates high-level C++ application framework classes with integrated Windows-hosted development tools, thus the development of an application can be viewed as subclassing general application templates supplied by the environment.

5.2 User Interaction Loci Identification and Classification

In the source application code, User Interaction Loci (UIL) can be easily identified by searching the occurrences of the Turbo Basic input instruction `INKEY$` (this is the only input instruction used in the application). This instruction reads one character from the keyboard: it can read both normal ASCII codes and extended codes, such as for example function or arrow keys.

By a preliminary analysis of the Turbo Basic source code we have identified the following interaction *clichés*: selection among several alternatives, confirmation dialog, data entry dialog.

In Fig. 4 the pseudo-Turbo Basic code relative to each cliché is illustrated.

This cliché matching approach works well because the number of Interaction Clichés (ICs) identified within the application is small.

```

REM A Selection Example
label:
  a$=inkey$
  if <a$ is not a function key> then goto label
  a% = ascii-to-int(a$)
  select case a%
    case <val1>:<action1>
    ...
    case <valN>:<actionN>
  end select

REM A Confirmation Example
label:
  a$ = inkey$
  if < a$ equals null string > then goto label

REM A Data Entry Example
num-input-chars = 0
label:
  a$ = inkey$
  if < a$ equals null string > then goto label
  input-field(num-input-chars) = a$
  incr num-input-chars
  if(num-input-chars < MAX) then goto label

```

Figure 4: *Example of interaction clichés.*

The cliché identification algorithm is very simple

and is implemented as a visit of each module's subtree within the general AST, searching for instances of `INKEY$` statements. Then, each of the user input requests is classified according to one of the clichés and inserted in the corresponding set. The classification is based on structural and flow information obtainable from the AST and exploits some heuristics at the source code level. Flow information extracted is stored in the User Interface Control Flow Graph (UICFG), a flow-graph representing the control flow of the program through each input statement. This flow graph is obtained using partial control flow analysis: in this case, the flow graph starts at an occurrence of an `INKEY$` and the criterion to stop control flow propagation is the occurrence of another `INKEY$`. The graph nodes consist of AUI object instances: each node also maintains, as an attribute, a pointer to the AST corresponding input statement.

The algorithm, whose details can be found in [15], starts from an input request and follows the flow of execution of the program until the input request has been classified.

The sample program of Fig. 3 contains three UIL: the first, at line 6, is part of the main selection construct, cliché#1, and contains another cliché#1 as a submenu, at line 19. Input at line 32 reveals a cliché#2.

5.3 Abstract Interaction Object Definition

The next step in the process is the generation of the GUI objects implementing the user interactions identified and classified in the previous phase.

To do this, an intermediate representation called AUI object level has been defined. This level holds a common conceptual representation of UI objects. It has been introduced to gain generality in the description of UI objects and to guarantee portability with respect to the target GUI system. Although a specification language for the reengineering of UI similar to that described in [9] could have been used, the translation from BASIC to C does not require the full spectrum covered by such a language. A data structure storing AUI object structural relations and callback AST fragments was sufficient for our purposes.

The process of AUI objects generation can be seen as a front-end that results independent on the final graphical environment and changing it requires only changing the back-end implementing GUI objects generation from AUI objects.

AUI object classes are presented in greater detail in [15]. Here we briefly describe the object classes in

the AUI language that correspond to the ICs found in the application code.

Each object class has a *name* attribute that identifies it among the others.

AUI menu objects have *parent* and *actions* attributes. The *parent* attribute deals with the menu hierarchy: parent-child relations can be computed using UICFGs. The *actions* attribute is a sequence (ordered collection) of couples, representing the mapping between a user menu choice, *item*, and the corresponding *action*. An *action* can be either a special action, called **POPUP**, which signals to popup another menu, or it can be an executable action, called **EXEC**. **EXEC** actions specify which portion of the functional code associated to that menu object is to be executed when that *item* is selected.

The other two kind of objects are similar, in that they are subclasses of the *dialog* class. The *confirm dialog* object, besides the *name* attribute, has only one attribute, the *message* attribute, that holds the information to be displayed to the user. The *data input dialog* has another attribute more, the *data field* attribute, which is used to read user supplied data during the interaction.

5.4 AUI Instantiation

The AUI object instantiation algorithm, whose details can be found in [15], uses the information contained in the input request sets created in the previous phase. AUI dialog objects are created in a very simple way, using default messages and default actions (associated to OK and Cancel dialog buttons).

To extract the information to build AUI menu objects, the algorithm exploits some application-dependent knowledge about the screen organization. It has been observed that, as regards output, the application screen can be divided into four zones. The top and the bottom zones remain constant along the different screens shown. The top zone contains application name and the logo of the producing company, while the bottom zone contains the function keys graphical representation (from F1 to F10). Below the top zone there is the work-area, that can vary with the different functionality selected. Between the work-area and the bottom zone, there is a single screen row, that holds the meaning of the function keys in a certain application state. The analysis of this row allows the extraction of the information to associate to each function key, in order to obtain the correct action for a selection.

Then, the *actions* field of menu objects are filled, by analyzing the **SELECT** statement associated to the

```
#AUI-MENU-OBJECT#-1
<NAME> = 'AUIMENU-1
<PARENT> = undefined
<ACTIONS> = (
  <"MENU", POPUP #AUI-MENU-OBJECT#-2>,
  <"CONFIRM", <'LABEL-F3, #BRANCH-SEQ#-1>>,
  <"EXIT", <'LABEL-F10, #BRANCH-SEQ#-2>>)

#AUI-MENU-OBJECT#-2
<NAME> = 'AUIMENU-2
<PARENT> = #AUI-MENU-OBJECT#-1
<ACTIONS> = (
  <"GO", <'LABEL-F1, #BRANCH-SEQ#-3>>)

#BRANCH-SEQ#-1 = ( goto confirm )
#BRANCH-SEQ#-2 = ( goto exit )
#BRANCH-SEQ#-3 = ( y%=2, goto C )

#AUI-CONFIRM-DIALOG-OBJECT#-1
<NAME> = 'AUIDIALOG-1
<MESSAGE> = "Strike a key to go on ..."
```

Figure 5: AUI objects for the sample program.

correspondent cliché. In order to decide between associating an **EXEC** or a **POPUP** action to a branch of the **SELECT**, the branch is followed in the UICFG: a successive cliché #1 leads to a **POPUP** menu otherwise an **EXEC** action is added.

The AUI objects generated by the algorithm for the sample program are shown in Fig. 5. The syntax < . . . , . . . , . . . > represent tuples and the object class **BRANCH-SEQ** represent a sequence of statements.

5.5 Functional Code Extraction and Translation

In this phase, the functional code to be associated to an AUI object is extracted. Since GUIs are event-driven, it is necessary to identify the sections of the application source code that represent the functional, non interactive sections to be associated to graphical objects and triggered by the events delivered by the window system.

To accomplish this task, **INKEY\$** instructions at the point of separation between areas of code pertaining to different AUI menu objects have to be taken into consideration. Hence, two successive **INKEY\$**s identify a potential callback. Partitioning the code in such a way, together with the extraction of the code implementing the user interface objects, allows to accomplish the paradigm shift needed to migrate from application driven control to user driven control.

To extract the code for a callback we need information contained in the partial UICFG. The construction of the callback proceeds like the case of the Turbo Basic **GOSUB** control statement translation. In fact, all

the paths that lead to another input request along the graph must be added to the callback.

Once identified, the code forming a callback is translated from Turbo Basic to C. In this phase all the Turbo Basic statements that regard the creation of UI objects are ignored because they have already been considered when creating AUI objects. They are thus eliminated from the resulting target code.

```

AUIMENU_1_CALLBACK(int selected_item)
{
    switch(selected_item)
    {
        case CONFIRM:
            goto LABEL_F3;
            break;
        case EXIT:
            goto LABEL_F10;
            break;
    };
    LABEL_F3:
        POPUP_DIALOG(AUIDIALOG_1);
    LABEL_F10: goto exit;
    exit: exit(0);
}

```

Figure 6: C language callback resulting from functional code extraction and translation.

The callback generation process needs also some information contained in the AUI menu objects, in particular the kind of action that is associated to a user selection (**POPUP**, **EXEC**). In the **POPUP** case, the action is merely graphical and is handled by menu object builtin methods.

Fig. 6 shows an example of a C language callback resulting from functional code extraction and translation for the AUI menu object **AUIMENU-1**. The **MENU** choice has no corresponding action in the callback because submenu creation is handled by graphical builtin methods. The **POPUP_DIALOG** function is associated to **LABEL_F3** because a **CONFIRM** choice leads to a cliché #2.

5.6 Visual C++ Graphical Objects Generation

Mapping AUI objects to Visual C++ graphical objects is quite straightforward. The output of this phase is a *resource compiler* (RC) file, that will contain as many objects as the AUI objects.

RC files contain textual specifications of all the graphical resources, such as bitmaps, icons, menus and dialogs, used by an application.

AUI objects contain all the information needed to generate the corresponding RC objects. In the case of

a menu object, the *parent* attribute is used to determine if a menu is to be considered a top-level menu (parent undefined) or not. The *actions* attribute is used to build the menu structure: for each *item* that is associated to a **POPUP** action, the corresponding item in the menu is set as a **POPUP** item. Otherwise a window system message is associated to *item*. Names for items and messages are constructed starting from *item*.

In the case of AUI dialog objects, only textual entry is considered and passed to the application code. Thus a dialog has a standard format, containing a message, an active input area, and two default pushbuttons OK and Cancel. The message to be displayed is derived from the corresponding field in the AUI dialog object.

```

IDR_MAINFRAME MENU PRELOAD DISCARDABLE
BEGIN
    POPUP "&Menu"
    BEGIN
        MENUITEM "&Go", ID_GO
    END
    MENUITEM "&Confirm", ID_CONFIRM
    MENUITEM "&Exit", ID_EXIT
    END
END

IDD_CONFIRM_DIALOG_1 DIALOG DISCARDABLE 34,22,217,55
BEGIN
    LTEXT "Strike a key to go on ..."
    DEFPUSHBUTTON "OK",IDOK,176,6,32,14
    END

```

Figure 7: Extract of the RC file for the sample program.

In Fig. 7 an extract of the RC file containing the Visual C++ graphical objects derived from the AUI objects of Fig. 5 is shown. Two graphical objects are created from three AUI objects: this is so because **AUIMENU-1** and **AUIMENU-2** are merged in one menu structure, named **IDR_MAINFRAME**. **AUIMENU-1** becomes the application menubar and **AUIMENU-2** a submenu pulldown selecting **MENU** button. The second object is a dialog box specification with a message string and an OK button, that implements a confirmation request. **ID_GO**, **ID_CONFIRM**, **ID_EXIT**, **IDOK** represent the messages associated to the events of selection.

5.7 Behavioral Information Association

This last phase completes the creation of the target application files. Our objective is to create a working application, that can be compiled and executed, but also to give the user the opportunity of using Visual

C++ as a tool to modify the results of the reengineering process. Hence it is necessary to create all the files composing an application, including those used as a support for the tools of Visual C++.

The RC file is already available from the previous phases: the remaining files are generated automatically in this phase, using template files that are filled in with C++ code, generated from the information contained in the RC file and the callback file.

The main task of this phase is merging the structural information contained in the RC file with the C language callbacks extracted from code. Up to now these files have no relationship. To associate behavior to the graphical objects in the RC file, it is necessary to link user actions on objects with functional code contained in the callback file. In the Visual C++ environment this link is made by creating message-handler functions and message map entries for each user interface object.

These message-handlers are built automatically, exploiting the information in both RC file and callback file. Non popup items of a menu must be associated to an event: this is done when the Visual C++ object is created from AUI object.

Dialog objects messages are predefined (OK and Cancel actions) thus their methods are generated automatically. The Ok method simply reads the string typed in the editable field of the dialog and then control is transferred to the associated functional code.

6 Experimental Context and Preliminary Results

The reengineering tool supporting the steps previously described is Refine[§]. It allows to build the AST of a program and to navigate inside it.

The GOSUB transformation algorithm described in Section 4 has been implemented in Refine. In particular the CFG for a subroutine is built by linking Refine objects associated to CFG nodes according to the control flow as it is determined through a traversal of the AST. A Refine function implementing the algorithm is applied to the AST of the subroutine using the CFG, giving as a result the AST of the corresponding Basic procedure. The Basic procedure is then transformed into a C function by applying the transform rules to its AST.

Results on the application of the GOSUB transform algorithm to the four chosen modules (see Table 1) are given in Fig. 8. On average the number

[§]A trademark of Reasoning Systems Inc.

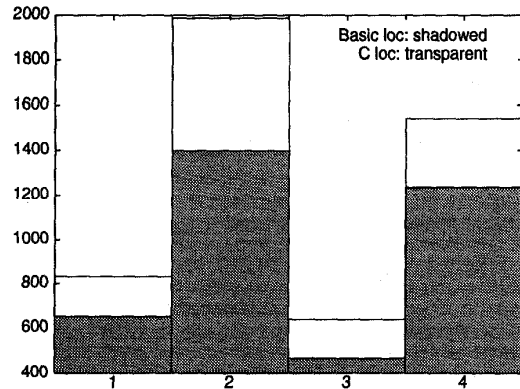


Figure 8: GOSUB Translation.

of loc's in the C modules is greater than the original loc's, because subroutine code is duplicated and put in the corresponding procedure body, but the increase is not dramatic (33%), and is compensated by a clearer call/return mechanism. The C function is also more readable in terms of single instructions. Fig. 9, 10 show an example of subroutine translation. Communication based on global variables is still used by the C function, but a list of variables modified or simply used by the function can be inserted as its argument sequence, passing the reference or the value respectively.

```

archold: * copies or deletes existing files
seek 1,position
if asc(mid$(list$,b%-1,1))=48 then compact%=1:
put$ 1,mkl$:goto archnew2 * zero reduction
open" b",2,srcfil$
if position+space+8>=lof(1) then seek 1,position-4:
put$ 1,"" else if lof(2)>space then put$ 1,mkl$
put$ 1,mkl$
goto archnew1
archnew: * copies new files
if asc(mid$(list$,b%-1,1))=48 then archnew2_
* not to be copied as "0"
open" b",2,srcfil$
archnew0:
seek 1,lof(1)
put$ 1,srcfil$
archnew1:
do while not eof(2)
get$ 2,8000,a$
put$ 1,a$
loop
close 2 archnew2:
list$=left$(list$,b%-2)
return

```

Figure 9: Example of BASIC subroutine.

It is worth noticing that one of our objectives is to give the programmers a new C code that resembles the old BASIC code, so that they can easily identify

```

void sub_archold ()
{
    /** copies or deletes existing files */
    goto archold;
    archold: seek (1, position);
    /** zero reduction */
    if (asc (mid_string (list_string, b_int - 1, 1)) == 48)
    {
        compact_int = 1;
        put_string (1, mkl_string);
        goto archnew2;
    }
    open ("b", 2, srcfil_string);
    if (position + space + 8 >= lof (1))
    {
        seek (1, position - 4);
        put_string (1, "");
    } else {
        if (lof (2) > space)
        {
            put_string (1, mkl_string);
        }
    }
    put_string (1, mkl_string);
    goto archnew1;
    archnew1:
    while (!eof (2))
    {
        get_string (2, 8000, a_string);
        put_string (1, a_string);
    }
    close (2);
    archnew2:
    list_string = left_string (list_string, b_int - 2);
    return;
}

```

Figure 10: Example of C translation.

the functional parts they have designed. It appears from Fig. 9, 10 that this has been reached by maintaining comments, identifier names, and by mapping constructs in a direct way, as far as possible.

The Refine environment has also been used for the implementation of all the AST analysis algorithms needed to perform the various steps in the UI migration process. Starting from this experimental setup a preliminary evaluation of the UI migration approach has been done. The test case is relevant with respect to user interaction aspects because it covers most of the selection masks of the program.

Table 2 shows some results obtained from the analysis of the test set. For each file included the number of UIL, and for each UIL found, the classification proposed by the algorithm are given. Some UIL result still unclassified (6% of the total): these cases are slight variants of the ICs and couldn't be classified due to the limitations of the actual UIL classification algorithm.

At present, unclassified UIL are signalled to the programmer to be handled manually.

From the set of the classified UIL, the correspon-

Module	Mod 1	Mod 2	Mod 3	Mod 4	Tot
loc	1318	2798	797	2819	7732
UIL	10	12	1	10	33
Cliché 1	0	12	1	5	18
Cliché 2	8	0	0	4	12
Cliché 3	0	0	0	1	1
Unclass.	2	0	0	0	2

Table 2: UIL classification information.

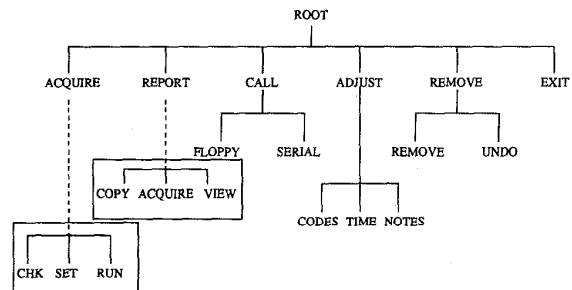


Figure 11: Extract of the menu structure in the reengineered application.

dent AUI objects have been instantiated. As regards AUI menu objects, hierarchies have been identified. It has not been possible to accommodate all menus in a single menu tree because the function of nodes in such a tree sometimes is not merely the display of a pull-down submenu, but involves a state change of the interface, switching to another mask in the work-area. Thus, a forest of menu hierarchies has resulted, in which each root becomes a menubar. An extract of the resulting hierarchy is illustrated in Fig. 11. Submenus connected with dashed lines and enclosed in rectangles are actually implemented as top-level menubars. With respect to the original hierarchy in the source application, the menu item **MENU**, which was present in each submask and whose function was to return to the previous mask, has been removed from the resulting menu objects.

7 Conclusions and Future Work

An approach to migrate applications and UI from Turbo BASIC to C has been presented. Our approach differs from [11] where the migration between different graphical environment (the original on a PC under Windows the target on a workstation under Motif) has been presented. We focused on a migration between character oriented UI and graphical UI: our approach is therefore somehow more similar to [10], but the in-

intermediate representation used is much simpler than a full UI specification language.

We have also proposed to populate automatically the target GUI builder space thus allowing designers/developers to fully exploit GUI builder modification capabilities on the generated UI.

An approach for Turbo BASIC UI clichés matching has been proposed and the sufficient set of clichés for UI migration has been identified. Given the small cardinality of the UI clichés set, the approach is feasible and effective.

The migration approach has been experimentally validated and preliminary results on some modules have been presented.

Improvements could be introduced in the GOSUB translation technique. The instructions inserted in the C function substituting the BASIC subroutine show several duplications with respect to the original BASIC main program. In order to isolate the set of statements that are not necessary any more in the main program and that could be removed, a global CFG reachability analysis could be performed [1].

Another aspect that could be improved is the presence of GOTOs in the target C code. The motivation behind this has been to preserve a programmer's familiarity with the reengineered code. Although this approach can have some advantages in the short term, it cannot be proposed as a final solution. The approach recently presented in [6] can be used as a subsequent step to remove GOTOs from the target code.

8 Acknowledgements

We are grateful to Giacomo Castellani and Paolo Gobbi for their technical support and useful suggestions about Windows and Visual C++ programming.

References

- [1] A. V. Aho, R. Sethi, J. D. Ullman, "Compilers Principles, Techniques and Tools" Addison-Wesley, 1988
- [2] R. Arnold, "Software Reengineering", *IEEE Computer Society press*, 1993
- [3] K. Bennett, "Legacy Systems: Coping with Success", *IEEE Software*, pp. 19-23, Jan 1995
- [4] P. Breuer, K. Lano, "Creating Specifications from Code: Reverse-engineering techniques", *Journal of Software Maintenance: Research and Practise*, pp. 145-162, John Wiley and Sons, 1991
- [5] E. Buss, et al., "Investigating Reverse Engineering Technologies for the CAS Program Understanding Project", *IBM Systems Journal*, Vol. 33, No. 3, 1994, pp. 477-500
- [6] A. M. Erosa, L. J. Hendren, "Taming Control Flow: A Structured Approach to Eliminating Goto Statements", *International Conference on Computer Languages*, 1994
- [7] R. Gupta, M.L.Soffa, "A Framework for Partial Data Flow Analysis", *Proceedings of the International Conference on Software Maintenance*, pp. 4-13, Victoria, 1994
- [8] L. Markosian et al., "Using an enabling technology", *Communications of the ACM*, May 1994, Vol. 37, No. 5, pp. 58-70
- [9] E. Merlo, J. F. Girard, K. Kontogiannis, P. Panangaden, R. De Mori, "Reverse Engineering of User Interfaces", *Proceedings of the First Working Conference on Reverse Engineering*, Baltimore, 1993
- [10] E. Merlo, P. Y. Gagné, J. F. Girard, K. Kontogiannis, L. Hendren, P. Panangaden, R. De Mori, "Reengineering User Interfaces", *IEEE Software*, pp. 64-73, Jan 1995
- [11] M. Moore, S. Rugaber, P. Seaver, "Knowledge-Based User Interface Migration", *Proceedings of the International Conference on Software Maintenance*, pp. 72-79, Victoria, 1994
- [12] J. Q. Ning, A. Engberts, W. Kozaczynski "Automated Support for Legacy Code Understanding", *Communications of the ACM*, May 1994, Vol. 37, No. 5, pp. 50-57
- [13] L. Van Sickle, Z. Y. Liu, M. Ballantyne, "Recovering User Interface Specifications for Porting Transaction Processing Applications", *Proc. Workshop on Program Comprehension*, July 8-9, 1993, Capri, Italy, pp. 71-76
- [14] P. Tonella, R. Fiutem, G. Antoniol, and E. Merlo "Language Translation from Basic to C", *IRST Technical Report No. 9507-05*, 1995
- [15] R. Fiutem, P. Tonella, E. Merlo and G. Antoniol, "Migrating a Basic character oriented application interface to a Visual C++ graphical user interface under Windows", *IRST Technical Report No. 9507-04*, 1995