Re-Engineering Graphical User Interfaces from their Resource Files with UsiResourcer

Óscar Sánchez Ramón[1,2], Jean Vanderdonckt[1], Jesús García Molina[2]
[1]Louvain Interaction Laboratory, Louvain School of Management (LSM) – Place des Doyens, 1
Université catholique de Louvain (UCL) – B-1348 Louvain-la-Neuve, Belgium
[2]University of Murcia, Facultad de Informática, 30100 - Campus de Espinardo – Murcia (Spain)
{oscar.sanchez, jean.vanderdonckt}@uclouvain.be – {osanchez,jmolina}@um.es

Abstract—**This paper addresses the problem of modernizing graphical user interfaces of interactive applications by re-engineering their resource files in four phases: *resource decompilation*, which extracts resource files from the executable code of an interactive application; *modeling the source user interface*, which transforms extracted resources into a resource model; *resource to user interface transformation*, which transforms the resource model into a Concrete User Interface model, and *forward engineering*, which offers two alternatives: after editing the user interface model, a new interface could be generated or recompiled into a resource to be incorporated back. The paper motivates and details this re-engineering approach by focusing on methods and algorithms implemented in *UsiResourcer*, a software tool that reverse engineers MS Windows resource files into a Concrete User Interface Model for further process. A discussion on the generalization of the approach is also provided.**

Index Terms— **User Interfaces, Modernization, Model Driven Engineering, Reverse Engineering, Reengineering.**

I. Introduction

The life duration of an interactive application is an important, yet challenging, aspect. The code we write for an interactive application is expected to run for a long time. It is not desired that it becomes outdated after a few years. A same interaction application could be also produced for a vast range of product lines (i.e., multiple versions) [1] and contexts of use (i.e., user, platform, and environment) [2] and not just one line or context. The Graphical User Interface (GUI) of an interactive application is subject to an intrinsic complexity that goes beyond merely programming algorithms [3]. GUIs are submitted to continuous changes in their development life cycle and organizations must therefore devote significant efforts to their maintenance and evolution to quickly adapt the GUIs to these changes [1]. As the technologies evolve, the requirements evolve too (e.g., additional versions [4], migration to the web [5], retargeting to other platforms [6]). The constant evolution of computing platforms and their associated Integrated Development Environments (IDEs) requires more efforts to cope with portability of interactive applications and their associated GUI. Different cases of GUI evolution may occur from the point of view of the existence of the GUI models and/or specifications and the GUI code obtained from them:

- *Specifications and/or models exist* that are of enough expressiveness to turn a legacy GUI [4] (e.g., using an old-fashioned technology like Character User Interfaces) into a modern GUI (e.g., using today's technology).
- *Specifications and/or models are no longer accessible*, but the code of the interactive application still exists. When the source code exists, transcoding techniques [7] could reverse engineer the GUI code into a new one; when the source code is lost, when no documentation (e.g., a conceptual model of the application) still exists, but the executable application is still available, a critical case appears where these techniques are no longer applicable.

Re-engineering GUIs remains an open challenge when there is a need to modernize an interactive application [8], to adapt its GUI to a new context of use [9], whether these applications are legacy or not. GUI revamping [7] requires widgets to be syntactically modified without changing the underlying functional core. Reengineering GUIs to a new computing platform [10] requires dealing with intricate issues in each stage (reverse engineering, restructuring and forward engineering).

In some IDEs such as the native applications for Windows that use the Windows API (formerly Win32 API), applications use Resource Files that define some GUI resources to be used, such as menus, dialogs, icons, or key accelerators. A *resource file* is a text file which defines resources for structural or behavioral aspects of GUIs. Resource files are compiled to machine code and are loaded into memory at runtime only when they are required in the execution, thus inducing file swapping when needed. We have tackled the reengineering of resource files-based GUIs when only compiled files are available and we have developed the *UsiResourcer* tool to support our reengineering approach. This paper presents these aspects.

In our proposal, resource files are an expressive starting point for performing GUI reverse engineering to feed a re-engineering process in four phases: (i) decompile resource files in textual format from the executable files, (ii) parse the resources defined in the resource files and instantiate a resource model reflecting the resources' contents, (iii) transform the platform-dependent resource model in a platform-independent Concrete User Interface (CUI) model by use of parameterized derivation rules, and (iv) edit this model for getting a new GUI.

The remainder of this paper is organized as follows. The next section shows that this approach is original with respect to the related work. Section 3 defines a resource file and its contents while motivating the conceptual differences between traditional GUI reverse engineering and our approach based on resource files. Section 4 explains the four phases of the re-engineering process based on resource files exemplified on a running example. Section 5 presents *UsiResourcer*, the software that supports resource recovery, illustrates some real-world cases, and also discusses how to extend the approach to various formats of resources files and its generalization. Section 6 concludes the paper by discussing existing shortcomings and presenting some future avenues.

II. Related Work

Reverse engineering [7] is "the process of analyzing an existing system to identify its components and their interrelationships and create representations of the system in another form or at a higher level of abstraction. Reverse engineering is usually undertaken in order to redesign the system for better maintainability or to produce a copy of a system without access to the design from which it was originally produced."

TABLE I. State-of-the-art in GUI Reverse Engineering.

Software	Input	Output	Techniques
Vaquita, 2001 [6,10]	HTML 4.0 site	XIML CUI model	Static analysis
WebRevenge, 2002 [13]	HTML 4.0 page	CTT Task model	Static analysis
GUIRipping 2003 [19]	GUI	CUI	Static and dynamic analyses
ReversiXML 2005 [12]	HTML 4.0 site	UsiXML CUI, AUI, user, device	Static analysis, model-to-model transformation, derivation rules
Swing2Script 2007 [16]	Java	CUI, then Ajax	Static analysis, abstract interpretation
ReverseAllUIs 2007 [14]	XHTML, VoiceXML	CUI, then AUI, then task model	Static analysis, operator merging, XLST transformation
Wrapper, 2008 [5]	GUI	SOA	Static analysis
Event handlers, 2010 [8, 24]	Any GUI	Dialog model	Dynamic analysis and model-to-model transformations
GUISurfer, 2010 [20]	Any GUI	Behavioural model	Static analysis, syntactic tree building, code slicing
UI Controls, 2011 [18]	Web application	Control representation	Static analysis of HTML, abstract interpretation of JavaScript
PureXML, 2011 [4]	Java GUI, screenshots	UsiXML CUI model	Image analysis, pattern-matching, static analysis
Prefab, 2011 [17]	Any GUI	GUI own representation	Image analysis, pattern matching, pixel-based methods
Muhairat 2011 [23]	Java GUI	Domain model (UML class diagram	Static and dynamic analyses, capturing into a Petri net, normalization and translation
ReGUI, 2011 [25]	Windows GUI	Spec# CUI model	Dynamic analysis of MS Windows code and tree building
Maria-Reverse, 2012 [15]	HTML 5 site	MariaXML CUI model	Static analysis of HTML code and CSS, encapsulation of JavaScript
***UsiResourcer,* 2013**	**GUI resource file**	**UsiXML CUI model**	**Static analysis, binary decompilation, parameterized derivation rules, model-to-model transformation**

Table 1 compares related works by chronological order according to three criteria: the "Input" column specifies the GUI type, the "Techniques" column lists all techniques used to support the reverse engineering, and the "Output" column characterizes its results by mentioning the level of abstraction according to the Cameleon Reference Framework (CRF) [2]: task & domain models, Abstract User Interface (AUI) model that is independent of any interaction modality, or Concrete User Interface (CUI) model that is for a given interaction modality (e.g., graphical), but independently of any implementation.

All existing approaches transform some GUI source code because it is available (e.g., from the past development process [4,22]) or accessible (e.g., HTML and JavaScript codes downloaded from the web [10,12,13,15]) into some CUI model.

Vaquita [10] statically analyses pages of a web site and turn them into a CUI model expressed in eXtensible Interface Markup Language (XIML – www. ximl.org), which serves for retargeting [6] to other platforms or apply a complete reengineering process [11]. The same process is used in [9] for reverse engineer a GUI also in XIML. *ReversiXML* [12], the successor of V*aquita* [6,10,11], does the same job into a CUI model, a user model, and a platform model expressed jointly expressed in User Interface eXtensible Markup Language (UsiXML V1.8.0 - www.usixml.org). The CUI Model, obtained by derivation rules, is then abstracted into an AUI Model by model-to-model transformations. Legacy GUIs [6], by static analysis [21], are integrated into a Service-Oriented Architecture (SOA).

WebRevenge [13] directly transforms a HTML web page into a CTT-compliant task model that could then serve for launching a forward engineering approach or any other purpose. Skipping intermediate levels (from final GUI to task model) is seducing, but challenging since many design intentions could be hidden at different levels of abstraction. For this reason, *ReverseAllUIs* [14] jointly reverse engineers a XHTML web page and a VoiceXML document into respective CUI models that are merged in an AUI model, from which a task model is obtained. *ReverseMaria* [15], the successor of *WebRevenge* [13], automatically generates a CUI model expressed in MariaXML from any web page (local or remote) expressed in HTML, along with CSS and JavaScript. In [16], Java GUI code is also reverse engineered into an Ajax script, which remains at the Final User Interface (FUI) level [2].

Prefab [17] applies image and pattern-matching filters and algorithms to graphically detect which interaction style or technique is active on GUI widgets, to record it, and to reproduce it. This system is independent from any computing platform since its algorithm works on raw pixel-based screen definitions. Prefab also detects any hierarchy of widgets and their contents to perform customization. Sometimes, only controls are subject to reverse engineering as in FireCrow [18], with or without event handlers [8], as opposed to entire web pages or entire GUI containers, thus providing some degree of flexibility.

GUIRipping [19] automatically recreates a Concrete User Interface model of a GUI in order to submit it to model-checking or test case generation. Although the primary goal of this software is to automate the software testing process, the resulting model could also be used for re-engineering.

GUISurfer [20] automatically extracts a behavior model from GUI source code by static analysis in order to test it, to maintain it, and to make it evolving with respect to new requirements. For this purpose, a language-dependent parser transforms the GUI source code into an abstract syntax tree, which then gives rise to a CUI model by code slicing. Then, language-independent software acts on the resulting models.

Interaction traces [21] detect the user behavior and suggest GUI optimizations according to the most frequently used interaction traces. They are also reverse engineered from C++ source code [22]. A software suite presented in [4] enables designers or end users to draw GUI prototypes in *GUILayout++* (with both low and high levels of fidelity) that are automatically abstracted into an AUI model. In this suite, *PureXML* supports loading GUI screenshots from a GUI design or an existing interactive application, and creating prototypes for those screenshots. These prototypes then initiate a CUI model for multiple purposes.

Muhairat *et al*. [23] combine static and dynamic analyses in order to reverse engineer a domain model (expressed as a UML Class Diagram) from a Java GUI in three steps: capturing the static and the dynamic aspects of the GUI into a Petri net with transitions expressing potential navigation schemes, normalizing the transitions, and translating it into a Class Diagram.

ReGUI [25] performs some dynamic analysis of a Windows GUI code in order to build a CUI model expressed in the Spec# description language that covers both the presentation and the navigation between the various windows, menus, and controls.

UsiResourcer, listed in the bottom line of Table 1, is different according to the three criteria considered:

- For the "Input", *UsiResourcer* is the **only** software that performs GUI reverse engineering from a resource file coming from the executable code of the application, and not its source code, thus requiring adequate handling of this format. Conceptual differences between a GUI expressed by its source code (e.g., as in HTML) and a resource file are significant: more controls are described (e.g., menu bars, pull-down menus, error and information messages, icons) with more static attributes (e.g., shortcuts, mnemonic keys, multi-resolution, multi-languages aspects, style effects) and dynamic attributes (e.g., activation/deactivation, normal vs default state, style behavior).
- For the "Output", *UsiResourcer* relies on UsiXML, a fifth-generation User Interface Description Language (UIDL) that not only captures various models, but also the traceability of the relationships between these models, all according to the same meta-models.
- For the "Techniques", *UsiResourcer* combines low-level techniques (e.g., static analysis, binary decompilation) and high-level techniques (e.g., parameterized derivation rules and model-to-model transformation) to uncouple the resource file format from the reverse engineering process.

III. GUI Resource file

A *resource file* is a structured text file that contains resources which are useful to one or many GUIs of an interactive application. The resources can be icons, menus, dialog boxes, strings tables, user-defined binary data, and other types of items. Once compiled into a suitable format, a resource file can be incorporated directly into an executable file, containing both functional code and GUI resources. At run-time, the interactive application can use the resource items in the embedded file. A resource file is particularly useful for the following reasons: *development independence* (the GUI development is clearly separated from the rest of the development of the interactive application), *separation of concerns* (the GUI design could be conducted by usability experts without requiring extensive programming, while the rest is conducted by developers), *reusability* (any GUI resource could be reused from one application to another), and *consistency* (a same resource could be systematically incorporated in different applications). All modern computing platforms (e.g., Windows 7, Mac OS, Linux) use some form of resource files with their own proprietary format.

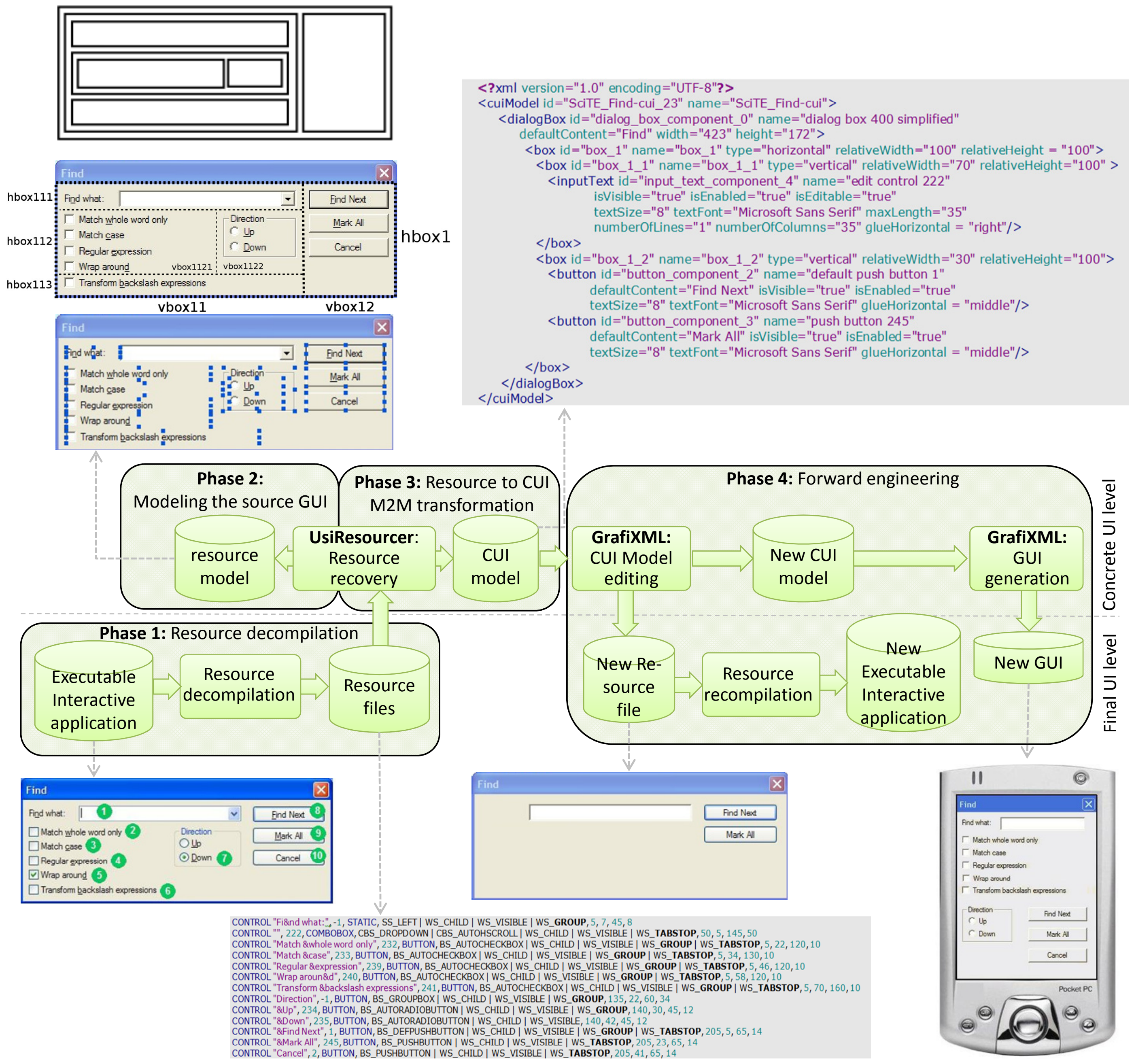
This paper focuses on Windows 7 resource files as an example throughout the re-engineering process. The approach could be equally used with other formats, as discussed later on in Section 6, where the generalization of the approach is discussed. In a Microsoft Windows 7 interactive application, resources are usually stored in the executable file (*.exe) or in a separate Dynamic Link Libraries file (*.dll). As the name suggests, these libraries are not statically linked to the executable when the application is run, but dynamically loaded into the system memory at runtime. A typical example of resource library is the *Common Dialog Library* (comdlg32.dll) that provides common dialogs used by Windows applications such as the open file and print dialogs. A W7 resource file is composed of resource-definition statements falling into three categories:

1. *Resources*: are static elements that can be used at any time, such as: strings, bitmaps, icons, cursors, accelerators, appearance and function of menus.
2. *Controls*: define the GUI widgets in terms of types, such as: check box, combo box, pushbutton, scroll bars.
3. *Statements*: assign values to control properties, such as caption, font, language, style, menus, and menu items.

IV. The Overall Re-engineering Process

The overall re-engineering process is composed of four phases (Figure 1): (i) a *resource decompilation* in order to extract designated resource files from the original executable file and to get a textual representation of these resources, (ii) *modeling the source GUI*, which transforms extracted resources into a resource model; (iii) *resource to CUI transformation*, which transforms the resource model into a Concrete User Interface (CUI) model by parameterized derivation rules; and (iv) *forward engineering*, which applies changes on the model according to any new requirement (e.g., a change of context, a new platform, an update of the GUI). Two cases occur: *resource recompilation*, which produces a new interactive application shipped with the newly obtained GUI or *generation*, which re-generates a new graphical user interface from the CUI model, for instance in a programming or markup language. The main difference between resource recompilation and generation in phase 4 is: in the former, the new GUI is transformed again in a resource file that becomes re-incorporated into the original application code (by resource compilation and re-insertion in the code), thus preserving the whole functional core; in the latter, the GUI will be reused for another application in another language. *Resource recompilation* is primarily required when the original application should be preserved while generation is preferred when there is a need to recover an existing GUI definition and make it gracefully evolve to a new version in another application, e.g., for direct reusing, for ensuring consistency, for repurposing a GUI to a new context of use.

The four phases of Figure 1 are further detailed in the next sub-sections, based on the running example of the Microsoft Windows standard dialog box for finding words in a text. This dialog box is introduced here for the purpose of understandability. More complex case studies will be discussed in Section V. Figure 1 reproduces some excerpts of the artifacts resulting from each phase: the textual representation of the resource file of the corresponding CUI model is partially reproduced in order to give the flavor of the approach.



```
<?xml version="1.0" encoding="UTF-8"?>
<cuiModel id="SciTE_Find-cui_23" name="SciTE_Find-cui">
    <dialogBox id="dialog_box_component_0" name="dialog box 400 simplified"
        defaultContent="Find" width="423" height="172">
        <box id="box_1" name="box_1" type="horizontal" relativeWidth="100" relativeHeight = "100">
          <box id="box_1_1" name="box_1_1" type="vertical" relativeWidth="70" relativeHeight="100" >
            <inputText id="input_text_component_4" name="edit control 222"
                   isVisible="true" isEnabled="true" isEditable="true"
                   textSize="8" textFont="Microsoft Sans Serif" maxLength="35"
                   numberOfLines="1" numberOfColumns="35" glueHorizontal = "right"/>
          </box>
          <box id="box_1_2" name="box_1_2" type="vertical" relativeWidth="30" relativeHeight="100">
            <button id="button_component_2" name="default push button 1"
                  defaultContent="Find Next" isVisible="true" isEnabled="true"
                  textSize="8" textFont="Microsoft Sans Serif" glueHorizontal = "middle"/>
            <button id="button_component_3" name="push button 245"
                  defaultContent="Mark All" isVisible="true" isEnabled="true"
                  textSize="8" textFont="Microsoft Sans Serif" glueHorizontal = "middle"/>
        </box>
    </dialogBox>
</cuiModel>
```

```
CONTROL "Fi&nd what:", -1, STATIC, SS_LEFT | WS_CHILD | WS_VISIBLE | WS_GROUP, 5, 7, 45, 8
CONTROL "", 222, COMBOBOX, CBS_DROPDOWN | CBS_AUTOHSCROLL | WS_CHILD | WS_VISIBLE | WS_TABSTOP, 50, 5, 145, 50
CONTROL "Match &whole word only", 232, BUTTON, BS_AUTOCHECKBOX | WS_CHILD | WS_VISIBLE | WS_GROUP | WS_TABSTOP, 5, 22, 120, 10
CONTROL "Match &case", 233, BUTTON, BS_AUTOCHECKBOX | WS_CHILD | WS_VISIBLE | WS_GROUP | WS_TABSTOP, 5, 34, 130, 10
CONTROL "Regular &expression", 239, BUTTON, BS_AUTOCHECKBOX | WS_CHILD | WS_VISIBLE | WS_GROUP | WS_TABSTOP, 5, 46, 120, 10
CONTROL "Wrap aroun&d", 240, BUTTON, BS_AUTOCHECKBOX | WS_CHILD | WS_VISIBLE | WS_GROUP | WS_TABSTOP, 5, 58, 120, 10
CONTROL "Transform &backslash expressions", 241, BUTTON, BS_AUTOCHECKBOX | WS_CHILD | WS_VISIBLE | WS_GROUP | WS_TABSTOP, 5, 70, 160, 10
CONTROL "Direction", -1, BUTTON, BS_GROUPBOX | WS_CHILD | WS_VISIBLE | WS_GROUP, 135, 22, 60, 34
CONTROL "&Up", 234, BUTTON, BS_AUTORADIOBUTTON | WS_CHILD | WS_VISIBLE | WS_GROUP, 140, 30, 45, 12
CONTROL "&Down", 235, BUTTON, BS_AUTORADIOBUTTON | WS_CHILD | WS_VISIBLE, 140, 42, 45, 12
CONTROL "&Find Next", 1, BUTTON, BS_DEFPUSHBUTTON | WS_CHILD | WS_VISIBLE | WS_GROUP | WS_TABSTOP, 205, 5, 65, 14
CONTROL "&Mark All", 245, BUTTON, BS_PUSHBUTTON | WS_CHILD | WS_VISIBLE | WS_TABSTOP, 205, 23, 65, 14
CONTROL "Cancel", 2, BUTTON, BS_PUSHBUTTON | WS_CHILD | WS_VISIBLE | WS_TABSTOP, 205, 41, 65, 14
```

FIGURE 1. THE OVERALL RE-ENGINEERING PROCESS EXEMPLIFIED ON THE "FIND" DIALOG BOX.

A. Phase 1: Resource Decompilation

When an interactive application is built, a resource compiler transforms resources from their initial format (structured text or XML format) into binary files to be linked with the application. Resource decompilation does the opposite: it unlinks and decompiles binary files into textual resource files. There are several applications that can decompile executable files in order to retrieve the resources, such as Resource Builder (http://www.resource-builder.com), Tuner (http://www.restuner.com), and Resource Hacker (http://www.angusj.com/resourcehacker). Since that the tools apply their own algorithms to detect and interpret bit patterns in the binary files and a window can be described in different ways, the resource file generated by each tool for the same window may not be the same. A comparative analysis of 6 resource decompilers was conducted on 8 case studies. Resource Hacker was observed as the one providing the most expressive, complete and accurate resource files in terms of properties, values and *recovery rate* (ratio between the controls recovered by decompilation and the total amount of controls in the GUI).

The resources such as dialogs can contain controls. Each control is attached to flags expressing constraints acting on the control: constraints inherited from the parent object (i.e., the parent window) and specific constraints that are local to the control. Such constraints affect properties such as, but not limited to: Child, Disabled, Visible, Border, TabStop, Group, VerticalScrollBar, HorizontalScrollBar. For instance, a combo box cannot use all the specific attributes of list box and edit controls: only the AutoHScroll, LowerCase, OEMConvert, UpperCase, Sort, DisableNoScroll, HasStrings, NoIntegralHeight, OwnerDrawFixed and OwnerDrawVariable attributes. Most constraints are expressed with type declaration with enumerate values. For example, a menu item can be disabled and greyed at the same time, a title bar cannot contain a question mark (ContextHelp = true) with a maximize box or a minimize box (MaximizeBox or MinimizeBox = true). Flags attached to controls can be expressed by means of a combination of textual identifiers (Figure 2) and/or a hexadecimal number that represents the bits of a flag. For example, the 19 different types of static controls are grouped in the five first bits, thus representing $2^5 = 36$ flags. Resource decompilers only extract the hexadecimal number, thus preventing it from determining appropriate static definitions of controls. For instance, a decompiler cannot differentiate whether the text of a static control is both centered (SS_CENTER flag) and right-justified (SS_RIGHT flag) in the rectangle, or it has an icon (SS_ICON flag). It is our responsibility to correctly associate this hexadecimal value to corresponding values of properties.

	4	3	2	1	0		4	3	2	1	0
SS_LEFT						SS_USERITEM		✓		✓	
SS_CENTER					✓	SS_SIMPLE		✓		✓	✓
SS_RIGHT				✓		SS_LEFTNOWORDWRAP		✓	✓		
SS_ICON				✓	✓	SS_OWNERDRAW		✓	✓		✓
SS_BLACKRECT			✓			SS_BITMAP		✓	✓	✓	
SS_GRAYRECT			✓		✓	SS_ENHMETAFILE		✓	✓	✓	✓
SS_WHITERECT			✓	✓		SS_ETCHEDHORZ	✓				
SS_BLACKFRAME			✓	✓	✓	SS_ETCHEDVERT	✓				✓
SS_GRAYFRAME		✓				SS_ETCHEDFRAME	✓			✓	
SS_WHITEFRAME		✓			✓						

FIGURE 2. FLAGS FOR CONTROLS IN WINDOWS.

Figure 1 (bottom center) contains an excerpt of the textual resource file obtained for the "Find" dialog box decompiled by Resource Hacker. In this resource file, the first field after the CONTROL keyword is the name of the control, the second field is the identifier of the control and the third field is the type of control (e.g., BUTTON or COMBOBOX). The following field represents the flags that affect the appearance and behavior of the control (e.g., WS_VISIBLE means that the control is visible and WS_TABSTOP means that the control can be navigated by means of the TAB key). The four last fields represent the X and Y coordinates of the upper left position of the control, its width and its height respectively.

B. Phase 2: Model the source GUI

We need to manipulate and query intensively the information contained in the resource files, so an approach that directly deal with the physical text files would be neither practical nor efficient. In this case, a model-driven reverse engineering process is a better option. Reverse engineering tasks can take advantage of Model-Driven Engineering (MDE) techniques. Metamodels are appropriate to represent at a high-level of abstraction the information harvested from the software artifacts, and model-to-model transformations allows the extraction process can be automated. In *UsiResourcer*, no transformation engine has been used. Rather, each transformation has been developed manually in a separate Java method. The definitions of the source metamodel (resource metamodel) and the target (CUI model) are obtained automatically by reflection of their definition into internal records.

Based on the definition of the Windows 7 resource file format, a large collection of resource files has been examined in order to come up with a metamodel for representing the GUI elements as expressed in resource files. An excerpt of the metamodel is showed in Figure 4. Only metaclasses needed to understand the approach are reported here.

In a graphical Windows-based application, dialog boxes (DIALOG) as well as graphical controls (CONTROL) are considered as windows (WINDOW) because each control is technically managed as a small window with predefined properties. A window (WINDOW) of the application has a style (GeneralStyle attribute), which can be:

- A *popup* window, which is a temporary subsidiary window.
- A *child* window, which divides a window in regions.
- An *overlapped* window, which is used as a main application window.

Dialog boxes (DIALOG) are windows which communicate with the user and to provide services that are not located in a menu. A dialog box is defined as a pop-up window (GeneralStyle= popup) containing various child control. A modeless dialog box allows the user to switch between the dialog box and the parent window, which is convenient when the dialog box should remain active. Dialogs have a thin border (Border=true), a caption bar and a potential system menu box (SystemMenu=true).

The controls (CONTROL) are contained in a dialog box and they allow performing input and output tasks. Controls are child windows (GeneralStyle=child) that many manipulate the attributes of the windows, such as a thin-line border (Border = true), the possibility that the control is navigated through the TAB key (TabStop=true), or that the element is visible but cannot be used (Disabled = true). Note that the TAB key is particular useful for recovering the ordering of fields in the navigation. For example in Figure 1 (bottom left), the windows "Find" shows this ordering based on that information.

The type of most attributes in the metamodel is Boolean because of the amount of flags that are available to modify the aspect and behavior of the graphical elements.

In order to adequately process the textual resource files, an Extended Backus-Naur Form (EBNF) abstract grammar has been defined to recognize the syntax of the resources. The bidirectional mapping between a metamodel and a grammar is a solved problem in MDE. The EBNF grammar covers each metaclass and its related attributes belonging to the metamodel of Figure 2. *UsiResourcer* therefore parses any resource file (e.g., the bottom right of Figure 1), identifies in the EBNF grammar the rules for each occurrence of a class of Figure 2, and creates an internal record. The structure of this record has been automatically generated from the EBNF grammar. Changing the ENBF automatically propagates the changes into the record structure.

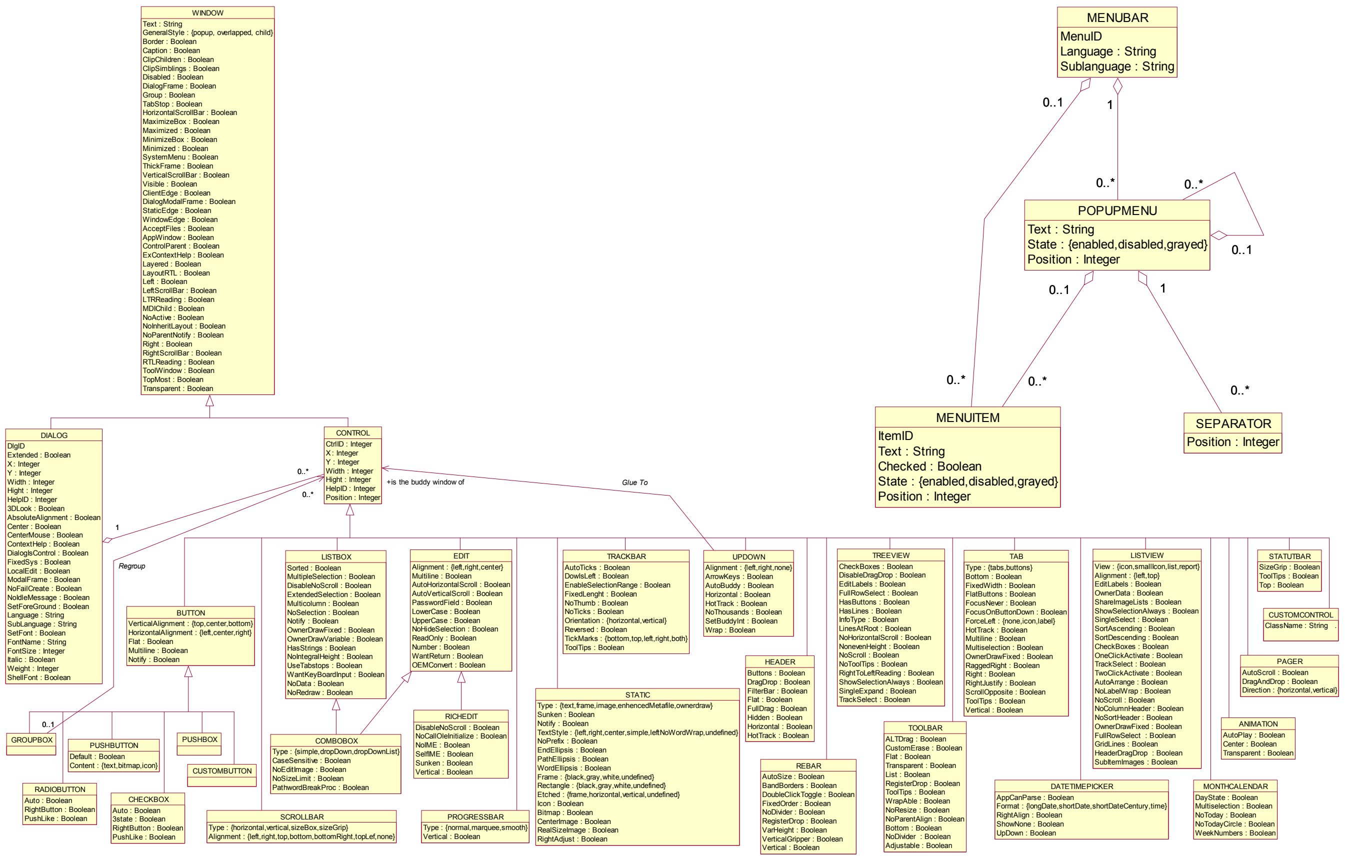


FIGURE 3. WINDOWS 7 RESOURCES METAMODEL.

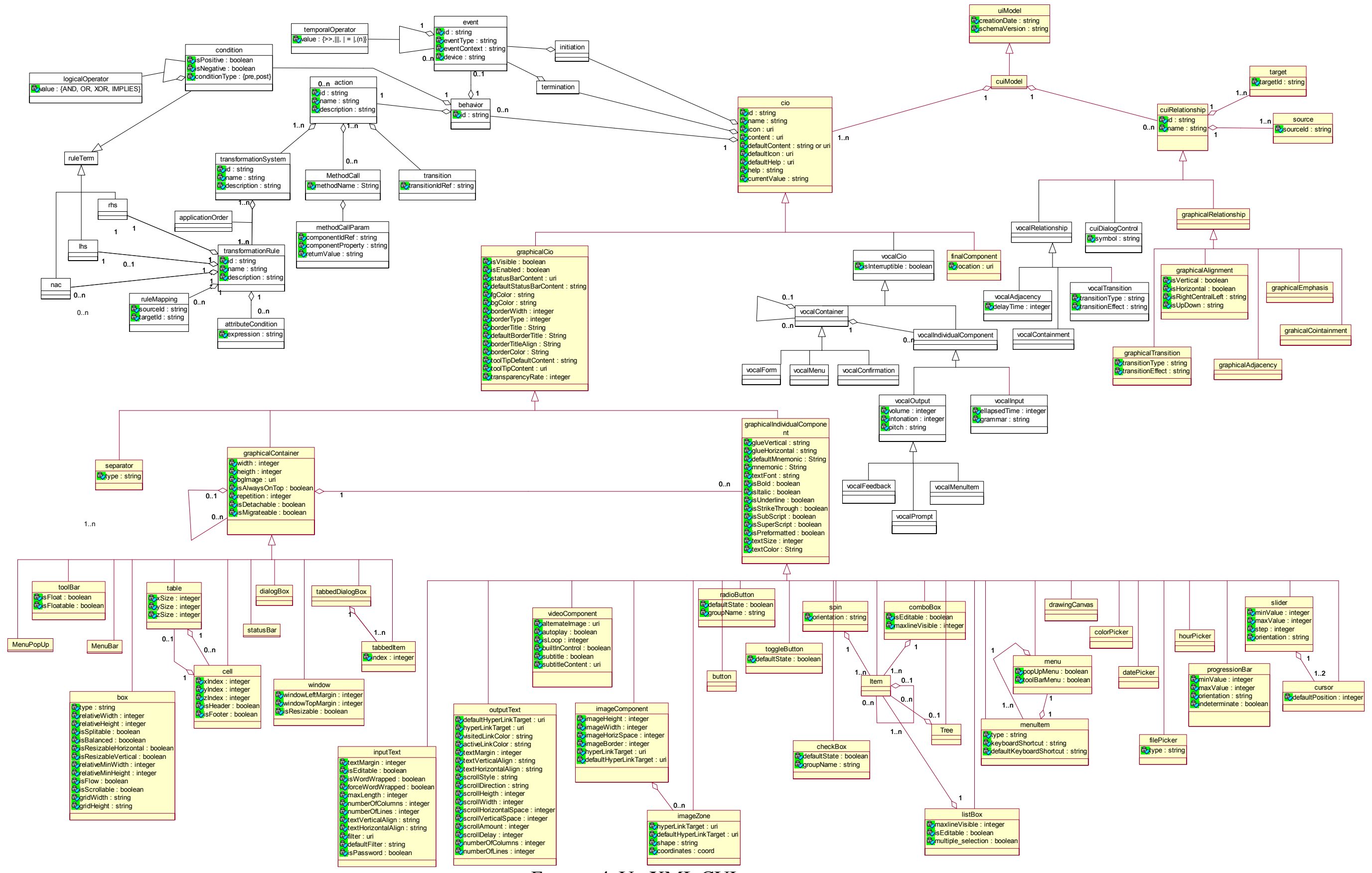


FIGURE 4. USIXML CUI METAMODEL.

The EBNF can be edited with a grammar editor. For instance, the grammar for expressing a dialog is:

```
<id> DIALOG[EX] <x>, <y>, <width>, <height> [, <helpId>]
 STYLE <style>* [EXSTYLE <ex_style>*]
 CAPTION "<text>" LANGUAGE <language>, <sublanguage>
 FONT <pointsize>, "<typeface>" [, <weight>, <italic>]
 {<control_def> }
```

where

- <S>: any entity of syntactical category S
- <S>*: suite of 0, 1 or more entities of category S, separated by the token |
- *text*, *typeface*: a character string (*text* may contain escape caracters, e.g. quote \", new line \n, tab \t, backslash \\)
- *id*: entire number or a character string
- *x*, *y*, *width*, *height*, *helpId*, *pointsize*: integer numbers
- *style*: window flag beginning with WS_ or a dialog box flag begining with DS_
- *ex_style*: extended window style flag (beginning with WS_EX_)
- *language*: primary language identifier flag beginning with LANG_
- *sublanguage*: sublanguage identifier flag beginning with SUBLANG_
- A ::= B : A is defined by B
- a b: a followed by b
- a & b: a and b (note that this meta-language symbol is different from the language token | used to combine several styles)
- val(A) : value of the attribute A, which may requires a binary or hexadecimal to integer or string conversion
- upper(A): converts the text A in upper case.
- [A]: A may occur only in a extended dialog box template (that is, a DIALOGEX resource and not a DIALOG resource)
- *control_def* : may contain any combination of control definitions (one by line)

Similarly, the grammar production rule that is used to recognize the controls is structured according to the following format:

```
CONTROL "<text>",<id>,<class>,<style>*,<x>,<y>,<width>,
<height>[,<ex_style>*,<helpID>]
```

Based on these grammars, *parameterized derivation rules*, an extension of derivation rules [12], are used to derive object instances and their properties from the resources. Table II provides an example of the parameterized derivation rules to map the controls of type PushButton into elements of the Resources metamodel. In the left column there is a list of the tokens that can be identified (according to the grammar production rule for controls that we have represented above), and the right column expresses the mapping to the metamodel elements, this is, the values that are assigned (with the <- operator) to the properties of the object. In the mapping we use two auxiliary functions.

The first function is val(), which obtains a concrete value of an alphanumeric string contained in the resource file. The second function is new() that creates an object of a specific type in the model. The mappings of the table are applied from the more concrete attributes to the general ones. It is not possible to know the type of the object just by looking at the class. Therefore, the flags must be considered. For instance, in order to identify a PushButton, we need to check that class = BUTTON and that either BS_PUSHBUTTON or BS_DEFPUSHBUTTON is specified for the control. Then, the PushButton attributes are set. After that the Button attributes are set, the Window attributes are finally specified.

TABLE II. RESOURCE MAPPINGS.

Token identified	Rules derivation for Resources metamodel
CONTROL keyword	-
<id>	CtrlId <- val(id)
<x>	X <- val(x)
<y>	Y <- val(y)
<width>	Width <- val(width)
<height>	Height <- val(height)
<helpID>	HelpID <- val(helpID)
<position>	Position <- val(position)
<class> = BUTTON	-
<style> = BS_TOP	VerticalAlignment <- top
<style> = BS_VCENTER	VerticalAlignment <- center
<style> = BS_BOTTOM	VerticalAlignment <- bottom
<style> = BS_LEFT	HorizontalAlignment <- left
<style> = BS_CENTER	HorizontalAlignment <- center
<style> = BS_RIGHT	HorizontalAlignment <- right
<style> = BS_FLAT	Flat <- true
<style> = BS_MULTILINE	Multiline <- true
<style> = BS_NOTIFY	Notify <- true
<class> = BUTTON and <style> = BS_PUSHBUTTON	new(PushButton) and Default <- false
or	
<class> = BUTTON and <style>= BS_DEFPUSHBUTTON	new(PushButton) and Default <- true
<style> = BS_TEXT	Content <- text
<style> = BS_BITMAP	Content <- bitmap
<style> = BS_ICON	Content <- icon

A notation can be used to express parameterized derivation rules for a UI specified in any language (or source model). The rules are applied on trees representing a UI: T_s is the source tree (an instance of the diagram modeling a Windows dialog box resource) and T_t is the target tree (an instance of the CUI model). The nodes of a tree T store hierarchically the elements composing the UI. Each connection (or arc) represents a containment relationship between the parent and the child. Each node of the tree represents the different elements composing the UI. Each node can possess zero or more attributes. To construct T_t, I will use the following predefined basic update operations:

- AddNode(*class*, *id*): add a new node with the identifier *id* storing an element which is an instance of *class*.
- AddAttribute(*id*, *name*, *value*): add to the node *id* the attribute *name* initialised to *value*.

- ModifyAttribute(*id*, *name*, *newname*, *newvalue*): suppress the attribute *name* of the node *id* and add the attribute *newname* with the value *newvalue*.
- AddArc (*idSource*,*idTarget*): connect the parent node *idSource* with its child node *idTarget*.

C. Phase 3: Resource to CUI transformation

In this section, we explain how a resource model can be transformed into a Concrete User Interface (CUI) model thanks to mappings between. This is a desirable step since the CUI models are platform-independent, unlike the resource model that in our case is tied to the Windows 7 platform (Figure 3). The UsiXML (User Interface eXtensible Markup Language) user interface description language was selected as the CUI definition formalism because its CUI metamodel (Figure 5 – only sections with a colored background are exploited by *UsiResourcer*) is publicly available, as well as its corresponding CUI model editor, GrafiXML [26]. Other User Interface Description Languages (UIDLs) could be equally used, such as MariaXML, as used in [15], or XIML as used in [9] without any loss of information or generality. Figure 4 reproduces the metamodel used for defining the CUI level.

An excerpt of the UsiXML CUI model for the "Find" dialog box is shown in Figure 1 (top right). The CUI model, expressed in XML, is hierarchically composed of Concrete Interaction Objects (CIOs). A CIO is any entity that the user can perceive and manipulate used for the acquisition or restitution of information. CIOs are grouped into two types: *graphical containers* (such as a window, a dialog box or a group box) and *graphical individual components* (such as an image, a check box or a progression bar). Graphical containers are arranged recursively in terms of vertical or horizontal boxes. The properties of each final element in the hierarchy, their specific and inherited attributes, are limited to describe characteristics of high common interest, independently from any GUI rendering.

D. Phase 4: Forward engineering

Once the GUI is represented by means of a platform-independent model (the CUI model), manual or semi-automatic transformations can be applied to perform changes in the original software. Two common scenarios prevail (Figure 1):

- *Resource recompilation*: the CUI model is transformed (e.g., for GUI adaptation, customization, localization or globalization) an into a new resource file that can be compiled again and re-incorporated into the executable file of the initial interactive application, thus creating a new version of this original application. For instance, some controls have been removed (Fig. 1 bottom center) and the new resource is recompiled into the original application.
- *New GUI generation*: the CUI model is transformed (e.g., by moving controls, re-aligning or reshuffling controls) in order to create a different CUI model and generate a new interface from this model. For instance, the "Find" dialog box has been reformatted to fit the constraints imposed by another screen resolution, here the one of a vertical PocketPC (Figure 1 bottom right).

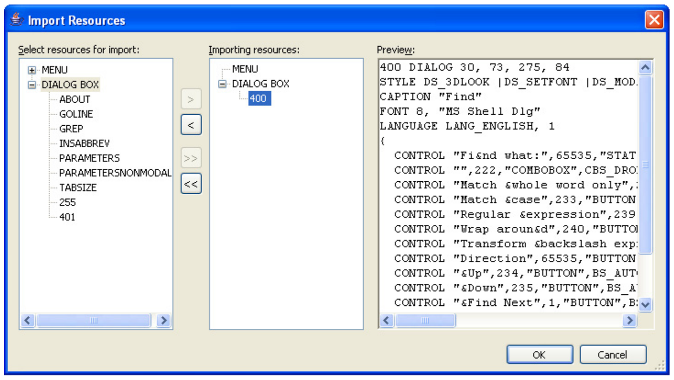
CUI models are serialized in UsiXML format by applying a generator that interprets transformation rules expressed with a template language (such as XSLT). When no transformation is applied, the CUI model is simply preserved, which is useful when a CUI element is to be reused in a consistent manner from one interactive application to another.

TABLE III. M2M TRANSFORMATION FROM WINDOWS 7 TO USIXML.

DIALOG	**dialogBox** or **window** **graphicalCio** *isVisible* = true
	box *type* = vertical
dlgID = n	**cio** *name* = n
Width = w	**graphicalContainer** *width* = w*4/xChar + 2*border width (where xChar is the average width of the dialog box font character in pixel) **box** *width* = w*4/xChar
Height = h	**graphicalContainer** *height* = h*8/yChar + title bar height + bottom border width (where yChar is the average height of the dialog box font character in pixel) **box** *height* = h*8/yChar
Text = t and *Caption* = true and t ≠ null	**cio** *defaultContent* = t
FontName = n and *SetFont* = true	**graphicalIndividualComponent** *textFont* = n
ShellFont = true and *FontName* = "MS Shell DLG"	**graphicalIndividualComponent** *textFont* = "Tahoma"
ShellFont = true and *FontName* ≠ "MS Shell DLG"	**graphicalIndividualComponent** *textFont* = "Tahoma"
FontSize = s and (*SetFont* = true or *ShellFont* = true)	**graphicalIndividualComponent** *textSize* = s
SetFont = false and *ShellFont* = false	**graphicalIndividualComponent** *textSize* = 8, *textFont* = "Tahoma"
Extended = true and (*SetFont* = true or *ShellFont* = true) and *Weight* ≥ 550	**graphicalIndividualComponent** *isBold* = true
Extended = true and (*SetFont* = true *t* or *ShellFont* = true) and *Italic* = true	**graphicalIndividualComponent** *isItalic* = true
Disabled = true	**graphicalCio** *isEnabled* = false
ThirckFrame = true	**window** *isResizable* = true
TopMost = true	**graphicalContainer** *isAlwaysOnTop* = true

V. THE REENGINEERING TOOL

The resource recovery phase outlined in Figure 1 has been implemented in *UsiResourcer*, a plug-in for the GrafiXML [26] IDE, that is capable of importing/exporting a CUI model in the required format. *UsiResourcer* consists of ±5,500 Java 5 LOC in GrafiXML which itself consists of ±110,000 Java LOC. In this GUI builder, the developer does not need to read, understand, or modify the internal representation of the CUI model, but only its visual appearance: controls can be dragged from a palette and dropped onto a working area. Selecting any elements enable editing its related properties, such as default values, colors, fonts, and location. The integration of *UsiResourcer* in GrafiXML is motivated by the following reasons: from the *usage viewpoint*, the developer does not need to manipulate the internal representation of the CUI model (e.g., in a XML-compliant format), the results of the reverse engineering are immediately available and visible in the editor; from a *development viewpoint*, the editor already contains built-in methods to generate and manage elements at the desired level of abstraction after having defined transformations rules, export in other formats is also available (e.g., XUL, Java, XHTML). The main window of *UsiResourcer* can be seen in Figure 5. The tool allows the user to select the resource file (in text format) and select the resources to be recovered.



```
400 DIALOG 30, 73, 275, 84
STYLE DS_3DLOOK |DS_SETFONT |DS_MOD.
CAPTION "Find"
FONT 8, "MS Shell Dlg"
LANGUAGE LANG_ENGLISH, 1
{
  CONTROL "Fi&nd what:",65535,"STAT
  CONTROL "",222,"COMBOBOX",CBS_DRO
  CONTROL "Match &whole word only",
  CONTROL "Match &case",233,"BUTTON
  CONTROL "Regular &expression",239
  CONTROL "Wrap aroun&d",240,"BUTTO
  CONTROL "Transform &backslash exp
  CONTROL "Direction",65535,"BUTTON
  CONTROL "&Up",234,"BUTTON",BS_AUT
  CONTROL "&Down",235,"BUTTON",BS_A
  CONTROL "&Find Next",1,"BUTTON",B
```

FIGURE 5: USIRESOURCER MAIN WINDOW.

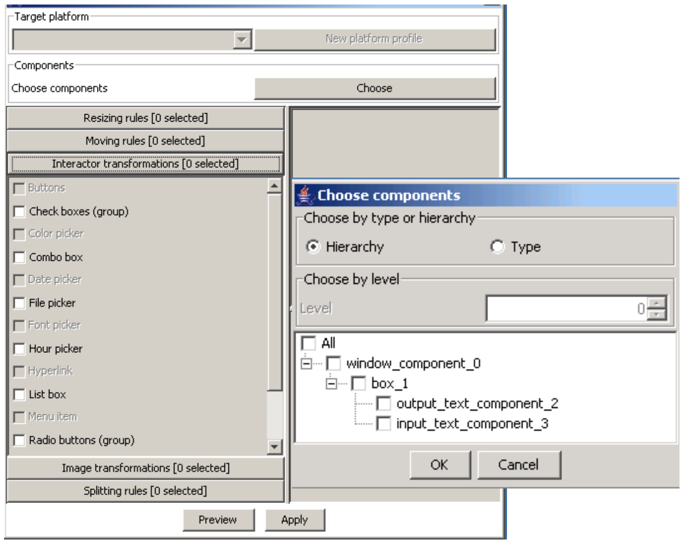


FIGURE 6. TRANSFORMATIONS APPLIED FOR GUI REFORMATTING.

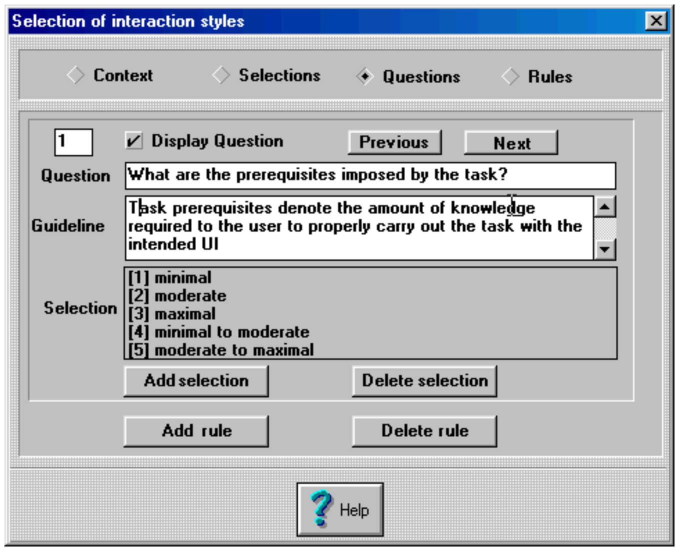
VI. CASE STUDIES AND DISCUSSION

UsiResourcer was applied in two real-world case studies:

1. *GUI reformatting*: in this case study, an information system for submitting and displaying basic data of a hotel and related touristic locations was initially developed for a 1420x1024 screen resolution of a desktop environment. As soon as touristic locations decided to buy a tactile interactive kiosk with a screen resolution of 1024x768, thus smaller than the initial screen resolution. Since only the executable code was available, it was not possible to redesign the GUI directly. This information system was a simple application containing only standard windows and controls (2 menu bars, 11 interactive forms containing a total amount of 268 controls). In order to reformat these controls, five types of transformations were applied on the CUI model resulting from the reverse engineering: resizing rules, moving rules, concrete interactor transformations, image transformations, and splitting rules (Figure 6). In this case, the recovery rate was about 95% since all controls were statically defined in standard windows.

2. *GUI localization*: one extreme situation exists when there is a need to localize an interactive application, for which only the executable code remains available. *GUIResourcer* could recover appropriate resources. Then, textual resources could be automatically translated (for instance, some on-line translation services localizes GUI strings from one native language – say French - to another one – say English, while preserving keywords, reserved terms, etc.). For this purpose, textual resources are converted into .CSV files and submitted on-line to automatic translation. Then, translated .CSV are sent and converted back into resource files to be compiled in order to obtain the new executable interactive application. *GUIResourcer* was used to localize a knowledge-based system for which no translation was available and for which only the executable code was available, because the development company went bankrupt. This case study involved: 52 windows and dialog boxes, 3 menu bars (used in different contexts) with 5 pull-down menus, for a total amount of 305 controls. *GUIResourcer* recovered 271/305 controls, giving a recovery rate of 88%. All textual resources have been automatically translated. Figure 7 reproduces a dialog ox that was localized in English (translation and revamping of the GUI).

Controls that escape from the reverse engineering process were essentially controls that were programmatically defined in the code of the application itself, therefore outside the scope of the reverse engineering process based on resource files. In this case, some other controls were not supported because they were statically defined in the resources, but only created: they were defined as invisible depending on the radio button on the top panel displayed in Figure 7. Again, their visibility was managed in the code of the application itself. Decompiling this code is too hard and imprecise in order to derive a significantly expressive behavior.



1 ✓ Display Question

Question: What are the prerequisites imposed by the task?

Guideline: Task prerequisites denote the amount of knowledge required to the user to properly carry out the task with the intended UI

Selection:
[1] minimal
[2] moderate
[3] maximal
[4] minimal to moderate
[5] moderate to maximal

FIGURE 7: GUI LOCALIZED.

VII. Shortcomings and Generalization

Based on experience using *UsiResourcer* in various case studies and trials, several shortcomings of this approach were identified that are discussed in this section. These shortcomings need to be considered when generalizing the approach to other resource files than Windows 7: this also discussed at the end of this section.

1) *Lack of mappings*

When performing the mapping from the resource mode to the CUI model, the maximum number of attributes of the CUI model must be filled in with information found in the resource file. The issue is that both metamodels can have different levels of expressiveness, i.e. not all the GUI attributes defining the appearance and behavior are covered by both metamodels (Figure 8): for example, a mnemonic of a menu item can be specified in both the resource file and the CUI model (intersection part of Figure 8), editability vs. read-only definition of a combo box can be specified in the CUI only, the visibility of the list box that is displayed permanently is not included in the CUI model.

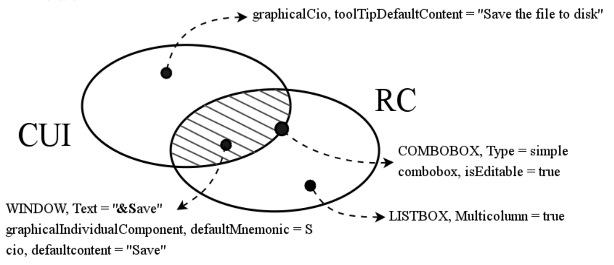

graphicalCio, toolTipDefaultContent = "Save the file to disk"
CUI
RC
COMBOBOX, Type = simple
combobox, isEditable = true
WINDOW, Text = "&Save"
graphicalIndividualComponent, defaultMnemonic = S
cio, defaultcontent = "Save"
LISTBOX, Multicolumn = true

Figure 8: Commonalities between Windows resources and CUI.

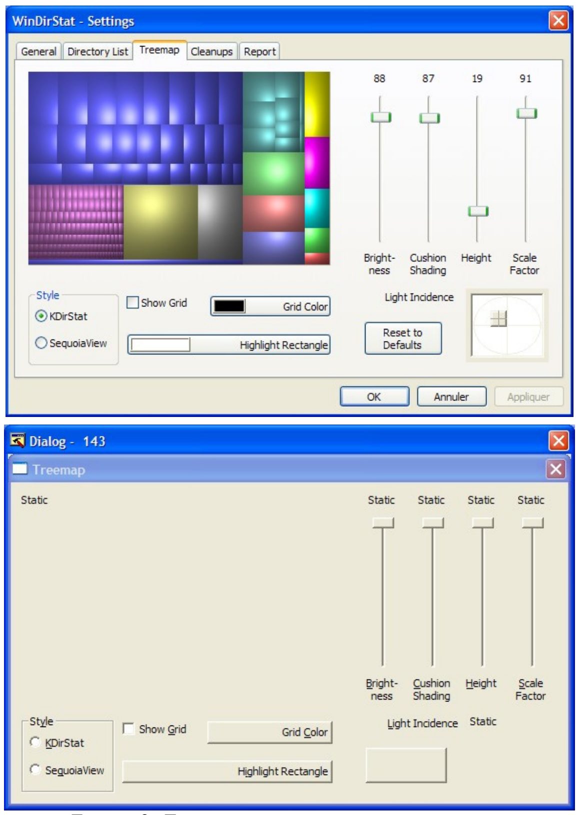


Figure 9: Example of an unsupported widget: (a) Initial window. (b) Recovered window.

2) *Unsupported widgets*

All widgets that are statically expressed through resource files are good candidates for reverse engineering according to our approach. Some widgets are outside this expression model: non-standard widgets (e.g., widgets that are not native in the Windows Software Development Toolkit), custom widgets that are typically hard-coded (e.g., with a dynamic behavior), programmatically-defined widgets (e.g., GUIs for specific tasks such as information visualization also escape from this handling. Widgets with dynamic behavior may require a dynamic analysis by observing its behavior over time and an analysis of interaction traces could be useful [22]. For example, Figure 9a reproduces a screenshot of a GUI including a color visualization tree that is lost in the translation as seen in Figure 9b. The custom widgets specifying the grid color and the highlight rectangle in Figure 10a are equally lost; the label "Reset to Defaults" also disappears since it is defined in the program code.

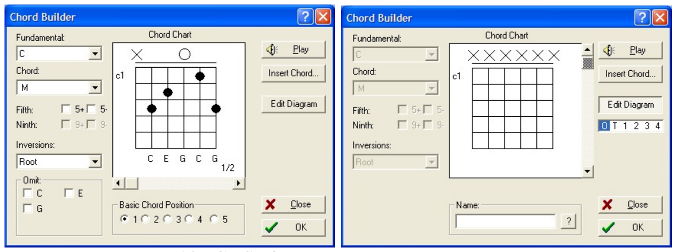


Figure 10: Custom widgets lost in the process.

3) *Not-understandable resource definitions*

Some widgets contained in resource files are accessible for decompilation, but are expressed in a proprietary format that prevent any reverse engineering to properly deduce a resource model from it. Such cases include binary, protected definitions, encrypted widgets (like in most Microsoft applications), thus preventing our approach to "steal" a GUI definition. Table 4 shows some mappings established between the resources and the CUI, especially for the PushButton control. There exists a one-to-one mapping for most of the attributes. It is worth remarking that to generate the name of the new control we use the line number of the textual resource file. The reason is that doing it this way they can be localized easily and mostly in a unique way.

Table IV. Mappings between the resource model and the Concrete User Interface model.

Resource model	CUI model
PushButton	Button
Position = n	name = "control_" + n
X, Y, Width, height	Serve to the creation of boxes with eventually some graphicalAlignment between two components inside a box
Disabled = true	isEnabled = false
Visible = false	isVisible = false
Text = t and '&' is a character of t and the control is a radio button, a push button, a customised button or a check box	defaultMnemonic = the character following '&' in t
Text = t and Content = text	defaultContent = t without the '&' character
Text = t and Content ≠ text	defaultContent = t
Default = true	graphicalEmphasis

4) Dimension of the dialogs

The dimension of a graphical container is expressed in pixels, whereas a resource file expresses the measures in horizontal dialog units and vertical dialog units. The dimensions in Windows are then defined in term of characters. One horizontal (vertical) unit equals 1/4 (1/8) of an average character width (height) of the font used. In order to work out the precise size of the dialogs, the average width and height of the different font types (with different font size) must be taken into account. In our case we assume an 8-point Tahoma font, which is the most common font type and font size in Windows 7 dialogs.

5) Layout of the controls

For a graphical control, the concept of position and dimension inside a window is absent in the CUI model. To place each component, horizontal and vertical boxes (Box metaclass) have to be defined, which requires analyzing the position and dimension of all the controls specified in the resource model. This task is particularly addressed in [8].

6) Dimensions of a control

The dimensions of a container (graphicalContainer metaclass) can be specified in the CUI, in contrast to the controls (graphicalIndividualComponent) which do not have attributes to keep these data. The actual dimension of the buttons, radioButtons, checkBoxes and toggleButton is defined by the length of the text which appears in it, and it cannot have a larger size. If an image is displayed in the button instead of text, its size is undefined since this image resource is not available. ListBox controls also present the same problem: the size depends on the length of its items because the strings contained in a list box are not specified in a resource file.

7) Platform-dependent colors

Sometimes resources use colors that are defined based on the Windows system colors. This means that you can use a color that depends on the system configuration (for example, the default color that is used to fill the window background). This kind of information is not present in the resource files, so the configuration of the platform must be also required.

8) Generalization of the approach

In order to generalize the approach and its support software *UsiResourcer* to other resource files, the following actions need to be undertaken:

- For another decompiler in the <u>same</u> computing platform: resource decompilers produce potentially inconsistent results, as shown in the preliminary study. Therefore, supporting another resource decompiler requires writing another EBNF grammar to reflect its structure. The rest is left unchanged: the Windows 7 resource metamodel does not change, only its injection from another format should change.
- For another version of the same resource file in the <u>same</u> computing platform: formats of resource file constantly evolve over time with new versions of operating systems. Thus, supporting another version or another format requires updating the resource metamodel and the corresponding EBNF grammars. Then, the transformations and the model injection need to be updated. The rest is left unchanged.
- Another UIDL for the CUI model. Since various UIDLs could be used, supporting another UIDL requires implementing the M2M transformations in the terms incorporated in the new UIDL. The rest is left unchanged.

VIII. Conclusion

In this paper, we have presented a re-engineering method in which the GUI resource files contained in its executable files are exploited to recover a CUI model from which various operations can take place, such as GUI editing, regeneration of a new GUI, or any modification. For this purpose, a resource meta-model has been created so as to establish mappings between any resource model obtained from these resource files and CUI model by model-to-model transformations. There are still cases where this process does not produce the full results that have been discussed, such as when resources files contain non-standard widgets, are programmatically defined, or encrypted.

Acknowledgments

The authors would like also to acknowledge the support of the ITEA2-Call3-2008026 UsiXML (User Interface eXtensible Markup Language) project supported by Région Wallonne DGO6 as well as the FP7-ICT5-258030 SERENOA project supported by the European Commission.

References

[1] G. Canfora, M. Di Penta, and L. Cerulo, "Achievements and challenges in software reverse engineering," Communications of the ACM, vol. 54, no. 4, April 2011, pp. 142–151.

[2] G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, L. Bouillon, and J. Vanderdonckt, "A Unifying Reference Framework for Multi-Target User Interfaces," Int. with Comp., vol. 15, no. 3, June 2003, pp. 289–308.

[3] P. Wegner, "Why Interaction is more Powerful than Algorithms," Communications of the ACM, vol. 40, no. 5, 1997, pp. 80–91.

[4] F. Montero, V. López-Jaquero, and P. González, "Model-based Reverse Engineering of Legacy Applications User Interfaces," in Proc. of 2nd Int. Workshop on User Interface Description Languages UIDL'2011 (Lisbon, September 6, 2011). Paris: Thalès, 2011, pp. 128–133.

[5] G. Canfora, A.R. Fasolino, G. Frattolillo, and P. Tramontana, "A wrapping approach for migrating legacy system interactive functionalities to service oriented architectures," Journal of Systems and Software, vol. 81, no. 4, April 2008, pp. 463–480.

[6] L. Bouillon and J. Vanderdonckt, "Retargeting of Web Pages to Other Computing Platforms with VAQUITA," in Proc. of 9th IEEE Working Conf. on Reverse Engineering WCRE'2002 (Richmond, October 28-November 1, 2002). Washington: IEEE Press, 2002, pp. 339–348.

[7] E.J. Chikofsky and J.H. Cross, "Reverse engineering and design recovery: A taxonomy," IEEE Software, vol. 7, no. 1, 1990, pp. 13–17.

[8] Ó. Sánchez Ramon, J. Sanchez Cuadrado, and J. García Molina, "Model-driven reverse engineering of legacy graphical user interfaces," in Proc. of 25th IEEE/ACM Int. Conf. on Automated Software Engineering ASE'2010 (Antwerp, Sept. 20-24, 2010). ACM Press, pp. 147–150.

[9] G. Di Santo and E. Zimeo, "Reversing GUIs to XIML Descriptions for the Adaptation to Heterogeneous Devices," in Proc. of the ACM Symposium on Applied Computing SAC'2007 (Seoul, March 11-15, 2007). New York: ACM Press, 2007, pp. 1456–1460.

[10] J. Vanderdonckt, L. Bouillon, and N. Souchon, "Flexible Reverse Engineering of Web Pages with VAQUITA," in Proc. of 8th IEEE Working Conf. on Reverse Engineering WCRE'2001 (Stuttgart, October 2-5, 2001). Washington: IEEE Computer Society Press, 2001, pp. 241–248.

[11] L. Bouillon, J. Vanderdonckt, and K. Chieu Chow, "Flexible reengineering of web sites," in Proc. of ACM Int. Conf. on Intelligent User Interfaces IUI'2004 (Funchal, January 14-16, 2004). pp. 132–139.

[12] L. Bouillon, Q. Limbourg, J. Vanderdonckt, and B. Michotte, "Reverse engineering of web pages based on derivations and transformations," in Proc. of 3rd IEEE Latin American Web Congress LA-Web'2005 (Buenos Aires, October 1 - November 2, 2005). IEEE Press, 2005, pp. 3–13.

[13] L. Paganelli and F. Paternò, "Automatic Reconstruction of the Underlying Interaction Design of Web Applications," in Proc. of 14th Int. Conf. on Software Engineering and Knowledge Engineering SEKE'2002 (Ischia, July 15-19, 2002). New York: ACM Press, 2002, pp. 439–445.

[14] R. Bandelloni, F. Paternò, and C. Santoro, "Reverse Engineering Cross-Modal User Interfaces for Ubiquitous Environments," in Proc. of IFIP Conf. on Engineering Interactive Systems EIS'2007 (Salamanca, March

22-24, 2007). Lecture Notes in Computer Science, Vol. 4940, Berlin: Springer, 2008, pp. 285–302.

[15] F. Bellucci, F. Ghiani, F. Paterno, and C. Porta, “Automatic reverse engineering of interactive dynamic web applications to support adaptation across platforms,” in Proc. of Int. Conf. on Intelligent User Interfaces IUI’2012 (Lisbon, Feb. 14-17, 2012). ACM Press, 2012, pp.217-226.

[16] H. Samir, E. Stroulia, and A. Kamel, “Swing2Script: Migration of Java-Swing Applications to Ajax Web Applications,” in Proc. of 14th IEEE Working Conf. on Reverse Engineering WCRE’2007 (Vancouver, October 28-31, 2007). Washington: IEEE Press, 2007, pp. 179–188.

[17] M. Dixon, D. Leventhal, and J. Fogarty, “Content and hierarchy in pixel-based methods for reverse engineering interface structure,” in Proc. of ACM Conf. on Human Aspects in Computing Systems CHI’2011 (Vancouver, May 7-12, 2011). New York: ACM Press, 2011, pp. 969–978.

[18] J. Maras, M. Štula, and J. Carlson, “Reusing web application user-interface controls,” in Proc. of the 11th Int. Conf. on Web engineering ICWE’2011 (Paphos, June 20-24, 2011). Lecture Notes in Computer Science, Vol. 6757, Berlin: Springer, 2011, pp. 228–242.

[19] A. Memon, I. Banerjee, and A. Nagarajan, “GUI ripping: Reverse engineering of graphical user interfaces for testing,” in Proc. of the 10th IEEE Working Conf. on Reverse Engineering WCRE’2003 (Washington, November 13-16, 2003). IEEE Computer Society, 2003, pp. 260–269.

[20] J.C. Silva, C.E. Silva, R.-D. Gonçalo, J. Saraiva, and J.C. Campos, “The GUISurfer tool: towards a language independent approach to reverse engineering GUI code,” in Proc. of 2nd ACM Int. Conf. on Engineering Interactive Systems EICS’2010. ACM Press, 2010, pp. 181–186.

[21] E. Stroulia, M. El-Ramly, L. Kong, P. Sorenson, and B. Matichuk, “Reverse engineering legacy interfaces: An interaction-driven approach,” in Proc. of the 6th IEEE Working Conf. on Reverse Engineering WCRE’99 (Atlanta, October 6-8, 1999). IEEE Press, 1999, pp. 292–303.

[22] P. Tonella and A. Potrich, “Reverse engineering of the interaction diagrams from C++ code,” in Proc. of the 19th IEEE Int. Conf. on Software Maintenance ICSM’2003 (Amsterdam, September 22-26, 2003). Washington: IEEE Computer Society Press, 2003, pp. 159–168.

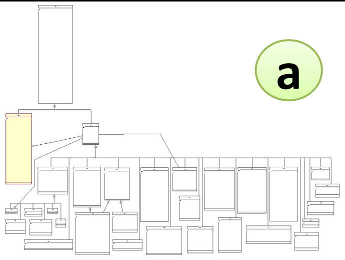
[23] M.I. Muhairat, R.E. Al-Qutaish, and B.M. Athamena, “From Graphical User Interface to Domain Class Diagram: A Reverse Engineering Approach,” Journal of Theoretical and Applied Information Technology, vol. 24, no. 1, 2011, pp. 28–40.

[24] Ó. Sánchez Ramon, J. Sanchez Cuadrado, and J. García Molina, “Reverse Engineering of Event Handlers of RAD-Based Applications,” in Proc. of 18th IEEE Working Conf. on Reverse Engineering WCRE’2011 (Limerick, October 17-20, 2011). IEEE, 2011, pp. 293–302.

[25] I.C. Morgado, A.C.R. Paiva, and J.P. Faria, “Reverse Engineering of Graphical User Interfaces,” in Proc. of the 6th Int. Conf. on Software Engineering Advances ICSEA’2011 (Barcelona, October 23-29, 2011), IARIA Press, 2011, pp. 293-298.

[26] B. Michotte and J. Vanderdonckt, “GrafiXML, A Multi-Target User Interface Builder based on UsiXML,” in Proc. of 4th Int. Conf. on Autonomic and Autonomous Systems ICAS’2008 (Gosier, March 16-21, 2008). IEEE Computer Society, Washington, 2008, pp. 15–22.

Appendix. Correspondence between Resource file (physical name) and Meta-model (logical name) for a dialog box in Windows 7: (a) portion of the meta-model concerned; (b) resource template; (c) static definition; (d) derivation rules; (e) flags.

 a	dialog box template : <id> **DIALOG**[**EX**] <x>**,** <y>**,** <width>**,** <height> [**,** <helpId>] **STYLE** <style>* **CAPTION** "<text>" [**EXSTYLE** <ex_style>*] **LANGUAGE** <language>, <sublanguage> **FONT** <pointsize>**,** "<typeface>" [**,** <weight>**,** <italic>] **{** <control_def> **}** b

Class DIALOG	**DIALOG**	Flag value
Aggregation relationship		
n ≥ 1 is the number of relationship instances in which the DIALOG instance participates	**{ }** and n ≥ 1 is the number of lines in <control_def>	
Attibutes		
DlgID	<id> ::= val(DlgID)	
Extended = true c	DIALOG**EX** d	e
X, Y	<x> ::= val(X)	
	<y> ::= val(Y)	
Width, Height	<width> ::= val(Width)	
	<height> ::= val(Height)	
HelpID ≠ -1	<helpId> ::= val(HelpID)	
3Dlook = true	<style> ::= DS_3DLOOK	
AbsoluteAligment = true	<style> ::= DS_ABSALIGN	0x00000004
Center = true	<style> ::= DS_CENTER	0x00000001
CenterMouse = true	<style> ::= DS_CENTERMOUSE	0x00000800
ContextHelp = true	<style> ::= DS_CONTEXTHELP	0x00001000
DialogIsControl = true	<style> ::= DS_CONTROL	0x00002000
FixedSys = true	<style> ::= DS_FIXEDSYS	0x00000400
LocalEdit = true	<style> ::= DS_LOCALEDIT	0x00000008
ModalFrame = true	<style> ::= DS_MODALFRAME	0x00000020
NoFailCreate = true	<style> ::= DS_NOFAILCREATE	0x00000080
NoIdleMessage = true	<style> ::= DS_NOIDLEMSG	0x00000010
SetForeGroud = true	<style> ::= DS_SETFOREGROUND	0x00000100
SysModal = true	<style> ::= DS_SYSMODAL	0x00000200
val(Language) not NULL	<language> ::= LANG_upper(val(Language))	0x00000002
val(Sublanguage) not NULL	<sublanguage>::=SUBLANG_<language>_upper(val(Sublanguage))	
SetFont = true	<style> ::= DS_SETFONT	
FontName	<typeface> ::= val(FontName)	0x00000040
FontSize	<pointsize> ::= val(FontSize)	
Italic = true, Italic = false	<italic> ::= TRUE, <italic> ::= FALSE	
Weight ≠ -1	<weight> ::= val(Weight)	
ShellFont = true	<style> ::= DS_SHELLFONT	0x00000048

Other attributes can be inherited from the WINDOW class (in addition to the attribute GeneralStyle always set to popup)