A Non-Invasive Object-Oriented Tool for Software Testing

P. Nesi[!] and A. Serra[*]

[!]Dipartimento di Sistemi e Informatica, Faculty of Engineering

University of Florence, Via S. Marta 3, 50139 Firenze, Italy

NESI@INGFI1.ING.UNIFI.IT

tel.: +39-55-4796265, fax.: +39-55-4796363.

[*]ASIC S.r.l., Torino, Italy

December 9, 1994

Abstract

A non-invasive approach for Capture and Playback ($C\&P$) can be a very useful tool for testing applications endowed of a graphic user interface in local and/or distributed environments, and in general for testing applications without modifying their run-time environment. In the software life-cycle, the phases of $C\&P$ are performed after the application design. Since these are close to the deadline of delivering, the time needed for application testing is considered as a high, and frequently unacceptable, cost. In this paper, a new approach for non-invasive $C\&P$ testing techniques is proposed. This is strongly based on the object-oriented paradigm at both hardware and software levels. In particular, a new board for image grabbing and pattern matching, and a new object-oriented language for specifying the tests have been defined. The main goals of this new approach are (i) the reduction of testing time by supporting the reuse of tests (coded by using a specific language) at each level of abstraction, and (ii) the anticipation of the capture-phase of testing with the system design.

Keywords: testing, capture and playback, non-invasive, object-oriented, object-oriented language, real-time, distributed applications.

1 Introduction

One the most powerful methods for testing software applications with man-computer interactions is the so-called Capture and Playback ($C\&P$) approach [2], [3]. This is based on two distinct phases, which are the Capture and the Playback (see Fig.1). During the Capture, each computer-user interaction is collected – i.e., all the messages which are displayed on the monitor (presentation of messages, windows, text, etc.), all the keys pressed coming from the keyboard and all the motion and clicks of the mouse. In this way, the histories of computer interactions in the form of sequences of interactions are collected, and stored in the form of a script file. The histories of man-computer interactions are reproposed during the Playback to the computer interfaces (simulating the presence of keyboard and mouse) for simulating the presence of the user itself. After each simulated stimulus the responses of the computer can be tested to verify the application answers on the video signal according to the application correct behavior (which is supposed to be known).

The definition of the test structure can be made on the basis of the application structure and functionalities by using a Test Generator – e.g., T of IDE, SoftTest of Bender & Associates, [20], [1], [14]. The history of sequences and the sequences of operations that must be performed for testing the applications are usually specified in an *ad-hoc* programming language, and thus are coded in script files (in ASCII form). This capability allows the direct modification of these programs, following the syntax and semantics of the script language [2]. Usually, in these script files references to the elementary *entities* which are typical of the man-computer interactions, such as pixel patterns on the video screens (with their absolute position), mouse absolute positions and clicks, and sequences of keys from keyboard, are present.

Automatic testing tools can be classified in software- and hardware-based $C\&P$ approaches. The software-based approaches are also called *invasive* approaches, since they consist of programs that run (capturing and playing back) on the same machine on which the application under test is running – e.g., XRunner by Mercury [5], Automator QA (Direct Technology), Auto Tester (Software Recording, Inc.), SQA-Robot (Software Quality Automation), CAP/BAK (Software Research), Sterling TestPro (Sterling Software, Inc.), etc. The hardware-based approaches (also called *non-invasive* approaches), consist of a *dedicated hardware* which is capable of grabbing the signals passing through the cables linking the monitor (i.e., video grabber), the mouse and the keyboard to the computer body itself. In this case, a second computer called "HOST" is needed to control the processes of $C\&P$ (see Fig.2) – e.g., CAP/BAK NI (Software Research), Ferret (Tiburn Systems, Inc.), Evaluator (Elverex and Eastern Software), etc. The non-invasive

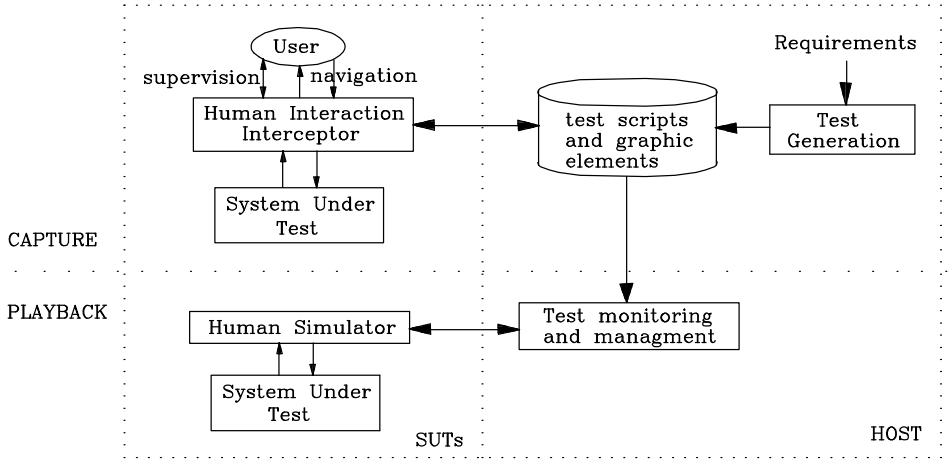

Figure 1: The Capture and Playback phases.

approach is much more robust with respect to the invasive, since the latter obviously changes the context of the System Under Test (SUT). This can also change the performance of the SUT and all testing results. Moreover, some failures could be due to the presence of the tool itself and some errors may not be revealed, especially if the operating system of the SUT is a multitasking system.

The *non-invasive* approaches can be very useful for executing tests in real-time and for testing distributed environments where several computers execute the same or different applications at the same time (running on the same or different operating systems). On the contrary, the *invasive* approaches are less suitable for testing: (i) time-dependent operations, (ii) distributed applications, and (iii) low-level software procedures such as part of the operating system, drivers, command shell, etc. The better performance of non-invasive approaches with respect to the invasive approaches is obtained at the expense of an increment of the tool costs and complexity. In fact, a dedicated hardware is needed in order to implement non-invasive approaches. On the other hand, these offer a performance improvement and the possibility of real-time testing of distributed applications.

$C\&P$ tools are usually employed for controlling the product quality and to assess the new versions of software product already used and tested. In both cases, the phase of $C\&P$ is performed after the design and formal test of the application. For companies which are focused on software development, this fact is strongly unsatisfactory, since the time which is necessary to perform the test is often passing the delivery date, due to delays accumulated during the software development. For this reason, the time which is needed to test the application by means of a $C\&P$ is psychologically considered as a high and frequently unacceptable cost, though the testing obviously produces benefits in the software quality. This is particularly evident on the

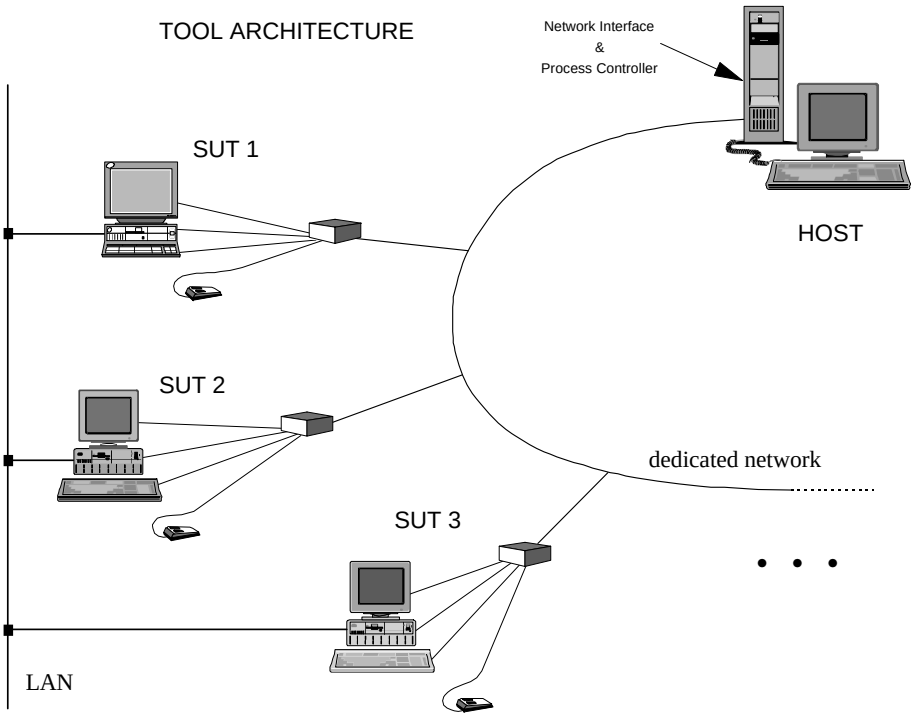

Figure 2: Non-invasive approach to Capture and Playback with capabilities for testing distributed (the SUTs could be connected or not to the same local area network, LAN) and/or multiple applications. The *dedicated hardware interfaces* are connected to the HOST by means of a dedicated network.

first marked release (since, no previous "history" is available) rather than on subsequent versions of the same product. This is due to the fact that, during the first time of testing by means of a $C\&P$ approach all the screen elements (windows, dialog boxes, buttons, scroll bars, list box, etc., see Fig.3), mouse motions and keyboard sequences must be captured, and the script must be generated and refined. On the contrary, for testing a new version of an already tested application (i.e., regression testing) only a part of the work performed in the last capture phase must be done again, for modifying the scripts and for capturing new graphics features, if any, according to the changes occurred in the application itself. This problem can be strongly reduced if the script generation is anticipated with respect to the end of the application development. This is possible only if the script files can be written in a high-level language without specifying the implementation details of the application under test.

As remarked in many studies and applications, one of the main limits in the first generation of $C\&P$ tools, when applied to perform regression testing on applications endowed of Graphical User Interface (GUI), is the difficulty in maintaining consistency between the application and the recorded scripts. This is due to the fact that these tools need the absolute stability (equality at the pixel level) of the graphic interface. Even a very negligible change in the user interface, such as the depth of a border of a window, makes the scripts recorded unusable on the new version of the software under test. This means that, in practice, a continuous activity of regenerating old scripts is needed, and the regression testing is no longer an automatic process.

Another problem of traditional $C\&P$ approaches is that the script programs are neither easy to be understood nor easy to be modified. Though, they are usually programming languages with mnemonic keywords and, thus, they are changeable by using a text editor. This is due to the fact that human operations described in the scripts (mouse motions and clicks, screen checking, etc.) are usually referenced by means of physical descriptions and not by using logical labels.

In this paper, an evolution of the non-invasive $C\&P$ approaches [2] is proposed. This new approach is strongly based on the object-oriented paradigm at both hardware and software levels of the systems: "Hardware" in the sense that a new object-oriented image grabber board for capturing and searching patterns in real-time has been implemented; "Software" in the sense that the entire application of $C\&P$ is object-oriented and a new language called LOOT (Language Object-Oriented for Testing) for describing the script sequences has been developed. It should be noted that the object-oriented paradigm allows the representation of relationships among the main entities which are present in the man-computer interaction such as windows, buttons,

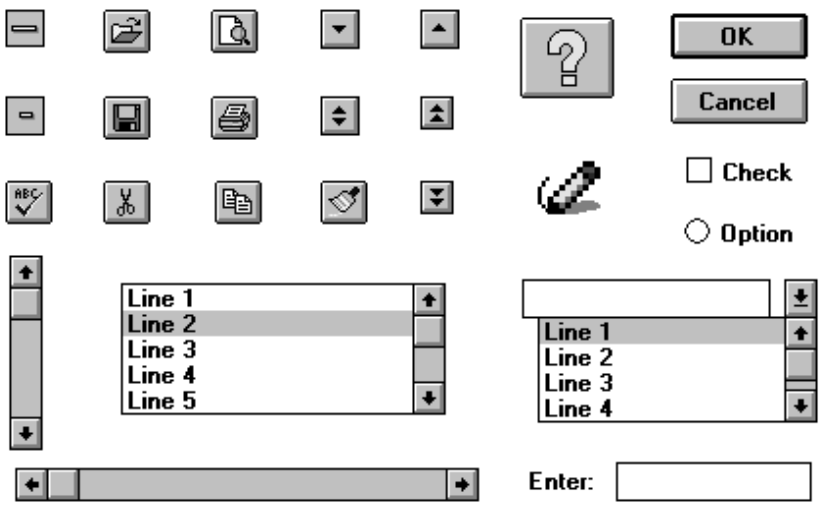

Figure 3: Examples of visual entities grabbed in the phase of capture.

patterns, mouse motions and actions etc., which have also been defined in many object-oriented user interfaces proposed in the literature [12], [15], [9], [16], [22]. In addition, the object-oriented paradigm has the capability to support reuse of already defined objects and to reduce the cost of maintenance and expansion. Therefore, the main goals of this new approach are (i) reducing of testing time – i.e., reducing the time of capture, (ii) overlapping of the testing phase with the earlier phases of Software Life-Cycle [11], and (iii) solving the above-mentioned problems related to the script maintainability and understandability, thus preserving the advantages of the non-invasive testing tools. The approach proposed is a sensible improvement with respect to other $C\&P$ tools based on object-oriented paradigm, such as Sterling TestPro (Sterling Software, Inc.) and XRunner (Mercury Interactive Corp.), since they are invasive and not fully object-oriented in the above explained meanings; therefore, they cannot provide all the facilities which are present in the $C\&P$ tool presented.

This paper is organized as follows. In Section 2, the main aspects related to the adoption of the object-oriented paradigm for the definition and implementation of a non-invasive testing tool for $C\&P$ are reported, with a discussion about the main drawbacks of the traditional approaches. In Section 3, the architecture of the testing tools is proposed. In Section 4, some examples about the Language Object-Oriented for Testing, LOOT, are reported in order to show its main capabilities. Conclusions are drawn in Section 5.

In the rest of the paper, the word "entity" will be used for representing simple and complex visual elements (e.g., windows, icons, borders, patterns, etc.) as well as for mouse and keyboard actions, while the word "gadget" refers only to simple or composite objects which are the typical elements of a dialog box or window (e.g., Button, ListBox, ScrollBar, etc.).

2 Object-Oriented-Based Capture & Playback

The approach proposed in this paper has been motivated by the fact that most of the classical $C\&P$ approaches have several drawbacks. In particular:

1. Testing of a new version of a previously tested application: it can happen that a slight change in the visual profile of a dialog box (or window, etc.), such as the displacement of a gadget (i.e., a button, a scrollbar, etc.) of the dialog box *is recognized* as a severe error even if the semantics of the dialog box (or window, etc.) is unchanged (see Fig.4).

2. Testing of a new version of a previously tested application: it can happen that a slight change in the visual profile of a dialog box (or window, etc.), such as the addition of a new gadget (or menu) *is not recognized* as a severe error even if the semantics of the dialog box (or window, etc.) has been *substantially* changed. Moreover, in this case, the old test, coded in the script, is performed on the new version of the dialog box (or window) without detecting the differences between the old and the new dialog box (or windows). The verification of differences about the shape of a composite visual entity must be explicitly coded in the script of the test. This could be a problem since in traditional approaches the comparison must be performed at the pixel level.

3. In testing a new version of a previously tested application, it can happen that a slight change in the visual presentation, such as the displacement of the window (or dialog box), when it is open, *is recognized* as a severe error even if the semantics of the application is unchanged and the graphic entities are the same. This happens even when a different environment is used during two different sections of testing the same application (of the same version).

4. During the capture, a considerable number of patterns (visual – i.e. 2D, textual – i.e. 1D, etc.) are collected and associated with the entities (menu, text, button, window, mouse motion, etc.) of the application under test. In this way, in the Playback phase the presence of these patterns can be verified. Consider a window-based application of medium

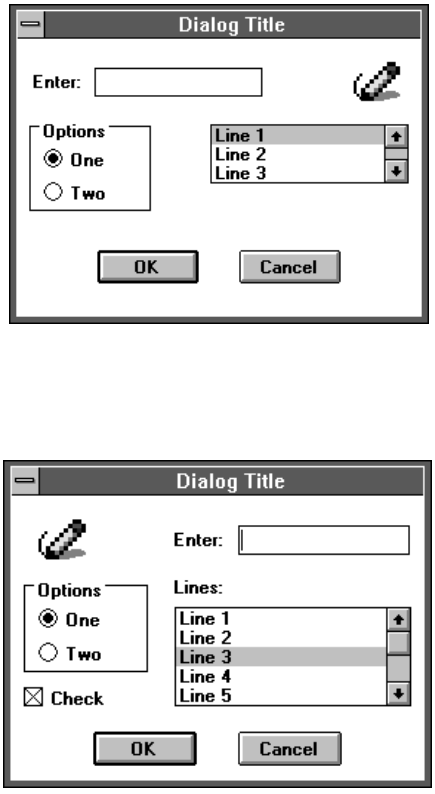

Figure 4: Semantically equal dialog boxes. These are usually sources of severe errors in some traditional $C\&P$ approaches.

size: during the capture phase, it can be very easy to have to collect 100-200 patterns. The tool of capture maintains this information by generating a script file describing the history of the capture. This script file is substantially the same file as that which will be used for testing the application during the playback. The information collected is very difficult to reuse for testing other applications, even if the patterns are the same as those of many other applications. However, the reuse of this information could shorten the time of capturing. This fact is more evident if we consider that most of the companies tend to reuse software components during the application development following the criteria of software engineering (for example the dialog box for selecting files). This is also due to the fact that there exists international standards for the definition of the actions allowed on user interfaces, in the sense that the semantics of certain visual entities and actions is standardized (position of certain selections on the menu, adoption of certain combinations of keys for closing an application, mechanism for selecting and drugging things on the screen, etc.).

5. Even if the script files can be generated by means of an automatic section of capture, the operation of manual capturing is sometimes mandatory – for example, when the application must be tested against specific combinations of keyboard characters, or sequence of commands, etc. The manual capturing is very difficult since the syntax and semantics of the script language are too much related to the low-level details of the visual aspects of the application. This means that it is impossible to recover the application structure only by knowing the generic behavior of the user interface of the application. This obviously means that the application is not completely observable from the outside, otherwise it should be a banal application. On the other hand, the main structure of the script can be generated by a generic description of the application behavior provided, since this is defined in the early stages of the application development.

These problems are mainly due to the fact that traditional script languages are usually strictly related to the low-level details of the application, such as the positions of graphics elements, the patterns to be searched, etc. Moreover, the script languages are also procedure-oriented and, thus, they are less suitable for modeling and maintaining the structure of the application. This means that the scripts do not model the relationships among the visual entities which can appear on the screen, and the relationships among the visual entities and other events which are needed to test the application. For example, the relationships between a window and its menu, items and subitems, or between a button and the possible mouse actions

which can be performed on it, are not described by using the traditional script languages for $C\&P$. In addition, if the information related to the visual entities (e.g., pattern, buttons, icons) and the corresponding actions (which can be performed on an entity) are maintained in a unique structure, then a powerful mechanism for reusing the old tests is obtained. This concept is in accordance with the object-oriented paradigm, by which the single entities of an application can be modeled with a unique class containing both data and behavioral aspects [13]. The Object-Oriented Paradigm (OOP) also has many other mechanisms which are very useful for modeling the applications and for providing a support for reusing the visual and textual entities of the applications together with their allowed operations (i.e., entity behavior) [17], [19], [4].

In order to define a fully object-oriented system for Capture & Playback both hardware and software components of the SUT must be in accordance with the object-oriented paradigm. As regards the hardware components, in the non-invasive testing systems, a board to grab and analyze the video screen in real-time is usually present. The screen patterns grabbed during the capture are compared in the phase of playback with specific areas of the current screen in order to *verify* their presence. In order to guarantee the real-time testing the verification is directly performed by hardware. It allows the verification of the presence of given patterns at *given image coordinates*. This is a strong limitation because due to slight modifications of the screen of the SUT, the process of verification fails. For example, a simple displacement of a gadget (i.e., a button, a listbox, etc.) in a dialog-box is confused with the absence of the gadget. This problem can be circumvented by software, reiterating the process of verification in each pixel of a given area, thus searching for a pattern in a given sub-frame of the screen. It should be noted that a software-based solution to this problem is completely unfeasible if real-time testing is required, thus a hardware real-time search is mandatory. In particular, a dedicated hardware based on programmable gate-arrays and fast correlators has been specifically developed for the $C\&P$ tool described in this paper. This hardware is capable of searching image patterns on the whole screen in real-time.

Therefore, this hardware supports the object-oriented management of the problem. In fact, if a dialog-box D_1 contains a certain gadget G_1, its presence must be directly recognized by asking the hardware where G_1 is in the D_1. Once G_1 has been found, then the request of the execution of an operation on that gadget can be required directly to that gadget. Therefore, the instruction used to test G_1 is semantically reduced to test G_1 of D_1. In this way, the script files can be independent of the visual position of the visual entities on the screen, hence only high-level relationships among these are needed. In this case, the script file can describe the

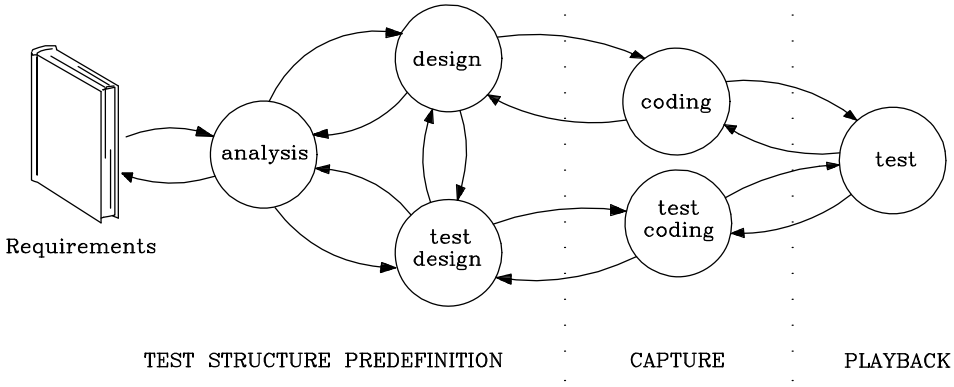

Figure 5: The software and test developing life-cycles.

testing procedure at a higher level with respect to what is possible by using position-dependent languages such as the traditional script files.

It should be noted that the object-oriented paradigm is present in the software architecture at various levels. In particular:

1. For defining a Language Object-Oriented for Testing (LOOT). By using this language the operations which must be performed for testing the application can be described at a high-level of abstraction. It also has the capability of describing the test at various levels of abstraction in order to reach the final version for refinement, possibly during the application development. In addition, it is capable of defining new entities and reusing the already defined entities. The adoption of this approach defines a different life-cycle (see Fig.5) for the development of the test scripts which is strongly anticipated with respect to the classical $C\&P$ approaches that can be used only after the coding of the software that has to be tested. In fact, a phase of test preparation, where the test structure is defined on the basis of system analysis and design (following a program-based approach at the tests preparation), is present. In this phase, the structure of test scripts is prepared; then, these are used during the coding of the applicative software for appending visual details (i.e., phase of details capture), and thus for completing the capture phase.

2. For defining the tool itself for $C\&P$. Since the tool architecture is also object-oriented, a major confidence is attributed to its capabilities of expressivity and expandability. This is due to the fact that most of the primitives which are available at the language level are also used inside the tool for generating the script files in the LOOT language; thus, the same object-oriented model is used for modeling the application under test, and during the capture and the playback.

3. For modeling entities which are used for applications testing. These entities are also stored into an object-oriented database. For each entity (button, dialog box, icon, etc.) both static and dynamic descriptions are stored. Static descriptions correspond, for the visual entities, to patterns, while dynamic descriptions are the possible operations which are allowed on that entity.

Most of the above-mentioned facilities allow the reduction of Capture time by reusing objects and classes and by beginning to specify the test of the application even if the final visual aspect of the application is not completely available (e.g., test structure pre-definition). In addition, the reduction of the Capture time is also obtained by reusing already defined entities. The reuse can be done at various levels of abstraction, thus it consists in both time saved for capturing patterns, and/or time saved for describing the procedures of testing of already tested parts of applications, such as dialog boxes, etc. Hence, as soon as the application if finished the scripts are ready for testing the application with the Playback approach.

3 Testing-Tool Architecture, and Application Modeling

The tool proposed for *C&P* includes a real-time object-oriented kernel supporting the concurrency at various levels. With this kernel the concurrency can be among objects of the same or different classes as well as among methods of the same class. This is indispensable since the testing tool must be capable of keeping under control at the same time several SUTs connected to the host by means of the dedicated network (see Fig.2 in Sect.1). Through the dedicated network, asynchronous messages can arrive at/from the several SUTs; thus, the kernel must be capable of reacting in real-time to the stimuli sent from the SUTs (such as alarms and events in general). For stressing distributed applications the responses of the SUTs against strict deadlines are usually tested – e.g., simulating a set of "contemporaneous" requests of the SUTs to the same host [21]. Therefore, a run-time kernel for the tool must be characterized by real-time capabilities [6]. To this end, the object-oriented real-time kernel of the CASE tool TOOMS has been adopted [7], [10] (i.e., the kernel of the real-time language TROL [8]). It consists of (i) a low-level interface between the object-oriented system of classes and the operating system (OS/2 2.1, or SUN Solaris 2.3), and (ii) a set of classes like Thread, Path, Temporal Constraint, Clause, etc.

In order to provide the basic elements for modeling the application under test several classes have been defined around the real-time kernel, such as `Pattern`, `Font`, `Button`, `Keyboard`, `Mouse`, `Screen`, `Window`, `DialogBox`, `TextInput`, `Text`, `Icon`, `OnOffButton`, `RadioButton`, `ScrollBar`,

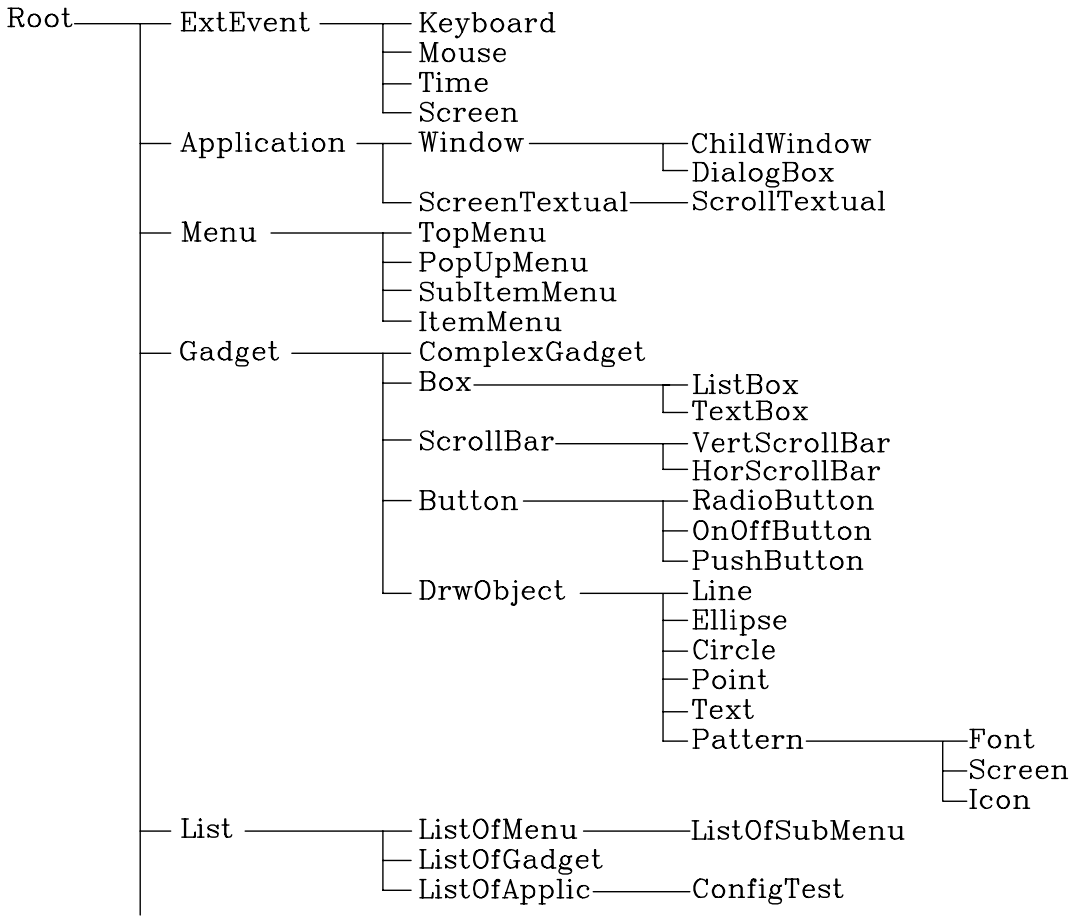

Figure 6: Hierarchy reporting the classes of the testing tool (a part) which are used for modeling the applications under test.

`Menu`, etc. These classes have been organized in a specialization hierarchy exploiting the object-oriented capabilities of inheritance and polymorphism (see Fig.6). Among these classes, elementary drawable objects have also been modeled, in order to provide a support for modeling more complex graphic entities.

In Fig.6, the class hierarchy of the testing tool proposed is reported. This hierarchy is quite different with respect to other hierarchies used for modeling GUIs endowed of windowing systems, such as CommonView, Zinc++, MS Visual C++, etc. This is due to the fact that this hierarchy models the graphic environment from a different point of view, which is the testing of the applications by manipulating the visual entities from outside the SUT, rather than visualizing the entities in order to represent certain actions for getting certain information as in the windowing systems.

The main objects of the testing system are instantiated from the classes specialized from class `ExtEvent`, which are: `Keyboard`, `Mouse`, `Time`, and `Screen`. These are the classes which

model the physical entities of the application under test. Therefore, for each application, there exists one object for each of these classes. Note that, these classes are internally concurrent and, thus, they are capable of satisfying asynchronous requests with respect to the current operation, such as for managing exceptions.

In this hierarchy, the classes `Window`, and `ScreenTextual`, have been specialized from the class `Application`. This means that an application can be either built on a graphical user interface (endowed of a windowing system) or on a classical ASCII interface. According to the OOP, the specialization mechanism defines that the subclasses inherit both Attributes and Methods from the superclasses; therefore, such inherited features are not reported in the definition of specialized classes:

```
Class root
{
Attributes:
       char *Name; // object's name
       ... ... ...
Methods:
       Set_Name(char n[]);
       char * Get_Name();
       ... ... ...
}

Class Application public : root
{
Attributes:
       Dimension dim; // screen dimension  (dx,dy)
       ... ... ...
Methods:
       Open();
       Close();
       Dimension Get_Dimension();
       ... ... ...
}
```

From the point of view of the testing tool, during the testing of a set of applications on several SUTs, the testing tool collects the information corresponding to each application in instances of the class `Application`, which in turn are collected by an object of the class `ConfigTest` (specialization of the class `ListOfApplic`).

At the level of the testing tool, an application is modeled by specifying its structural hierarchy starting from an instance of a class of the Application sub-hierarchy. A window application is modeled as an object of class `Window`. This is turn contains an object of the class `ListOfGadget` and one of the class `ListOfMenu`, etc. Therefore, by using polymorphism, instances of the sub-hierarchies Gadget and Menu can be collected in the corresponding lists, respectively:

```
Class Window public : Application
{
Attributes:
       Pattern id;     // pattern of identification
       ListOfMenu TheMenu;
       ListOfGadget TheGadget;
```

```
        Position pos;   // position on the screen
        ... ... ...
Methods:
        Window(Pattern);
        Set_id(Pattern);
        Pattern Get_id();
        Localize(); // Set position ``pos'' by calling hardware facilities
        Position Get_Position();
        Move(int, int);
        Size(int, int);
        Add_Gadget(Gadget *);
        Add_Menu(Menu *);
        Remove_Gadget(Gadget *);
        Remove_Menu(Menu *);
        ... ... ...
}

Class ChildWindow public : Window
{
Attributes:
        Window * father;
        ... ... ...
Methods:
        ... ... ...
}
```

Classes belonging to the class `Window` sub-hierarchy have the attribute `id` which is the pattern of identification for the window. This allows the testing tool to find the position of the window on the screen by calling the image grabber board inside the method `Localize()`.

The object `TheGadget`, belonging to the class `ListOfGadgets`, maintains the structural relationships among the elementary components belonging to the window and the application itself, while the `TheMenu` (of class `ListOfMenu`) describes the menu facilities of the Window. Each menu may be logically related to either the opening of a DialogBox or a ChildWindow. This information is maintained at the menu level by means of Pointers. The referenced DialogBox or ChildWindow is in turn defined in the same way. Hence, the application is hierarchically described inside the testing tool. It should be noted that the structural description is independent of the low-level details comprising the details about the application behavior. Low-level details (such as the position of a window or the presence of a particular pattern) can be automatically recovered or added later. Therefore, the hierarchical structure of an application can be described a lot of time before delivering the final version of the application itself; in particular, this can be done after the early phases of the software life-cycle. The description of an application hierarchy can be saved into the object database for its future reuse. The same mechanism is performed to describe the structure of DialogBox; therefore, the future reuse of these complex entities is strongly facilitated with respect to a procedural approach of testing tools. Analogous mechanisms are used to define complex gadgets by using a set of simple Gadgets. This is made possible by means of the class `ComplexGadget` which is a Gadget containing an object of the class `ListOfGadget`.

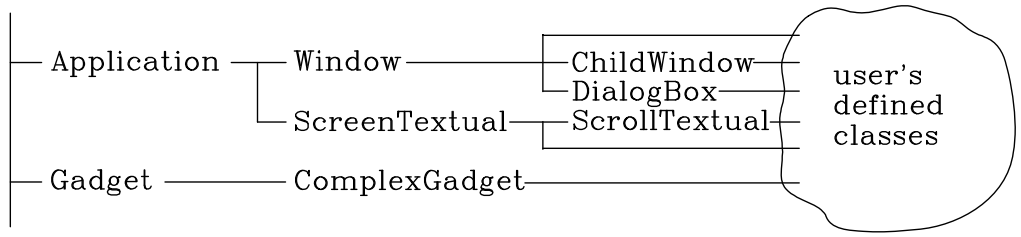

Figure 7: Classes which can be conceptually specialized by the user and their relationships with the tool classes.

The specific details about the applications structure and behavior are described only when the final version of the application under test is available. For example, (a) two different applications can present the same structural hierarchy, but a different detailed behavior, (b) two versions of the same application can have the same structural description while presenting behavior changes. Therefore, by modeling these two cases with the tool proposed, static (structure) and dynamic (behavior) aspects are managed in two different ways. It should be noted that both these conditions generate descriptions which can be reused and maintained related to each other for ensuring congruence.

The hierarchical organization is the main support for decreasing the Capture time and allowing the reuse of parts (e.g., DialogBox, ChildWindow, Button, Icons, etc.) of applications already captured and described. As was pointed out, this organization also allows the script generation of the application after the early phases of the application development.

In order facilitate the building of the structural and behavioral descriptions of a whole Application as well as those of DialogBox, Buttons, etc., these descriptions can be specified by means of a formal language called LOOT (Language Object-Oriented for Testing). This language allows the description of the instances of the leaf classes of the hierarchy presented in Fig.6. For example, the details about the attribute values of a ListBox, id (pattern of identification), dimensions, color, etc. can be specified. Moreover, in order to confer a high degree of flexibility to the language, new classes as conceptual specializations of Window, ChildWindow, DialogBox, etc. can be defined as depicted in Fig.7.

From the point of view of the testing tool, the user's defined classes in LOOT are instances of their corresponding conceptual superclass, while at the language level the user's defined classes are a sort of template for instances, which in turn can be further specialized, for example see Fig.8 (this mechanism has also been used in TOOMS/TROL [7], [18]). The user's defined classes are hierarchically organized and stored into the object databases (i.e, following the specialization hierarchy). Therefore, the specialization among the user's defined classes improves the

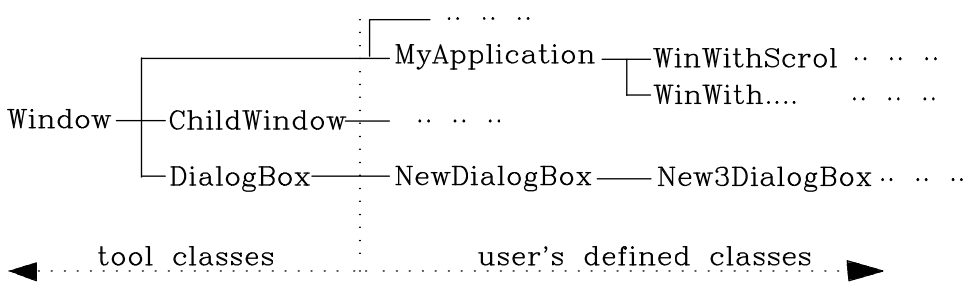

Figure 8: Example of conceptual specialization of user's defined classes and sub-hierarchies from tool classes.

reusability of the modeled applications. The user's defined classes also support the process of instantiation; this allows the description of very complex applications, with a minimal effort.

For the user's defined classes, specific methods called operations can be defined. These operations can be used to describe the elementary actions which can be performed on these entities in order to test them. For example, the operation `DoubleClick()` defined for a class `TextButton` (derived from the `ComplexGadget` and containing both a Text and a Button) can be associated with the necessity of text editing. In this way, both structural and behavioral aspects of the user's defined classes can be stored in a single chunk and reused in the future.

It should be noted that many standard windowing systems, such as MS-Window, PM OS/2, Motif, etc. have the capability of defining some visual details of the application by using particular resources languages, such as the Resource or UIL (User Interface Language). In the future, the testing tool proposed will be able to convert this information for building the corresponding classes inside the testing tool. This will further reduce the capture time.

4 Language Object-Oriented for Testing, LOOT

The object-oriented paradigm allows the definition of the structure and behavior of new abstract objects by means of the concept of class and method. According to this paradigm, classes describing the entities under test must belong to the testing procedure and, thus, the so-called elementary operations of test belong to the classes as discussed in the previous section. This point of view transforms the traditional concept of a sequential script file in which the low-level details are distributed with the high-level details and the behavior of each entity cannot be clearly identified and reused.

Following this new point of view, the script file is transformed in a sequence of class declarations followed by a very short sequence of high-level operations. This is due to the fact that the details about the test of the major entities (e.g., window objects, dialogbox objects)

are encapsulated inside the operations of the classes describing them. These in turn use the operations defined for the smaller objects (e.g., Buttons, Icons, etc.) during their test.

For these reasons, LOOT allows both the definition of new classes (conceptually derived from the testing tool classes as depicted in Fig.7 and Fig.8) as well as the instantiation of objects from these classes and the elementary classes of the testing tool (see Fig.6). The definition of a new class is performed by describing both the structural (internal data, i.e., attributes) and the behavioral (allowed operations) aspects of the class. For example, the definition of class `NewDialogBox` comprising two `TextBoxes` and two `PushButtons` plus several operations, is conceptually derived from the class `DialogBox`, specified as:

```
Class NewDialogBox specialization DialogBox
Gadgets:
       Name, Surname : TextBox;
       Close, Cancel : PushButton;
Operations:
       NewDialogBox (na:String, su:String, cl:PushButton, ca:PushButton)
         {  // operations of instantiation
         Name.Set_id(str2pattern(na)); // identification set
         Surname.Set_id(str2pattern(su)); // identification set
         Close= cl;
         Cancel=ca;
         }
       WriteText(t1: String, t2: String)
         {
         Name.Write(t1); // call the method Write() of class TextBox
         Surname.Write(t2);
         }
       ReadText(t1: String, t2: String)
         {
         Name.Read(t1); // call the method Read() of class TextBox
         Surname.Read(t2);
         }
       CheckText(na: String, su: String)
         { // checking the presence of known Name and Surname
         tmp1, tmp2: String; // temporary objects
         Name.Read(tmp1);
         Surname.Read(tmp2);
         if (tmp1==na and tmp2==su) return(1);
         else return(0);
         }
       Close() // close the dialog box by clicking on the icon Close, saving strings
         { Close.Click(); // call the method Click of class PushButton }
       Cancel() // close without saving strings
         { Cancel.Click(); // call the method Click of class PushButton }
       .....
       .....
       .....
Events:
       .....
       .....
end;
```

The above class inherits from the class `DialogBox` the standard operations such as `Move()`, `Size()`, `Get_Position()`, `Localize()`, etc. (most of these are directly inherited from the class `Window`). Among the user's defined operations, `CheckText()` has been defined in order to verify the presence of known strings in the corresponding TextBox. The class definition for describing the structure and the elementary operations of an application can be defined since the early

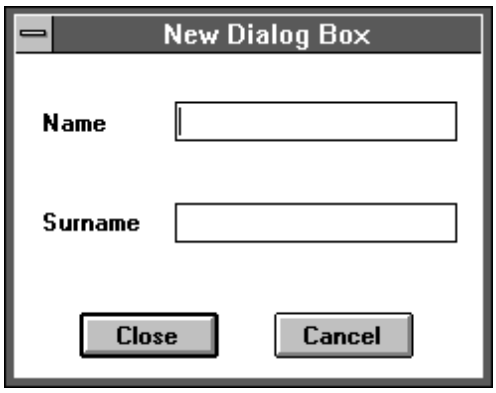

Figure 9: Visual aspect of an object instantiated by the class `NewDialogBox`.

phases of the software life-cycle. In fact, the structures of the dialog boxes are frequently the first things to be defined.

After a class definition, instances of this class can be defined and used (see Fig.9). On these instances, both the operations inherited from the fundamental classes of the testing tool and defined in the class description itself, are allowed. The process of instantiation consists in describing the details of the class attributes (i.e., Gadgets and Menus). This is performed by following the rules defined in the operation of instantiation of the class, for example:

```
CANCEL (Patt_ca), CLOSE (Patt_cl): PushButton
.  .  .
A_NewDialogBox ( "Name", "Surname", CLOSE, CANCEL ) : NewDialogBox
```

where the objects `CLOSE`, and `CANCEL` are two already defined PushButtons objects with their patterns, which could be used in many other dialog boxes.

Complex classes can be defined by means of a process of specialization, such as the class `New3DialogBox` which has been obtained by specializing the class `NewDialogBox` (see Fig.10 for the specialization hierarchy):

```
Class New3DialogBox specialization NewDialogBox
Gadgets:
       Age : TextBox; // new gadget
       pat : Pattern; // constant pattern
Operations:
       New3DialogBox (na:String, su:String, ag: Integer,
                      cl:PushButton, ca:PushButton)
         {  // operation of instantiation
         Name.Set_id(str2pattern(na)); // identification set
         Surname.Set_id(str2pattern(su)); // identification set
         Age.Set_id(integer2pattern(ag)); // identification set
         Close= cl;
         Cancel=ca;
```

Figure 10: Visual aspect of an object instantiated by the class `New3DialogBox`.

```
        pat=str2pattern("New3DialogBox"); // static initialization
        pat.Set_Position(30,45);          // static initialization
        }
      .....
      .....
Events:
      .....
      .....
end;
```

This new class inherits from the class `NewDialogBox` all its `Gadgets`, `Operations`, and `Events`. According to OOP, in a subclass the `Operations`, `Gadgets`, and `Events` (that will be discussed later) inherited from the superclass can be overwritten, and new `Operations`, `Gadgets`, and `Events` defined, following the rules of monotonic inheritance. For example, the specialized `DialogBox` has been obtained by adding two new Gadgets: `Age` which is a TextBox, and the Pattern `pat`. In Fig.10, an example of an object instantiated from that class is given. It should be noted that it is only a particular case that the instances shown in Fig.9 and Fig.10 present the same values for the attributes inherited (in the sense of dimension of TextBoxes, type of Texts associated with the TextBoxes, position of Gadgets in general, type of PushButtons, etc.). In fact, all dialog boxes having three TextBoxes, two PushButtons and one Pattern, placed in any position, could be regarded as instances of class `New3DialogBox` and, thus, they can be stored in the Object Database and subsequently recovered by means of a conceptual navigation.

Once the class definitions are performed, the class which represents the application under test (for example, as a specialization of class `Window`), and the structure of the application have also been defined. At this point, the main script can be written describing at a high-level the process of testing:

```
// class definitions
......
......
```

```
// instantiation
theapp: MyApplication;
tmpDB : DialogBox;
FB : FoundBox; // window confirming the entry found and containing the
               // the telephone number etc..
NFB : NotFoundBox; // window affirming that the entry was not found
......
......

// program
theapp.Open();
theapp.OpenFile("MyAgend");

tmpDB = theapp.Searching(); // request for opening the DialogBox for searching
                            // it is of type NewDialogBox

tmpDB.WriteText("Mark","Smith");
tmpDB.Close();

When (Screen.Appear(FB)) do
  FB.Get();
  ......
  ......
  if (FB.Telephone()) then
    ........
  else
    ........
  endif
  ......
  ......
  for i=1 to 2 do  /* loop */
    ........
    ........
    ........
  endfor;
  ........
  ........
  FB.OK();
endwhen;

When (Screen.Appear(NFB)) do
  NFB.Get();
  ........
  ........
  NFB.OK();
endwhen;

........
........

theapp.Close(); // close the application ``theapp''
........
........
```

It should be noted that by default a number of *active instances* is defined, in particular: Mouse, Keyboard, Screen, and Time. Active means that they lead into concurrent threads of executions. These objects are the only allowed instances of their respective and homonymous classes of the testing tool. The service of the objects can be requested (by sending messages) in every position of the script and in every operation defined for the classes. For example, to move the mouse to the position (10,30) is enough to write `Mouse.Move(10,30)` or `Mouse.Move(P1)` if `P1` is a `Point` with coordinates (10,30); to find a pattern on the screen:

```
P1 : Position;
apattern : Pattern;
```

```
.......
.......
P1=Screen.Find(apattern);
```

While a timer to the object `Time` must be requested to set a timeout:

```
T1 : Timer;
.......
T1=Time.Start(DOWN, 5 ); // timer of 5 seconds
.......
When ( Time.TimeOut(T1) ) do
  .......
  .......
  .......
endwhen;
```

In the above example, the construct `When-endwhen` has been used to define the actions which must be executed once asynchronous exceptions are detected (e.g., a specific message from a SUT to the HOST). These exception handlers can also be defined inside the class body under the field `Events`. In this case, if an exception is redefined inside a class, and its occurrence reaches the system during the execution of the operations of that class, then the script specified for the exception inside the class is executed instead of the script associated with the exception at the global level. The exception can be defined only on the basis of the operations allowed for the *active instances* (i.e., Mouse, Keyboard, Screen, and Time). In fact, the construct `When-endwhen` is defined at the level of the class `ExtEvent` in the testing tool. The class `ExtEvent` (see Fig.6) is interfaced with the TROL kernel in order to support concurrency and the definition of events with temporal constraints.

In the following, a part of the class definition of class `MyApplication` is reported. It should be noted that the attribute variable `pos` is used inside the definition of class operations in order to request the absolute displacements of the mouse:

```
Class MyApplication specialization Window
Menus:
        TheTopMenu : TopMenu;
Gadgets:
        VS : VertScrollBar;
        HS : HorScrollBar;
Operations:
        MyApplication (.............)
          {
          .......
          .......
          .......
          }
        Point GetVertScrollBarPosition() { return(VS.GetPosition()); }
        Point GetHorScrollBarPosition()  { return(HS.GetPosition()); }
        SetVertScrollBarPosition(p: Point)
          {
          Mouse.Move(pos+p);
          Mouse.ClickLeft();
          }
        SetHorScrollBarPosition(p: Point)
          {
```

```
          Mouse.Move(pos+p);
          Mouse.ClickLeft();
          }
        SelectAMainItemMenu(value: String)
          {
          p: Point;
          p=TheTopMenu.WhereIs(value);
          Mouse.Move(pos+p);
          Mouse.ClickLeft();
          }
        SelectASubItemMenu(itemvalue: String, subitemvalue: String)
          {
          p,p2: Point;
          im: ItemMenu;
          im=TheTopMenu.WhoIs(itemvalue);
          if ( im.HasSubItem() ) then
            p2=im.WhereIs(subitemvalue);
            Mouse.Move(pos+p2);
            Mouse.ClickLeft();
          endif;

          }
        .........
        .........
Events:
        .........
        .........
end;
```

In addition, to what has been shown in the previous examples, also the constructs of `Repeat-until`, `While-do-endwhile`, `Do-while-endwhile`, and `Switch-do-case-else-endswitch` are present in LOOT. Moreover, the definition of procedures and functions is also allowed.

In the LOOT languages, there exists a particular construct to specify which operations must be performed on the different SUTs which are connected on the dedicated network. This is particularly important to test distributed applications. With the construct `OnNode`, the part of a LOOT code which must be executed on a set of SUTs is specified. For example, with:

```
OnNode (n1, n2, n3)
{
.....
Where (Screen.Appear(....) ) do
   ..........
endwhere;

.....
}
OnNode (n4)
{
.....

.....
}
```

it is specified that the LOOT code reported inside the brackets is executed on the SUTs named `n1,n2,n3`, while the other is executed only on SUT `n4`. Obviously, the presence of a given SUT in the construct is mutually exclusive. In the body of an `OnNode` construct, the construct `Where` is used to define little changes of the test behavior depending on the occurrence of particular exceptions. This can be useful if the hardware of the declared SUTs is different or to test

particular operating conditions. In the phase of Capture, the testing tools help the user to define the classes and to inspect the Object Database for reusing already defined classes and objects.

5 Conclusions

An object-oriented non-invasive $C\&P$ approach has been presented. It is suitable for testing applications in distributed environments and for testing applications without modifying their run-time environment. The approach proposed is strongly based on the object-oriented paradigm at both hardware and software levels: "Hardware" in the sense that a new object-oriented image grabber board for capturing and searching patterns in real-time has been implemented; and, "Software", since the entire $C\&P$ tool is object-oriented– i.e., the run-time kernel and the script language for describing sequences of testing called LOOT (Language Object-Oriented for Testing). The LOOT language allows the specification of (i) the structural hierarchy of the application under test, (ii) the application behavior, and (iii) the test procedures. All these descriptions can be saved and reused.

The main features of this new approach are (a) a reduction of testing time by supporting the reuse of old tests (in the form of script programs written in LOOT language) at each level of abstraction, (b) an anticipation of the testing phase overlapping the last phases of the software life-cycle, and (c) a strong improvement in tools behavior with respect to the above-mentioned problems related to the script maintainability and understandability, thus preserving the advantages of the non-invasive testing tools.

Acknowledgments

The authors would like to thank E. Miller of Software Research, S. Bartlett of Tektronix for their suggestions, the technical staff of ASIC s.r.l. for their valuable contribution, and the anonymous reviewers for their comments which were useful for improving the paper.

References

[1] P. C. J. adn C. Erickson, "Object-Oriented Integration Testing", *Communications of the ACM*, Vol. 37, pp. 30–38, Sept. 1994.

[2] ASIC, "User's Manual Hardware Simulator", tech. rep., ASIC S.r.l., Via S. Clemente, 6, 10143, Torin, Italy, 1993.

[3] B. Beizer, *Software Testing Techniques*. Van Nostrand Reinhold, 1983.

[4] G. Booch, *Object-Oriented Design with Application*. California, USA: The Benjamin/Cummings Publishing Company, 1991.

[5] N. S. Bradley, "The GUI Test BUilder: Breaking the Test Bottleneck", in *Proc. of 6th International Conference on Software Engineering and Its Applications (sponsored by: EC2, CXP, CIGREF, and SEE)*, (Le CNIT, Paris la Defense, France), pp. 115–124, 15-19 Nov. 1993.

[6] G. Bucci, M. Campanai, and P. Nesi, "Tools for Specifying Real-Time Systems", *Journal of Real-Time Systems*, p. in press, March 1995.

[7] G. Bucci, M. Campanai, P. Nesi, and M. Traversi, "An Object-Oriented CASE Tool for Reactive System Specification", in *Proc. of 6th International Conference on Software Engineering and Its Applications (sponsored by: EC2, CXP, CIGREF, and SEE)*, (Le CNIT, Paris la Defense, France), 15-19 Nov. 1993.

[8] G. Bucci, M. Campanai, P. Nesi, and M. Traversi, "An Object-Oriented Dual Language for Specifying Reactive Systems", in *Proc. of IEEE International Conference on Requirements Engineering, ICRE'94*, (Colorado Spring, Colorado, USA), 18-22 April 1994.

[9] G. Bucci and P. Nesi, "Impiego di tecniche visuali per la programmazione di controllori industriali", *L'ELETTROTECNICA rivista dell'associazione Elettrotecnica ed Elettronica Italiana*, Vol. 78, June 1991.

[10] M. Campanai and P. Nesi, "Supporting Object-Oriented Design with Metrics", in *Proc. of the International Conference on Technology of Object-Oriented languages and Systems, TOOLS Europe'94*, (Versailles, France), 7-11 March 1994.

[11] M. Chandrasekharan, B. Dasarathy, and Z. Kishimoto, "Requirements-Based Testing of Real-Time Systems: Modeling for Testability", *Computer*, pp. 71–80, April 1985.

[12] B. Cox and B. Hunt, "Object, Icons, and Software-ICS", *Byte*, pp. 161–176, Aug. 1986.

[13] A. DelBimbo and P. Nesi, "Blackboard-Based Concurrent Object Recognition Using and Object-Oriented Database", in *Proc. of the IEEE International Phoenix Conference on Computers and Communications, IPCCC'92, Scottsdale, AZ, USA*, pp. 172–180, April 1-3 1992.

[14] M. J. Harrold, J. D. McGregor, and K. J. Fitzpatrick, "Incremental Testing of Object-Oriented Class Structures", in *Proc. of 14th International Conference on Software Engineering*, (Melbourne, Australia), pp. 68–80, IEEE press, ACM, 11-15 May 1992.

[15] D. Hu, *Object-Oriented Environment in C++ – A User-Friendly Interface.* 14th Ave. Portland, Oregon, USA: MIS Press, Management Information Source, Inc., 1990.

[16] M. A. Linton, J. M. Vlissides, and P. R. Calder, "Composing User Interfaces Using InterView", *IEEE Computer*, Vol. 22, pp. 8–22, February 1989.

[17] B. Meyer, *Eiffel: a Language and Environment for Software Engineering.* Prentice-Hall, Englewood Cliffs, 1988.

[18] P. Nesi, "An Object-Oriented Language and Compiler for Reactive Systems", tech. rep., RT 13/93 Dipartimento di Sistemi e Informatica Facolta di Ingegneria, Universita di Firenze, Florence, Italy, 1993.

[19] O. Nierstrasz, "A Survey of Object-Oriented Concepts", in *Object-Oriented Concepts Databases and Applications* (W. Kim and F. H. Lochovsky, eds.), pp. 3–22, New York, USA: Addison-Wesley Publishing Company, ACM Press, 1989.

[20] R. M. Poston, "Automated Testing from Object Models", *Communications of the ACM*, Vol. 37, pp. 48–58, Sept. 1994.

[21] W. Schutz, "Fundamental Issues in Testing Distributed Real-Time Systems", *Journal Real-Time Systems*, Vol. 7, pp. 129–157, 1994.

[22] M. A. Tarlton and P. N. Tarlton, "Pogo: A Declarative Representation System for Graphics", in *Object-Oriented Concepts Databases and Applications* (W. Kim and F. H. Lochovsky, eds.), pp. 151–176, New York, USA: Addison-Wesley Publishing Company, ACM Press, 1989.

Biographies

Paolo Nesi received the degree in Electronic Engineering from the University of Florence, Italy. In 1992, he received the Ph.D. degree in Electronic and Informatics Engineering from the University of Padoa, Italy. In 1991 he was a visitor at the IBM Almaden Research Center, CA, USA. Since November 1991 he is with the Department of Systems and Informatics of the University of Florence, Italy, as a Researcher and Assistant Professor of both "Computer Science" and "Software Engineering". Dr. P. Nesi is an editorial board member of the *Journal of Real-Time Imaging*, Academic Press, and the project manager of MEPI DIM45 ESPRIT III for the University of Florence. Since 1987 he is active on different research topics, real-time systems, formal specification languages, software metrics, parallel architectures, physical models, image processing.

Antonello Serra received the degree in Electronic Engineering from the University of Turin, Italy. His main interests and research areas include software testing, software quality assurance, distributed systems. He is the president of ASIC s.r.l. Turin, Italy, since its constitution. He is also a consultant of Olivetti Italia.