OpenHMI-Tester: An Open and Cross-Platform Architecture for GUI Testing and Certification

Pedro Luis Mateo Navarro, Gregorio Martínez Pérez
Departamento de Ingeniería de la Información y las Comunicaciones
University of Murcia, 30.071 Murcia, Spain

Diego Sevilla Ruiz
Departamento de Ingeniería y Tecnología de Computadores
University of Murcia, 30.071 Murcia, Spain

Abstract
Software testing is usually used to report and/or assure about the quality, reliability and robustness of a software in the given context or scenario where it is intended to work. This is specially true in the case of user interfaces, where the testing phase is critical before the software can be accepted by the final user and put in execution mode. This paper presents the design, and the later implementation as a contribution to the open-source community, of a Human-Machine Interface (HMI) testing architecture, named OpenHMI-Tester. The current design is aimed to support major event-based and open-source windowing systems, thus providing generality, besides some other features such as scalability and tolerance to modifications in the HMI design process. The proposed architecture has been also integrated as part of a complex industrial scenario, which helped to identify a set of realistic requirements of the testing architecture, as well as to test it with real HMI developers.

Key words: Graphical User Interfaces, GUI Testing, Testing Tools, GUI Verification, Test-based Frameworks, GUI Certification

Email addresses: pedromateo@um.es (Pedro Luis Mateo Navarro), gregorio@um.es (Gregorio Martínez Pérez), dsevilla@um.es (Diego Sevilla Ruiz).

Preprint submitted to IJCSSE Special Issue on Open Source Certification July 1, 2009
1 Introduction

GUIs (Graphical User Interfaces) can constitute as much as 60 percent of the code of an application today [1], and their importance is increasing with the recognition of their usefulness [2]. Given their importance in current developments, testing GUIs for correctness can enhance the safety of the entire system, robustness and usability [3]. This should lead us to conclude that advanced GUI testing tools are present in most of the developments today, but it is not true in most cases.

While use of GUIs continues increasing, GUI testing has, until recently, remained a neglected area of research [2]. Although there is evidence that advanced development processes and tools have helped organizations reduce the time to build products, they have not yet been able to significantly reduce the time and effort required to test them. Clearly, there is a need for improvement in testing support [4], but since GUIs have special characteristics, techniques developed to test conventional software cannot be directly applied to GUI testing [2].

Apart from other limitations we can find in GUI Testing (i.e. coverage criteria, verification process, etc.), one of the most important limitations comes from the amount of different windowing system employed in developments today. GUIs can be implemented using a windowing system chosen from a wide range of alternatives, so open architectures that support different windowing systems should be a clear requisite for GUI testing tools.

This is why the research, design, and development of an open and cross-platform GUI Testing tool is an interesting challenge.

Nowadays in Graphical User Interface Testing we can classify available methods, tools and technologies in three different approaches, depending on the way the test case generation process is performed.

The first approach builds a complete testing model matching the whole GUI Model of the application. This model includes all the objects or components and their properties, and it is analyzed in order to explore all the possible paths on an automated test case generation process [5,6,7,8,9,10,11].

Techniques belonging to the second approach do not build a complete GUI Model: they build a smaller model corresponding only to the part of the GUI to be tested, thus reducing the number of generated test cases. To build and annotate that model, these techniques usually employ modeling languages such as UML [4,12].

Finally, techniques belonging to the third approach do not build any model,
as test cases are generated directly using the GUI to be tested. These techniques normally use capture and replay tools which capture events from the tested application and use them to generate test cases that replay the actions performed by the user [13]. These techniques allow developers to perform a lightweight testing process that only takes into account the required elements, actions and properties of the GUI to be tested, avoiding the rest of unneeded test cases that get automatically generated using other approaches [14].

In this paper we describe a GUI Testing tool architecture belonging to the third approach described above. We propose an open architecture which describes a capture/replay tool [15,16] based on GUI Events System, which represents one of the most used techniques in current windowing systems. The OpenHMI-Tester architecture, by performing a non-intrusive application hooking, is able to capture generated events from an application and post new events to it. The proposed architecture also allows adapting the implementation of a few modules of the OpenHMI-Tester to fit both the windowing system and the operating system in use.

This paper is structured as follows. Related work is presented in Section 2. In Section 3 we present the requirements that provide the driving philosophy behind this design. The OpenHMI-Tester architecture is described in Section 4 and the implementation in Section 5. In Section 6 we discuss some of the most relevant issues of the design presented in this paper. Finally, Section 7 provides conclusions and lines of future work.

2 Related work

As commented before, GUI Testing Tools can be classified in three different approaches depending on the test case generation process:

- First approach: tools that build a complete GUI Model which is explored on an automated test case generation process.
- Second approach: tools that build a smaller model corresponding to the part of the GUI to be tested. These tools use modeling languages to define and annotate the model in order to guide the test case generation process.
- Third approach: these tools do not build any model, as test cases are generated directly using the GUI itself to be tested. These techniques usually use capture and replay tools.

One of the techniques belonging to the first approach is the described in [6] by Memon, Banerjee, and Nagarajan. They describe the GUI Ripping process, a method which traverses all the windows of the GUI and analyses all the events and elements that may appear to automatically build a model composed of
a GUI Forest (a tree composed of all the GUI elements) and an Event-Flow Graph EFG (a graph which describes all the GUI events). This model has to be verified, fixed and completed manually by the developer.

In [7] and [8] they describe DART Framework, which follows this philosophy. In DART, once the model is built and manually verified, the process explores all the possible test cases. Of those, the developers select the set of test cases identified as meaningful, and the Oracle Generator \(^1\) creates the expected output. Finally, test cases are automatically executed and their output compared with the Oracle expected results.

In [5] and [9] White, Almezen, and Alzeidi describe a technique that follows a similar approach. They describe a GUI Model based on reduced FSMs (finite-state machines.) They also introduce the concept of Responsibility (a desired or expected activity in the GUI) and define a Complete Interaction Sequence (CIS) as a sequence of GUI objects and actions that will raise an identified responsibility.

Once the test cases are automatically generated from the model, they are executed to find “defects” (serious departures from the specified behavior) and “surprises” (user-recognized departures from the expected behavior, but not explicitly indicated in the specifications of the GUI.)

This approach [18] focuses part of its efforts on building a model of the GUI from which the test cases are generated automatically. Create and maintain these models is a very expensive process [19]. Since a GUI is often composed of a complex hierarchy of widgets in which many of them are irrelevant to the developer (the same happens with their properties), this process may generate a vast amount of “useless” and “senseless” test cases. It gets worse if we consider GUI elements like graphic panels or similar, which have complex properties whose values are very difficult to store and maintain.

This leads to other problems, such a scalability and modifications tolerance. In these techniques, adding a new GUI element (e.g. a new widget or event) has two worrying side effects: first, it may cause the set of generated test cases to grow exponentially (all paths are explored); second, it forces a GUI Model update (and a manual verification and completion) and the regeneration of all affected test cases. The problem gets worse if we consider that the model contains all the properties of the GUI elements, so minimal changes in the appearance or distribution of the GUI (e.g. the window-border size has changed) may cause a lot of errors during the validation process because the expected outputs used by oracles may become obsolete [1].

\(^1\) A Test Oracle [17] is a mechanism which generates outputs that a product should have for determining, after a comparison process, whether the product has passed or failed a test.
Finally, there are other limitations associated with this approach as, for instance, the fact that the model has to be manually corrected and completed, or that the test case generation process does not take into account dynamic GUI behavior performed by the application code (for example, disabling a widget), or specific widgets properties (e.g. its “disabled” property).

In [4] Vieira, Leduc, Hasling, Subramanyan, and Kazmeier describe a method belonging to the second approach in which UML Use Cases and Activity Diagrams are used to respectively describe which functionalities should be tested and how to test them. The main goal in this approach is to generate test cases automatically from an enriched UML Model.

Models may be enriched in two ways: first, the refinement of activities on UML Activity Diagrams in order to improve the accuracy (e.g. decreasing the level of abstraction); second, making annotations on the activity diagrams by using custom UML Stereotypes which represent additional test requirements.

The basis of this approach is closer to the needs of GUI verification, because testing an scenario usually can be performed in three steps: launch the GUI, perform several use cases in sequence, and exit. Its scalability is better than the previously mentioned approach, because it focuses its efforts only on a section of the model, though the combination of functionalities would lead to a very large number of test cases. The use case refinement also helps to reduce the number of generated test cases. On the other hand this method, as the previously described approach, has two very important limitations: first, the developers have to spend so much effort building, refining and annotating the model, which is not a lightweight process (inconceivable in some methodologies such as, for instance, Extreme Programming [20]); second, these techniques have a low tolerance to modifications, since a change in the GUI forces to review and update the model and to regenerate the affected test cases.

Finally, the techniques belonging to the third approach work as follows: once the application to be tested is launched, the developer interacts with the GUI which generate GUI events that are automatically captured and stored into a test case. This test case can be replayed whenever the developer wants. The generated test cases can be completed by adding new actions or meta-events in order to insert messages, verification points, or anything that can help to refine it. The process of capturing, executing, and analyzing executions is an example of Observation-based Testing [13].

In [14] Steven, Chandra, Fleck, and Podgurski describe a tool for capturing and replaying Java [21] program executions called jRapture. jRapture employs an unobtrusive capture process that captures interactions (GUI, file, and console inputs) between a Java program and the environment. The captured inputs are replayed with exactly the same input sequence observed during
The event capture process uses a modified version of the Java API which lets *jRapture* interact directly with the underlying operating system or windowing system (Peer Components). During this process, *jRapture*, along with the modified API, constructs a *System Interaction Sequence (SIS)* which represents the sequence of inputs to the program together with other information necessary for the future replay. Once the SIS sequences are correctly created, they can be replayed by reproducing the effects of calls to Java API methods (stored in the SIS).

The *OpenHMI-Tester* also belongs to this approach, and follows a philosophy similar to *jRapture*, but it provides an open and portable architecture instead (as described in Section 4). This allows *OpenHMI-Tester* to be independent of the operating system (e.g. Windows, Linux, FreeBSD, etc.), windowing toolkits and systems (e.g. Qt [22] or GTK+ [23]), event capture processes (e.g. by using event listeners or peer components), event filtering rules (e.g. capture only GUI events or also capture signaling events), event execution techniques (e.g. by sending captured events or by using a GUI Interaction API), etc. All these mix and match features depend on the actual configuration of the software being tested.

The *OpenHMI-Tester* architecture is described in Section 4; its implementation in Section 5, and discussions to this approach are included in Section 6.

### 3 Requirements

Before we discuss the details of the architecture of the *OpenHMI-Tester*, the requirements that provide the driving philosophy behind this design are introduced. These requirements have been extracted from GUI developments belonging to medium and large applications done under industrial environments.

Since one of the strongest requirements that we impose to the *OpenHMI-Tester* is that it has to be cross-platform and open to any windowing system (e.g. Qt, GTK, etc.), it has to have an open and flexible architecture. The *OpenHMI-Tester* also should perform a non-intrusive application hooking to the tested software in order to be compatible with both software under development and old developed software.

This architecture has to allow us to implement a clear and easy to use tool which includes both event capture process and event execution process. These processes should work into the real software in order to ensure that test case
execution process (explained later in Section 4) matches with the real execution of the tested software. Also the architecture should provide the tester a stable testing environment (e.g. missing objects tolerance, window moving and resizing support, etc.) and a good overall performance during capture and execution processes.

The developer also should be able to implement advanced testing features (e.g. property pickers, screenshots, etc.) under this architecture.

Finally, the OpenHMI-Tester architecture also requires a data model description which supports the representation of any test case and any GUI or non-GUI event in a scalable way. The architecture also should allow the developer to add new events or actions which include new functionality to the test cases (e.g. pauses, breakpoints, messages, etc.) and to implement a validation process which check if any object property value has the expected value during the test case playback.

The exact representation of this data model is left to the developer (either Markup languages, Script code, etc.), but in any case it has to allow its storing and retrieving during capture and execution processes respectively.

4 Software architecture

The architecture presented in this paper is composed of two main software elements, each one with a different purpose. The first one is the HMI Tester, whose aim is to control record (capture) and playback (execution) processes and manage test suite creation and maintenance. The other one is the Preload Module, a software element which will behave like a module “injected” on the tested application, capturing the generated events and executing new events. Both modules will communicate with each other.

This architecture also may be divided into two parts according to the functionality of the modules. Some functionality is implemented as generic and never changes (e.g. record and playback processes); this functionality is depicted using non-colored boxes in Figure 1, and represents the major part of the whole proposed architecture. Other functionality has to be adapted in order to support the characteristics of the testing environment (e.g. operating system, GUI system, etc.); it is represented by colored boxes in Figure 1.

2 A Property Picker is a tool used in GUI Testing which allows the tester to check the properties of a selected object.

3 By scalable we mean that as the size of the GUI system increases, the number of tests required by the strategy increases linearly with the size of the GUI system [9].
As we can see in Figure 1, the whole process involves communication between three software elements: the two mentioned before and the tested application. The communication between the tested applications and the Preload Module will be performed using GUI events: during capture process the tested application will generate events that will be captured by the Preload Module; during execution process the Preload Module will post new events which will be executed by the tested application. The communication between the Preload Module and the HMI Tester will be performed using sockets or any other IPC mechanism: during capture process the Preload Module will send events generated by the tested application to the HMI Tester for it to store; during execution process the HMI Tester will send events to the Preload Module for it to execute them on the tested application.

Communication between the HMI Tester and the Preload Module will be done by using a communications channel, but how is the communication between the Preload Module and the tested application possible? What is the method or process by which two independent applications can send events to each other? The response is preloading.

Preloading method [24] [25] involves including new functionality on a “closed” application by preloading a library which includes new classes and methods. In the HMI Tester architecture, the Preload Module represents a dynamic library which includes the functionality needed to perform preloading action, event capture and execution processes and communication to the other side. When the tested application is launched by the HMI Tester (in both event capture process and event execution process), it first enables in the operating system the preload option pointing to the Preload Module dynamic library and then
launches the application to be tested. During tested application launching, the Preload Module will be loaded and all testing functionality will be included in the tested application. The HMI Tester will be able to communicate with the Preload Module and use its functionality as mentioned above.

4.1 Architecture Actors

In the OpenHMI-Tester architecture, two different roles or actors can be identified. The tester corresponds to the user that interacts with the OpenHMI-Tester environment. The tester also provides the application to be tested. The other one is called developer, whose responsibility is to write the code to adapt the OpenHMI-Tester to a given testing environment (implement colored boxes in Figure 1).

4.2 Data Model overview

The Data Model (Figure 2) is the data structure used to describe a set of test cases which can be performed over the tested application. This structure, which is divided into three levels (like other existing datamodels such as CppUnit [26] or jUnit [27]), can include all the necessary information to represent all the characteristics of a set of tests (test suite).

![Figure 2. Data Model.](image)

The Data Model is structured as follows:

- **Test Suite**: this element includes a set of test cases referring to the same application and usually with a common goal. It may also include meta information and a reference to the tested application.
- **Test Case**: this element describes a set of ordered test items to be performed on the tested application and it also may include meta information (e.g. test case description and purpose).
• **Test Item**: it is the smallest element in the data model description and represents a single action which can be performed on the tested application and its meta-information. In the *OpenHMI-Tester* a test item matches an event as described below in Subsection 4.3.

The fact that a complete description of a test suite is encapsulated in a single object eases other tasks and processes like, for instance, to dump a test suite description to a file or apply a filter to an entire test suite.

### 4.3 Events overview

Along the *OpenHMI-Tester* architecture will appear different types of events which can include different kinds of information. In this architecture, each event is represented by a Test Item object, in which the “type” and “subtype” values determine its nature.

These events are classified in four groups according to their purpose:

- **GUI Events**: events that contain information related to GUI elements (e.g. layout change events, mouse click event, etc.) These events normally are posted towards a single widget.
- **Non-GUI Events**: these events do not contain information related to the GUI and their elements, but may contain relevant information about the tested application (e.g. timer events, meta-calls events, etc.)
- **Meta Events**: events defined by the developer that implements actions which are not natively supported by the windowing system (e.g. messages in dialog boxes, pauses, sounds, etc.)
- **Control Events**: these events are also defined by the developer and are used in control signaling along the architecture (e.g. “execution process started” event).

### 4.4 HMI Tester architecture

*HMI Tester Module* is the software element with which the tester will interact and has two main functions: the first one involves controlling both test case recording (event capture) process and test case playback (event execution) process; the second one is to manage test suite lifecycle (e.g. create a new test suite, add new test cases, include received test items in current test case, etc.). It also provides a graphical user interface which allows the tester to perform the tasks mentioned above.

In order to perform control and keep track of event capture and execution pro-
cesses (explained below in sections 4.6 and 4.7), the HMI Tester communicates to the Preload Module by using sockets or an IPC mechanism.

As shown in Figure 3, the HMI Tester Module architecture is composed of a set of modules which include the functionality necessary to manage recording and playback processes, and a special module whose aim is to perform, depending on the operating system, the preloading process described at the beginning of this section. Also, another special submodule lets the developer adding his or her own representation of the Data Model to the architecture.

The most significant modules are described as follows:

- **Data Model Adapter**: to integrate into the OpenHMI-Tester architecture a custom representation of the data model, the developer has to implement his own Data Model Adapter submodule in order to provide Data Model Manager with the functionality needed to manage test suites lifecycle.
- **Comm module**: this module includes functionality related to communications. It will be used by both the Playback Control Module to send new events (GUI events and Control events) to the Preload Module, and the Recording Control Module to receive captured (and controlling) event data.
- **Recording Control module**: this module controls test case recording (capture) process by sending control signaling events to the Preload Module when necessary; it also manages (and stores in the current test case) event captured data received from the Preload Module.
- **Playback Control module**: this module controls test case playback (ex-
execution) process by sending GUI events to the Preload Module and, when necessary, including control signaling events to guide the process remotely.

- **Preloading Action module**: this module is intended to perform the preloading process on the operating system where the testing process is being done (so, it has to be adapted depending on the OS). The Preloading Process includes the establishment of the preloading library and then the launching of the tested application.

4.5 Preload Module architecture

The Preload Module is the software element which will hook up to the tested application in order to capture the generated events and post new events received from the HMI Tester. As we can see in Figure 4, it is composed of some modules that implement common behavior (e.g. communication to the HMI Tester, event data encoding, etc.) and other modules whose behavior changes depending on the windowing system.

![Figure 4. Preload Module detailed architecture.](image)

Modules that implement common behavior are the following:

- **Logic module**: its main task is the initialization process, in which all modules (Comm, Event Consumer and Event Executor) have to be created, installed and initialized in order to hook up the Preload Module to the tested application properly.
- **Comm module**: it works similarly as described in last subsection. In this case, it is used by the rest of modules to send data (e.g. captured events data) to the HMI Tester; it also delivers the received messages to the corresponding module (e.g. control events will be delivered to Logic module and...
normal events to Event Executor module).

As mentioned at the beginning of this section, in order to design an open architecture some modules have to be extended depending on the windowing system. These modules are the following:

- **Preloading Control module**: this module is responsible for detecting the application launching somehow and call the Logic initialization method. Since this module might use non-cross platform methods, it is probably that it also has to be extended depending on the operating system.

- **Event Consumer module**: this module captures generated events, manages the data contained in them and notify that a new event has been captured for it to be sent. This module should also implement a configuration method if the installation of one instance or any similar process has to be performed.

- **Event Executor module**: this module executes events received from the HMI Tester. Once a new event is received, it has to extract event data and post it into the application event system (or execute an equivalent action) which performs the requested behavior. This module should also implement a configuration method if needed.

### 4.6 Event Capture process

Capture Process is the process by which the HMI Tester gets events generated by the tested application. In order for the HMI Tester not to be intrusive, it uses the Preload Module, which captures events generated by the tested application by preloading some classes while launching the application to be
Capture process can be summarized in the following steps:

1. **Event Generation**: while the tester interacts with the GUI (e.g., a button clicked, a key pressed, etc.), it is generating events (GUI events and non-GUI events). Data included in these events has to be captured and sent to the **HMI Tester**.

2. **Event Capture and Control signaling**: on this phase, the **Preload Module** gets the events generated by the tested application and encapsulates their relevant data on new objects (Test Items) which will be sent to the **HMI Tester**. Control Signaling is performed on this phase too; the **Preload Module** may notify the **HMI Tester** the tested application state (e.g., the execution has finished) or other interesting information.

3. **Event Handle**: “Comm Module” in the **HMI Tester** notifies that a new Test Item has been received. If it is a control event it has to be handled; if not, it will be stored.

4. **Event Store and Control Event Handle**: the new Test Item is stored in the corresponding Test Case respecting its order of arrival unless it is a control event, then it will be handled.

### 4.7 Event Execution process

Execution process is the process by which the **HMI Tester** send stored (and control signaling) events to the **Preload Module**. When the **Preload Module** receives new events, it will post them into the tested application event system. These events describe the actions to be performed over the tested GUI.

![Figure 6. Event Execution process.](image-url)
Execution process may be described in these steps:

1. **Event Dispatching and Control Signaling**: events (Test Items) stored in the current Test Case are sent to the Preload Module; new control events could also be sent in order to notify about the process state (e.g., execution finished), actions to be taken (e.g., stop capturing events), etc.

2. **Event Pre-handle**: when a new event is received in the Preload Module, its type value is used to decide if the event is a GUI event (it has to be posted to the tested application) or a control event (handled by the Preload Module).

3. **Event Posting and Control Event Handle**: the Preload Module performs the received action (event) by posting a new system event (built from the received event data) into the application or by executing an equivalent action (e.g., call the click method in a GUI button). If the received event is a control event it has to be handled by the Preload Module.

4. **Event Handle**: posted events (or posted indirectly by the equivalent action performed by the Preload Module) arrive to the GUI event system and are fulfilled.

5. **Implementation**

As stated before, a few modules of the OpenHMI-Tester architecture have to be adapted to suit the windowing environment used by the tested application. The adaptation encompasses implementing just the specific behavior required to interact with that particular environment. By leveraging the common architecture, the adaptation modules thus allow the OpenHMI-Tester architecture to be flexible enough to support a wide range of existing operating and windowing systems.

This section shows some of the implementation details of the common functionality; then, we describe the implementation of an OpenHMI-Tester Prototype using the Qt toolkit on a X11 windowing system.

### 5.1 Common Functionality Implementation Details

#### 5.1.1 Generic Data Model Implementation

In OpenHMI-Tester, a data model is introduced to describe the different tests that can be performed to an application. As we can see in Figure 7, each event is identified as a test item. Test cases are introduced as a set of ordered test
items to be performed. Finally, a test suite represents a set of test cases for the application.

This Generic Data Model is flexible enough to represent all the data carried by the different types of events needed by the adaptation modules in a generic way. The data maps allow storing different data associated to each event; each element of the data model has also its own properties.

The adaptation modules, in turn, are responsible for converting the different events between the module’s internal representation and the generic representation, in order for them to be managed by the architecture.

5.1.2 Recording Process Implementation

The recording process refers to the process of capturing and storing the events produced during an interaction with the tested application.

In the OpenHMI-Tester Prototype the recording process is implemented using a series of control events to signal the start of the recording, pause, stop, etc. (Figure 8). During the process, the set of events produced by the application are stored in the actual test case.

Figure 7. Generic Data Model Hierarchy.

Figure 8. Control Signaling Events Hierarchy.
All the control signaling events are defined as derived of a generic ControlTestItem which also inherits from the generic test item. These control events are used in both recording (capture) and playback (execution) processes.

5.1.3 Playback Process Implementation

The playback process reproduces sequences of events captured in a previous recording process. The events are injected into the application as if a real human tester would be using it.

In the OpenHMI-Tester Prototype the playback process is implemented using the test items (events) stored in a test case. These events are sent from the HMI Tester to the Preload Module, which will handle them and will answer with a CTI_EventExecuted control event for each executed test item (the control event allows to synchronize the process.) Similar signaling events are used to indicate the start, stop, and pause of the playback process, etc.

5.2 Specific Functionality Implementation Details

The adaptation to a given windowing and operating environment requires a few specific code to be written and plugged into the architecture.

5.2.1 Data Model Adapter Implementation

The adaptation modules can define a test suite using the representation that better suits the specific environment. However, they have to provide code to convert the information of the test suite back and forth into the generic representation described in Subsection 5.1.1. This piece of code conforms the Data Model Adapters, which allow the Data Model Manager to manage the information of the test suites in a generic way.

In the OpenHMI-Tester Prototype, where it has been selected a XML description as the Test Suite representation, this problem has been dealt by using two methods: one method which creates a Test Suite object from a given file path by using a XML DOM Parser; another one which performs the opposite process (Test Suite to XML file) using a set of XML visitors.

---

4 The sequence of events stored in a Test Case could have been modified by an external editing tool.
5 A XML DOM Parser creates a data tree from a XML file.
6 A XML Visitor is capable of extracting the data from an object and returning the corresponding XML string.
5.2.2 Preloading Process Implementation

One of the key points of the OpenHMI-Tester architecture is the process of preloading a library that hooks into the unmodified application being tested and allows intercepting the flow of graphical events managed by the application. This process is divided into two steps:

First, the HMI Tester has to indicate the operating system that the preload library has to be loaded just before the tested application. The Preloading Action Module is in charge of executing this specific code that depends on the operating environment. On most UNIX systems, such as the one used in the prototype, the LD_PRELOAD environment variable is used to modify the normal behavior of the dynamic linker.

The second step is performed in the Preload Module whenever the application starts. The Preloading Control Module includes an interception function that filters the events received by the application. In the case of the combination of Qt under X11, the QWidget::x11Event call is captured.

5.2.3 Capture Process Implementation

The representation of the events by the adaptation modules is usually arranged in a hierarchical way. This event hierarchy can leverage the test item object included in the generic Data Model described in Section 5.1.1. Figure 9 shows the three-level event architecture chosen for the OpenHMI-Tester Prototype.

Once an event hierarchy has been defined, it is necessary to capture the events and send them to the HMI Tester module. The capture process has to be performed depending on the chosen windowing system (e.g. by installing event filters as in Qt [22] and in GTK+ [23], using event listeners, or using peer components as in Java [21]). In the prototype we use a Qt Event Filter which gets established at the preloading code initialization. Events of interest for the module are then redirected to the Event Consumer Module.
5.2.4 Execution Process Implementation

The process of executing (applying) the events received from the HMI Tester module is what we call the execution process.

In the OpenHMI-Tester Prototype, when a new event is received, the Event Executor Module classifies it using the “type” and “subtype” attributes. Once the type of the event is determined, and the generic test item is converted to its proper specific most-derived class, the event executor extracts the event data and performs the action required by that event.

Some of the functionality required by the events may be simulated. For instance, if the mouse movement is not relevant, a simulation of movement may be carried from the starting point to the destination point.

5.3 “Open HMI Tester” Prototype

To show the validity of our approach, as stated before, a prototype implementation has been developed following the architecture introduced in this paper. The prototype adapted the generic architecture to a specific environment consisting of:

- A Linux distribution as the operating system.
- Trolltech’s Qt4 toolkit [22] under X-Window as the windowing system.

As described in this section, this prototype implementation includes the basic functionality related to the event capture and execution processes.

It captures events from the tested application, uses them to build a Test Suite structure and stores this structure on a file by using a XML representation. The set of captured events includes a basic representation of mouse and keyboard events (e.g. mouse press, mouse double click, key press, etc.) and some window events (e.g. close window).

Also, it is capable of executing the events mentioned above. The execution process uses a Test Suite object built from a representation included in a XML file and allows the tester to choose among all available Test Cases to execute them. All the captured events can be performed on the tested application. This prototype also implements some extra functionality as, for instance, the mouse moving simulation.
5.3.1 Prototype technical specifications

This prototype has been written C++ and uses Qt library version 4.x. This prototype can be downloaded from [29].

![Calendar Widget](image)

Figure 10. Open HMI Tester prototype at work.

5.3.2 Prototype validation

In order to check the viability of the proposed architecture and implementation, the OpenHMI-Tester Prototype has been tested with some of the Qt available demo applications offered by Trolltech in [30]. The set of applications selected for testing include those with a rich GUI with many of the common widgets, as well as other special widgets (e.g. graphic panels, calendar widgets, etc.).

The first performance analysis obtained during the evaluation of the OpenHMI-Tester Prototype (Figure 10 belongs to one of the performed tests) presents a promising result; during the testing process the OpenHMI-Tester could capture the events generated during the tester interactions with the different GUIs and replayed them later by simulating keyboard and mouse events. Nevertheless, since the OpenHMI-Tester Prototype has been released quite recently, its implementation has to be refined in order to improve a few aspects of the simulation such as the drag-and-drop movement and other advanced actions which can be performed by a human tester.
6 Discussion

As mentioned in Section 2, the OpenHMI-Tester belongs to the approach in which no model or representation is built since test cases are generated directly over the area of the GUI of interest to the tester. The OpenHMI-Tester uses a capture and execution process which work as follows: once the application to be tested is launched, the tester performs a set of actions in the GUI which generate GUI events. These GUI events are automatically captured and stored to be used later to generate a test case that executes the actions performed by the tester. Generated test cases can be refined and completed by adding meta-events (e.g. messages, verification points, etc.). The process of capturing, replaying and analyzing executions is an example of Observation-based Testing [13].

6.1 Architecture

The OpenHMI-Tester architecture is fully portable since its definition is not linked to any operating system or any windowing system. The open architecture of the OpenHMI-Tester makes it agnostic to the windowing system (e.g. Qt or GTK), the event capture process (e.g. by using event listeners or peer components), the event filtering rules (e.g. capture only GUI events or also capture signaling events), the event execution technique (e.g. by sending captured events or by using a GUI Interaction API), etc.; the mix and match of features depend on the available implementation used by the developer at some point. Also, the developer may use his own implementation in order to have capture and execution processes under control. Adding these implementations to the architecture allows developer to control what events are captured and how, during the capture process, and also to select what events are going to be executed and how, during the execution process.

The architecture describes a lightweight method to automatically generate test cases. The developer do not need to build (and maintain) any model definition or GUI representation, as they only have to select the area to be tested (it might involve the whole GUI), launch the target application, and perform the actions corresponding to the test case. A new test case including all the actions performed in the last step will be automatically generated and it may be executed as many times as needed. Then, these test cases can be used as the application evolves to check that he expected functionality of the GUI application holds against application changes (i.e. replayed as regression tests).
6.2 Test Case Generation

In the other mentioned approaches, the test generation phase includes searching all possible test cases by traversing a GUI model or representation (e.g. DART [7,8] and GUI Model Driven Testing [4]). However, in the approach presented in this paper, the test case generation process is tester-guided since the tester is responsible for indicating, during the capture process, what widgets and actions are relevant in the application by performing actions within the GUI. The coverage criteria, then, is determined by the tester.

A tester-guided test case generation process allows the tester to focus testing efforts on the relevant widgets, widget properties and actions, and avoiding all those not interesting for the test. In most cases this leads to a smaller group of generated test cases. Sometimes, however, since we are describing a human-guided process, some test cases may be missing, causing that those widgets or actions to be out of the testing process. Robustness of the process would not be affected but it would be incomplete if those widgets and actions were on the testing plan; therefore the responsibility of creating a complete test suite falls on the tester.

6.3 Verification Process

In the approaches mentioned previously, the verification process involves generating a set of expected results by using a Test Oracle. These results are then compared with the obtained output after the test case execution.

In [1] Atif Memon introduces the use of test oracle tools in GUI testing. Since usually a test oracle compares the output with the expected results once the test case execution has finished, Memon describes that during GUI testing the GUI should be verified step by step, and not only at the end of the test case execution, because the final output may be correct but intermediate outputs might be incorrect.

Thus, in OpenHMI-Tester we describe two ways to perform verification process: the first way is as simple as a visual verification performed by the tester during a test case execution. The fact that the tester has to check if everything is done correctly during test case execution may be a very tedious process, so a semi-automated alternative is proposed. The OpenHMI-Tester allows the developer to implement their own verification method. We propose a verification process based on an on-demand introduction of meta-events called verifica-

---

7 A Test Oracle is a tool that generates expected results for a test case and compares them with the obtained results. It is usually invoked after the test case execution.
tion points which could verify one or more properties belonging to one or more GUI elements at a given time. During test case execution process these verification points would be executed to perform the requested verifications and the results may be reported to the tester at the end of the execution. These verification points may include, for example, expected values in text boxes or the background color of some other widget.

The fact that the test case generation and execution processes are performed on the software at execution time, causes that the actual application is acting on the GUI. So, all the modifications and restrictions stored both in the application code and in the widget properties are active, and are thus tested by the OpenHMI-Tester at execution time. This does not necessarily holds if the test case generation process is not performed at the real application execution time, as in [7] and [4], where the test cases are extracted from a model.

6.4 Modifications Tolerance, Robustness and Scalability

If a new element is added to the GUI, the tester can deal with the problem in two ways: by creating a new test case involving the new GUI object (a new test case would be added to the existing test suite) or by editing an old test case and adding the new actions to be performed (the structure of the test suite would not be modified). If a GUI element is removed, the tester can also deal with the problem in two ways: by replacing the test cases having actions over the deleted object by other test cases, or by deleting them from those test cases. Should the tester decide not to take any of the proposed solutions, the execution process can foresee the missing widgets and do not perform the corresponding actions.

This makes the OpenHMI-Tester architecture highly tolerant to change, since the tester can chose among several options to approach a GUI modification (or even let the system take care). The fact that no model has to be maintained also helps the test case generation process to be more tolerant to changes.

Another strong point in the OpenHMI-Tester architecture is robustness. The fact that human intervention is kept to a minimum during test case generation process, in which the tester only has to perform the actions that are going to be analysed later, increases the process robustness as the tester does not have to build, verify or complete any model or test case description. Moreover, the generated test cases can be edited to add meta-events or to remove existing events. Robustness is a very important feature all over the process, because keeping it during capture and test generation processes will let the system perform a better event execution process. In order not to put the process robustness at risk, the editing process should be performed by using an editing
tool which allows the tester to edit test cases in a safe way.

Scalability is another of the strengths of this architecture. When new elements or actions are added to the GUI, as commented a few paragraphs before, the tester may create a new test case involving those new elements or actions, or might simply edit an old test case and add the new actions to be performed. If the tester chooses the first option, the test suite (set of generated test cases) will increase linearly in the worst case, but if the tester opts for the second option, the test suite size will not be increased. The tester will be solely responsible for the amount of test cases that will compose the test suite, since a sufficiently high number of test cases has to be generated to test all the relevant functionality of the GUI.

6.5 Performance Analysis

Since the OpenHMI-Tester Prototype has been released quite recently, its performance has not yet been evaluated extensively in open-source frameworks. However, the first performance analysis obtained during the evaluation of the downloadable prototype (described in subsection 5.3) presents promising results.

During capture process (which implies catching events, handling and packaging event information and sending these events to the HMI Tester,) the observed behavior does not impose any problem or delay during both event handling and data transmission. Since delays are negligible, capture process is done at the same time the tester is performing the test over the tested application. During the execution process (which implies event sending, managing and executing,) the observed behavior is the same as the described before. The fact that relevant GUI events, in most cases, do not represent a half of the amount of available events, and that the event generation is performed on-demand, in response to the actions done by the tester, leads to an irregular data flow between the HMI Tester and the tested application, which in turn makes the architecture to work without any major efficiency problem.

The fact that no GUI model or representation has to be built allows the OpenHMI-Tester avoid tedious test case and oracle generation processes, which might delay the process with somehow complex GUIs. In the architecture described in this paper, both test case generation and execution processes are performed in real time, while the tester is interacting with the GUI or while the events are being sent to the Preload Module respectively.

Nonetheless, the performance of the architecture might be compromised if the developer does not use an efficient implementation of the event handling in both capture and execution processes. If the implementation chosen by
the developer does not perform an acceptable event filtering during capture process, and is not able to rule out unsuitable events (e.g. no GUI events, application timing events, object hierarchy events, etc.), it could lead to the emergence of bottlenecks during event handling and data transmission.

Another performance limitation could be that a capture/replay tool may spend a lot of efforts trying to replay executions faithfully, but it cannot execute them with complete fidelity [14]. The execution environment may have different features (e.g. open windows, screen resolution, system memory, CPU speed, etc.) than the capture environment, and to store all the environment features and all the performed events would be inconceivable. In most cases it poses no problem because a small set of GUI events is enough to simulate tester actions, and environment configuration does not matter (an application should work fine on a wide range of environment configurations).

7 Conclusions and Future Work

The Human-Machine Interface (HMI) of any software represents the means, in terms of inputs and outputs, by which users interact with that software. Although it usually tends to be as simple as possible, in medium and large projects, specially in the industry, the HMI used to control a given system or platform takes usually a significative part of the design and development time. In fact, it requires of specialised tools to be developed and tested. Although plenty of tools exist to assist with the design and implementation, testing-support tools are not that frequent, specially in the open-source community. Additionally, testing platforms in use for other parts of the software are not directly applicable to HMI. The development of such systems would help reducing the time needed to develop a software product, as well as providing robustness and increasing the level of usability of the final product. In this context, this paper provides the definition of a general and open HMI testing architecture named OpenHMI-Tester and the details of an open-source implementation, whose requirements have been driven by industrial HMI applications, and which has been also tested with real scenarios and applications.

As a statement of direction, we are currently working on the implementation of the adapting modules for different windowing systems, and performing extensive performance measurements in real scenarios. We are also working on the design and latter implementation, also as another contribution to the open-source community, of an editor which helps any user of the OpenHMI-Tester architecture to create any meta-action that can be of certain interest in a given scenario. This tool will also help editing any currently existing test and adapt it as needed.
Acknowledgements

This paper has been partially funded by the Catedra SAES of the University of Murcia initiative, which is a joint effort between SAES (Sociedad Anónima de Electrónica Submarina, http://www.electronica-submarina.com/) and the University of Murcia to work on open-source software, and real-time and critical information systems.

References


